Apache Kafka – 360 View

Apache Kafka – 360 View

Course Outline

- ❑ Course Introduction
- ❑ Apache Kafka Introduction
- ❑ Low-Level Architecture
- ❑ Advanced Kafka Producers and Consumers
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Chapter 1

Course Introduction

Apache Kafka – 360 View

Course Outline

- ❑ **Course Introduction**
- ❑ Introduction to Data Ingestion
- ❑ Apache Kafka Introduction
- ❑ Low-Level Architecture
- ❑ Advanced Kafka Producers and Consumers
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Chapter Outline

- ❑ **About this Course**
- ❑ Course Logistics
- ❑ Course Agreements
- ❑ About you !
- ❑ About the Instructor
- ❑ General Instructions for Exercises

Course Introduction

In This Course

In this course you will learn about

- ❑ Basics of Data Ingestion and tools
- ❑ High Level Introduction to Apache Kafka and components involved and required for kafka
- ❑ Setup of Multi-Broker Kafka cluster
- ❑ Ingesting data with Kafka Producers and Consumers – Hands-on
- ❑ Low Level Apache Kafka Architecture
- ❑ Advanced Kafka Producers and Consumers – Hands-on
- ❑ Schema Management in kafka – Hands-on
- ❑ Kafka Security – Theory
- ❑ Kafka Disaster Recovery – Mirror Making
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy with Hands-on
- ❑ Kafka Connect with Hands-on
- ❑ Kafka Streams with Hands-on

Chapter Outline

- ☑ About this Course
- ❑ **Course Logistics**
- ❑ Course Agreements
- ❑ About you !
- ❑ About the Instructor
- ❑ General Instructions for Exercises

Course Logistics

- ❑ Course start and end times
- ❑ Break
- ❑ Lunch
- ❑ Ask Questions
- ❑ Restrooms

Chapter Outline

- ✓ About this Course
- ✓ Course Logistics
- **Course Agreements**
- About you !
- About the Instructor
- General Instructions for Exercises

Course Agreements

- ❑ Mobile should be on silent mode
- ❑ Ask question as soon as you have it
- ❑ Only one conversation in room at a time during the training
- ❑ Keep me informed if getting late
- ❑ Please maintain eye contact and be with me.
- ❑ Have proper sleep during training days so that you don’t sleep during training hours :)

Chapter Outline

- ☑ About this Course
- ☑ Course Logistics
- ☑ Course Agreements
- ☑ **About you !**
- ☐ About the Instructor
- ☐ General Instructions for Exercises

Introduction

About you !

- ❑ Name
- ❑ Company Name
- ❑ Role
- ❑ Total Experience
- ❑ Any Experience with SQL
- ❑ Any Experience with Linux Administration
- ❑ Any Experience with Big Data Technologies
- ❑ Expectations from this Training

Chapter Outline

- [x] About this Course
- [x] Course Logistics
- [x] Course Agreements
- [x] About you !
- [x] **About the Instructor**
- [ ] General Instructions for Exercises

About the Instructor

- ❏ Industry Experience
- ❏ Experience with various Technologies
- ❏ Projects Experience
- ❏ Trainings Imparted
- ❏ Certifications

Chapter Outline

- [x] About this Course
- [x] Course Logistics
- [x] Course Agreements
- [x] About you !
- [x] About the Instructor
- [x] **General Instructions for Exercises**

General Instructions for Exercises

- The training course is intended for providing the complete understanding about the Apache Kafka – Development and Admininstration. Also this involves the Kafka tools which supports for Data ingestions

Chapter 2

Introduction to Data Ingestion

Course Outline

- ❑ Course Introduction
- ❑ **Introduction to Data Ingestion**
- ❑ Apache Kafka Introduction
- ❑ Low-Level Architecture
- ❑ Advanced Kafka Producers and Consumers
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Introduction to Data Ingestion

- ❏ **What is Data Ingestion?**
- ❏ Lambda Architecture
- ❏ Types of Data Ingestion
- ❏ Data Ingestion parameters
- ❏ Good Practice in Data Ingestion

Introduction to Data Ingestion

What is Data Ingestion?

- Data Ingestion is the process of accessing and importing data for immediate use or storage in a database.
- When you work with big data, you are always receiving a lot of data in diverse formats thought the sources, so you have to organize how to filter this data and literally “ingest” in order to generate information in a organized way.
- This layer is the first step for the data coming from variable sources to start its journey. Data here is prioritized and categorized which makes data flow smoothly in further layers.

Introduction to Data Ingestion

Data Ingestion

Data are getting created without bounds :

- Financial Transactions
- Sensor Networks
- Server Logs
- e-Mails and Text Messages
- Social Media
- Machine Feeds

And we are generating data faster than ever

- Automation
- Faster Internet Connectivity
- User-Generated Contents
- IoT
- Bots

Introduction to Data Ingestion

Data Ingestion

Twitter processes more than 500+ million tweets per day

Facebook users generate 7 billion comments and “Likes”

YouTube gets 30 Million users per day

5 Billion videos are watched in Youtube a day

Every Minute Amazon makes $283,000 sales

Google gets 3.5 Million searches every minute

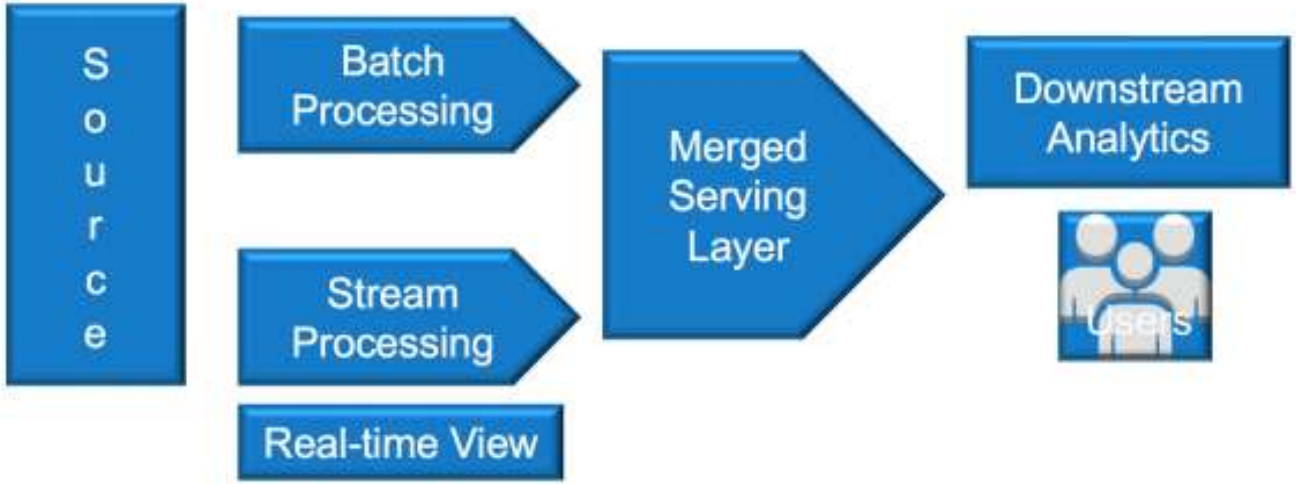
400,000 new tweets every minute

Introduction to Data Ingestion

- ☑ What is Data Ingestion?
- ❑ **Lambda Architecture**
- ❑ Types of Data Ingestion
- ❑ Data Ingestion parameters
- ❑ Good Practice in Data Ingestion

Introduction to Data Ingestion

Lambda Architecture



Introduction to Data Ingestion

Lambda Architecture

- **Batch Layer**
  - The Batch Layer manages the master data and precomputes the batch views.
- **Speed layer - Streaming**
  - The Speed Layer serves recent data only and increments the real-time views.
- **Serving Layer**
  - The Serving Layer is responsible for indexing and exposing the views so that they can be queried.

Introduction to Data Ingestion

- ☑ What is Data Ingestion?
- ☑ Lambda Architecture
- ❑ **Types of Data Ingestion**
- ❑ Data Ingestion parameters
- ❑ Good Practice in Data Ingestion

Introduction to Data Ingestion

Types of Data Ingestion

❑ **Batch Data Ingestion**

- Batch processing is most often used when dealing with very large amounts of data, and/or when data sources are legacy systems.
- Batch processing works well in situations where you don't need real-time analytics results

- **Use Cases**
  - Load data from Application DB to Warehouse in timeframe

❑ **Near Real-Time Data Ingestion**

- Near real-time processing is when speed is important, but processing time in minutes is acceptable in lieu of seconds.

- **Use Cases**
  - Recommendation Engines
  - Event Partitioning Engines

Introduction to Data Ingestion

Types of Data Ingestion

- **Real-Time or Streaming**

  - Real time processing requires a continual input, constant processing, and steady output of data.

  - A great example of real-time processing is data streaming, radar systems, customer service systems, and bank ATMs, where immediate processing is crucial to make the system work properly.

- **Use Cases**

  - Fraud Detections in Banking
  - Alert Mechanism
  - IoT

Introduction to Data Ingestion

- ☑ What is Data Ingestion?
- ☑ Lambda Architecture
- ☑ Types of Data Ingestion
- ❑ **Data Ingestion parameters**
- ❑ Good Practice in Data Ingestion

Introduction to Data Ingestion

Data Ingestion Parameters

- **Data Velocity**

  - Data Velocity deals with the speed at which data flows in from different sources like machines, networks, human interaction, media sites, social media. The movement of data can be massive or continuous.

- **Data Size**

  - Data size implies enormous volume of data. Data is generated from different sources that may increase timely.

- **Data Frequency**

  - Data can be processed in real time or batch, in real time processing as data received on same time, it further proceeds but in batch time data is stored in batches, fixed at some time interval and then further moved.

Introduction to Data Ingestion

Data Ingestion Parameters

❑ Data Format

- Data can be in different formats, mostly it can be the structured format, i.e., tabular one or unstructured format, i.e., images, audios, videos or semi-structured, i.e., JSON files, CSS files, etc

❑ Network Bandwidth

- Data Pipeline must be able to compete with business traffic. Sometimes traffic increases or sometimes decreases, so Network bandwidth scalability is biggest Data Pipeline challenge.

❑ Heterogeneous Technologies and System

- Tools must provide data serialization format, that means as data comes in the variable format so converting them into single format will provide an easier view to understand or relate the data

Introduction to Data Ingestion

- ☑ What is Data Ingestion?
- ☑ Lambda Architecture
- ☑ Types of Data Ingestion
- ☑ Data Ingestion parameters
- ❑ **Good Practice in Data Ingestion**

Introduction to Data Ingestion

Good Practices in Data ingestion

- ❑ Do not create Change data capture for smaller tables; this would create more problem at a later stage.
- ❑ Reduce time required to develop & implement pipelines
- ❑ Create Raw Zone, Staging Zone, Trusted Zone and Refined Zone
- ❑ Prepare the Timeline
- ❑ Create Dataflow for Views refresh
- ❑ Isolate Batch and streaming

Introduction to Data Ingestion

- ☑ What is Data Ingestion?
- ☑ Lambda Architecture
- ☑ Types of Data Ingestion
- ☑ Data Ingestion parameters
- ☑ Good Practice in Data Ingestion

Course Outline

- ✅ Course Introduction
- ✅ Introduction to Data Ingestion
- ❑ **Apache Kafka Introduction**
- ❑ Low-Level Architecture
- ❑ Advanced Kafka Producers and Consumers
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Apache Kafka Introduction

- ❏ **Why Apache Kafka**
- ❏ Apache Kafka Architecture
- ❏ Overview of Key Concepts
- ❏ Apache Zookeeper
- ❏ Cluster, Nodes and Kafka Brokers
- ❏ Kafka Topic and Kafka APIs
- ❏ Kafka partitions, Records and Keys
- ❏ Consumers and Producers
- ❏ Kafka Logs
- ❏ Kafka Partitions for Write Throughput
- ❏ Partitions for Consumer Parallelism
- ❏ Replicas, Followers and Leaders
- ❏ Disaster Recovery – High Level
- ❏ High Water Mark
- ❏ Consumer Load Balancing
- ❏ Fail-over

Apache Kafka Introduction

Why Apache Kafka

- Kafka is a distributed, scalable, fault-tolerant, publish-subscribe messaging system that enables you to build distributed applications
- Kafka was developed around 2010 at LinkedIn.
- The problem they originally set out to solve was low-latency ingestion of large amounts of event data from the LinkedIn website and infrastructure into a lambda architecture that harnessed Hadoop and real-time event processing systems.
- Real-time systems such as the traditional messaging queues (think ActiveMQ, RabbitMQ, etc.) have great delivery guarantees and support things such as transactions, protocol mediation, and message consumption tracking, but they are overkill for the use case LinkedIn had in mind.

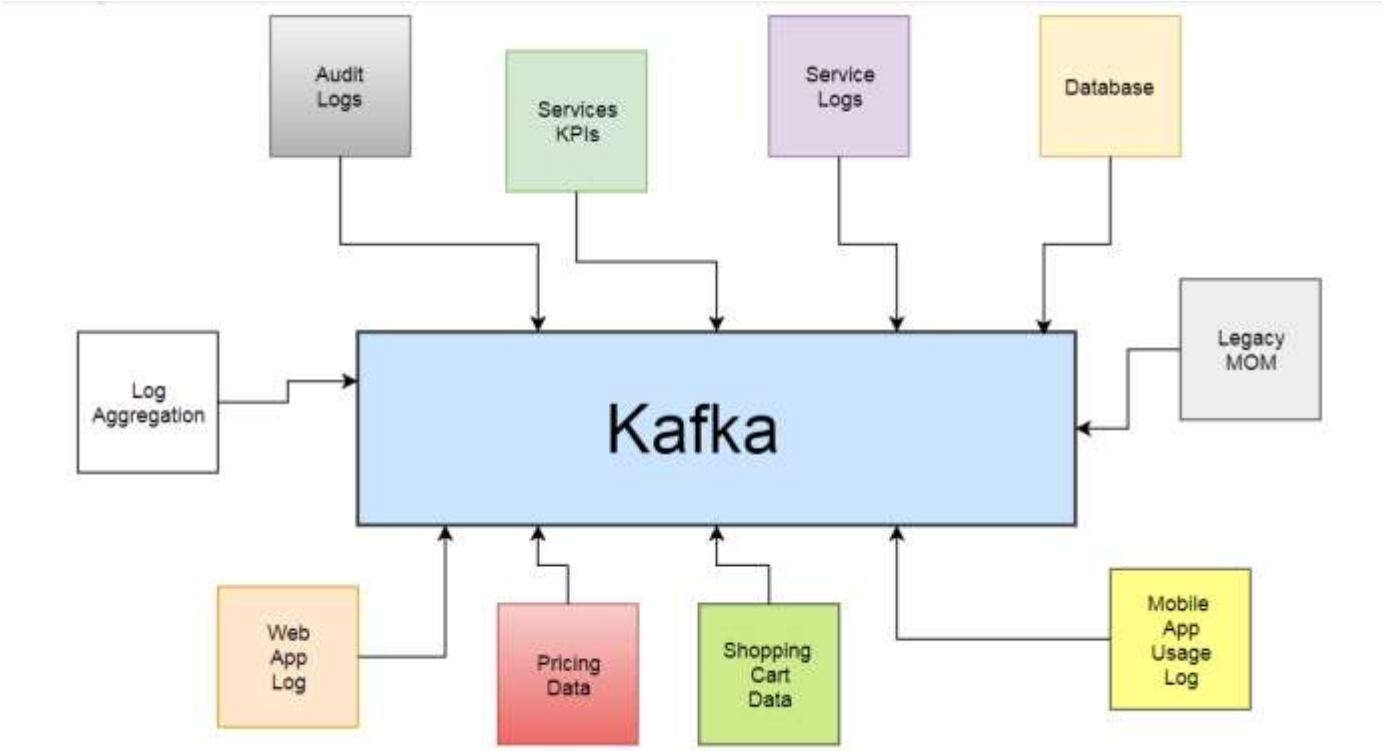
Apache Kafka Introduction

Why Apache Kafka

- Kafka was developed to be the ingestion backbone for this type of use case.
- Kafka was ingesting more than 1 billion events a day.
- LinkedIn has reported ingestion rates of 1 trillion messages a day.
- Kafka looks and feels like a publish-subscribe system that can deliver in-order, persistent, scalable messaging. It has publishers, topics, and subscribers.
- Kafka can also partition topics and enable massively parallel consumption
- All messages written to Kafka are persisted and replicated to peer brokers for fault tolerance, and those messages stay around for a configurable period of time

Apache Kafka Introduction

Why Apache Kafka

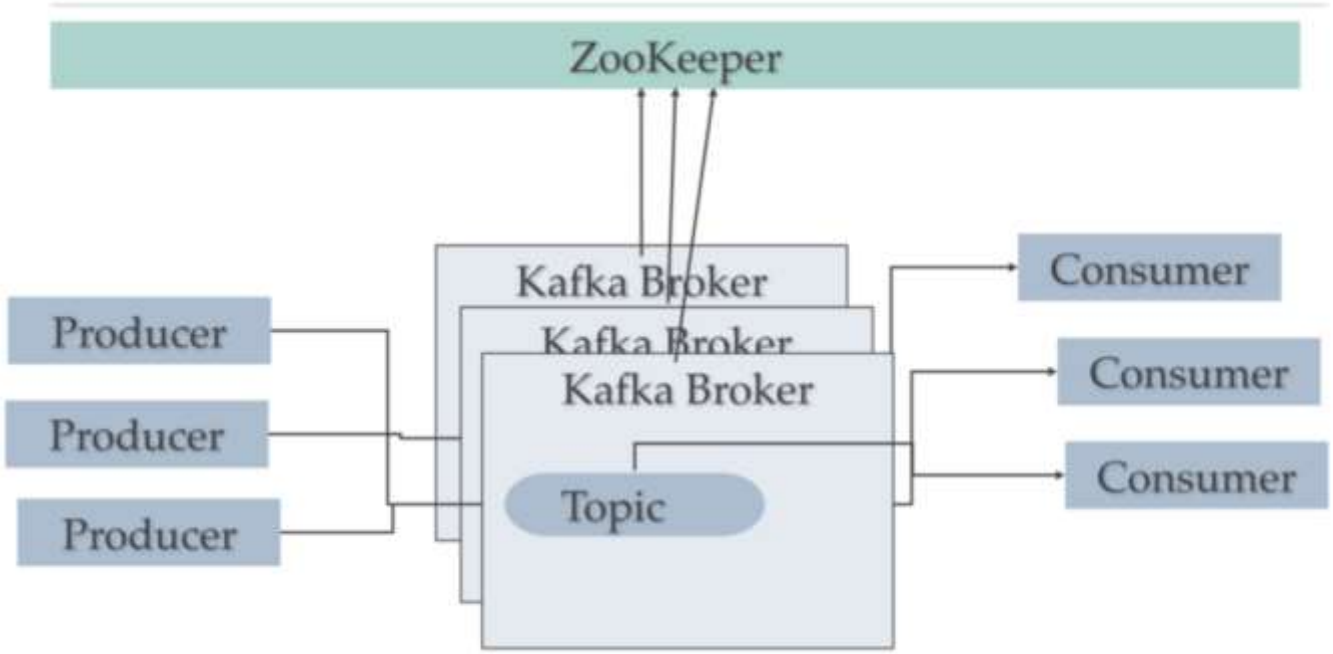


Apache Kafka Introduction

- ✓ Why Apache Kafka
- **Apache Kafka Architecture**
- Overview of Key Concepts
- Apache Zookeeper
- Cluster, Nodes and Kafka Brokers
- Kafka Topic and Kafka APIs
- Kafka partitions, Records and Keys
- Consumers and Producers
- Kafka Logs
- Kafka Partitions for Write Throughput
- Partitions for Consumer Parallelism
- Replicas, Followers and Leaders
- Disaster Recovery – High Level
- High Water Mark
- Consumer Load Balancing
- Fail-over

Apache Kafka Introduction

Apache kafka Architecture



Apache Kafka Introduction

Apache kafka Architecture

- Kafka consists of Records, Topics, Consumers, Producers, Brokers, Logs, Partitions, and Clusters. Records can have key (optional), value and timestamp.
- Kafka Records are immutable
- A Kafka Topic is a stream of records
- A topic has a Log which is the topic’s storage on disk.
- A Topic Log is broken up into partitions and segments.
- The Kafka Producer API is used to produce streams of data records

Apache Kafka Introduction

Apache kafka Architecture

- The Kafka Consumer API is used to consume a stream of records from Kafka.
- A Broker is a Kafka server that runs in a Kafka Cluster.
- Kafka Brokers form a cluster
- The Kafka Cluster consists of many Kafka Brokers.
- Kafka uses ZooKeeper to manage the cluster

Apache Kafka Introduction

Apache kafka Architecture

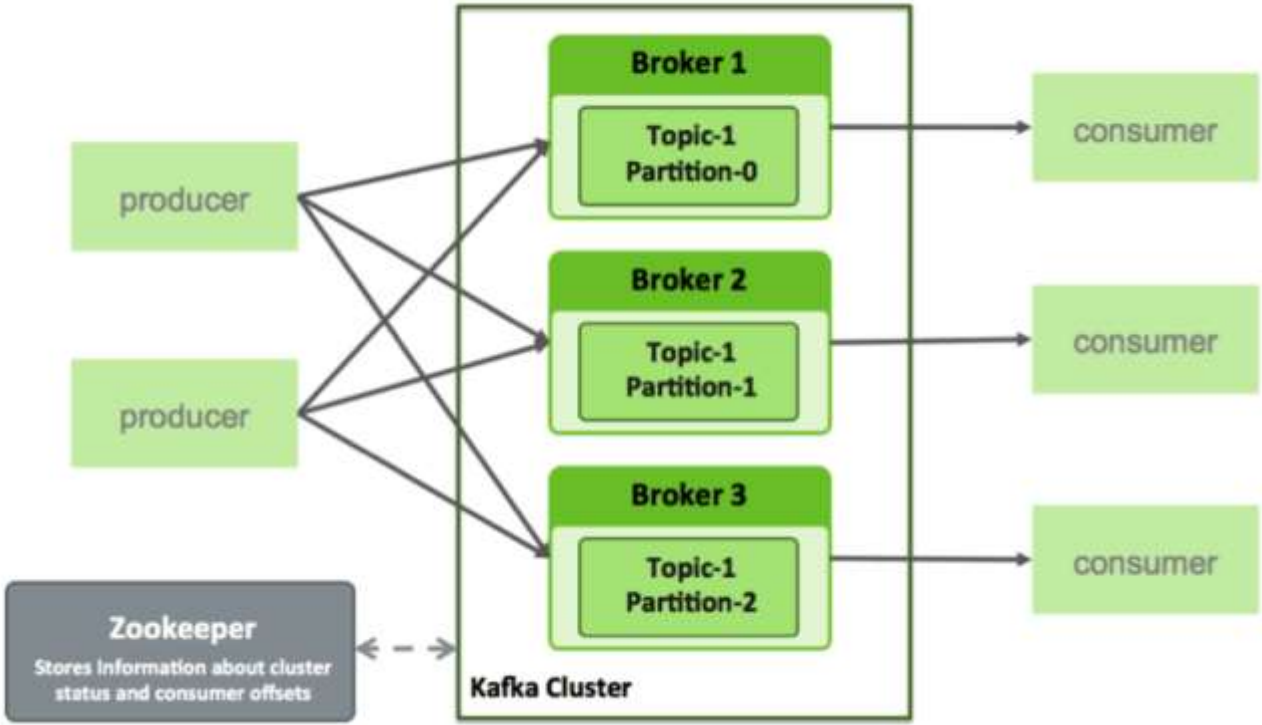
- Kafka communication from clients and servers uses a wire protocol over TCP that is versioned and documented.
- Kafka promises to maintain backwards compatibility with older clients, and many languages are supported.
- Kafka is fast because it avoids copying buffers in-memory (Zero Copy), and streams data to immutable logs instead of using random access.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ❑ **Overview of Key Concepts**
- ❑ Apache Zookeeper
- ❑ Cluster, Nodes and Kafka Brokers
- ❑ Kafka Topic and Kafka APIs
- ❑ Kafka partitions, Records and Keys
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Kafka Introduction

Overview of key Concepts



Apache Kafka Introduction

Overview of key Concepts

- **Topics**

  - Kafka maintains feeds of messages in categories called topics. Each topic has a user-defined category (or feed name), to which messages are published.

  - For each topic, the Kafka cluster maintains a structured commit log with one or more partitions

Apache Kafka Introduction

Overview of key Concepts

- **Partitions**

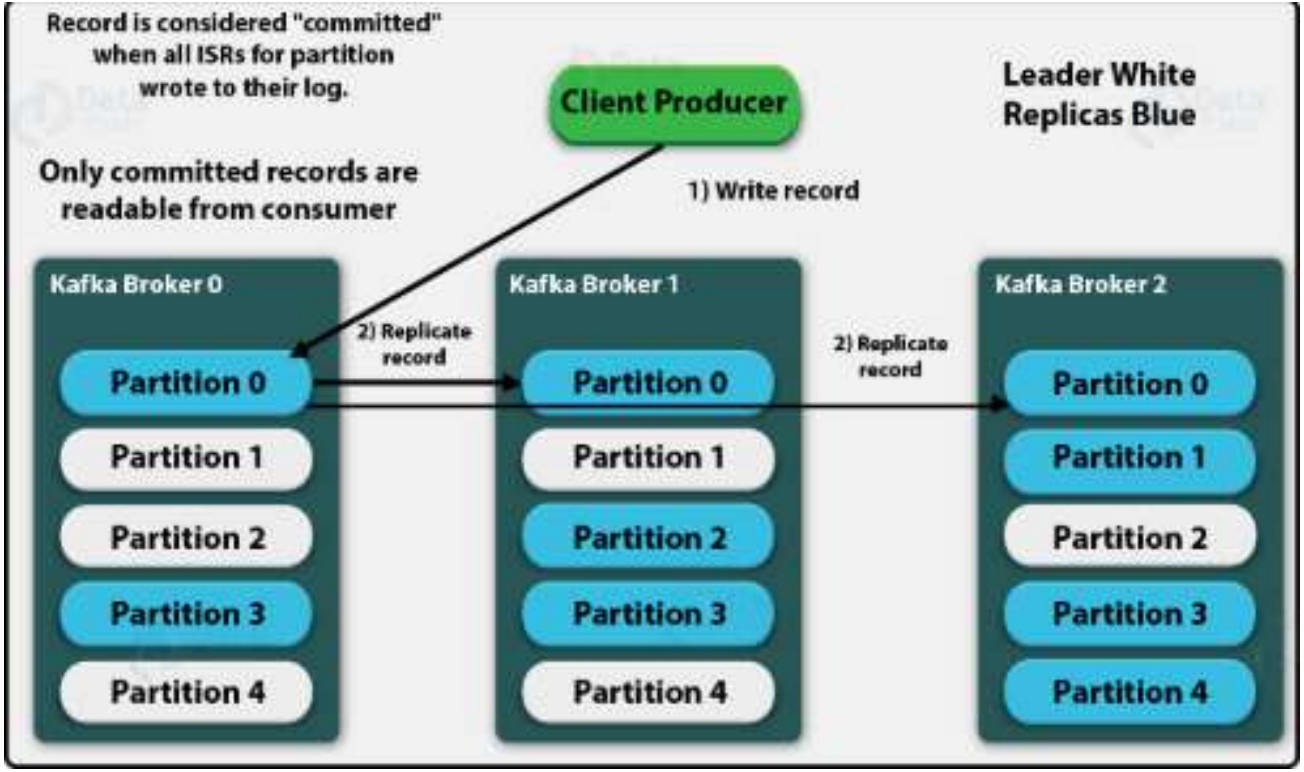
  - Kafka appends new messages to a partition in an ordered, immutable sequence.

  - Each message in a topic is assigned a sequential number that uniquely identifies the message within a partition which is called offset.

  - Partition support for topics provides parallelism. In addition, because writes to a partition are sequential, the number of hard disk seeks is minimized. This reduces latency and increases performance.

Apache Kafka Introduction

Overview of key Concepts



Apache Kafka Introduction

Overview of key Concepts

- **Producers**

  - Producers are processes that publish messages to one or more Kafka topics.

  - The producer is responsible for choosing which message to assign to which partition within a topic.

  - Assignment can be done in a round-robin fashion to balance load, or it can be based on a semantic partition function.

Apache Kafka Introduction

Overview of key Concepts

- **Consumer**

  - Consumers are processes that subscribe to one or more topics and process the feeds of published messages from those topics.

  - Kafka consumers keep track of which messages have already been consumed by storing the current offset. Because Kafka retains all messages on disk for a configurable amount of time, consumers can use the offset to rewind or skip to any point in a partition.

Apache Kafka Introduction

Overview of key Concepts

- **Broker**

  - A Kafka cluster consists of one or more servers, each of which is called a broker.

  - Producers send messages to the Kafka cluster, which in turn serves them to consumers. Each broker manages the persistence and replication of message data.

  - Kafka Brokers scale and perform well in part because Brokers are not responsible for keeping track of which messages have been consumed. Instead, the message consumer is responsible for this. This design feature eliminates the potential for back-pressure when consumers process messages at different rates.

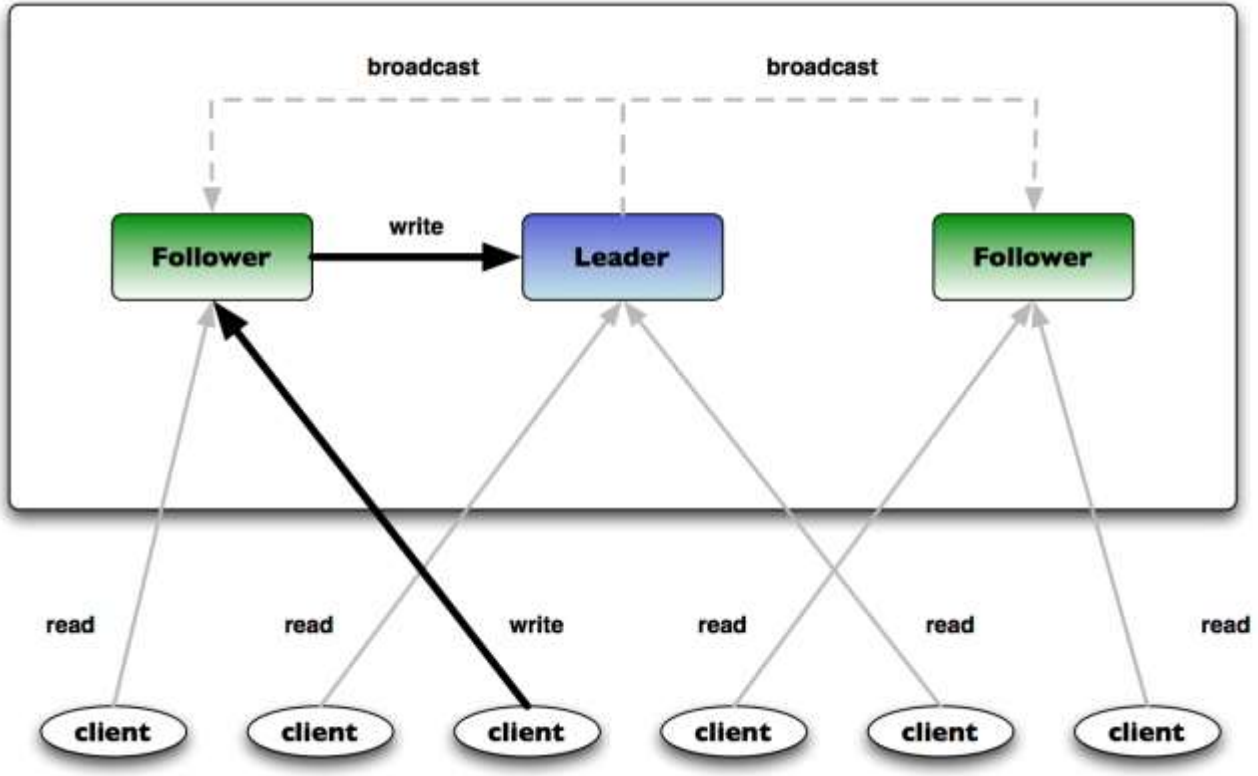
Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ❑ **Apache Zookeeper**
- ❑ Cluster, Nodes and Kafka Brokers
- ❑ Kafka Topic and Kafka APIs
- ❑ Kafka partitions, Records and Keys
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Zookeeper

- Apache ZooKeeper is an open-source project which deals with maintaining configuration information, naming, providing distributed synchronization, group services for various distributed applications.
- Zookeeper implements various protocols on the cluster so that the applications need not to implement them on their own.
- Developed originally at Yahoo, ZooKeeper facilitates synchronization among the process by maintaining a status on ZooKeeper servers that stores information in local log files.
- The Zookeeper servers are capable of supporting a large Hadoop cluster. Each client machine communicates to one of the servers to retrieve information.

Apache Zookeeper



Apache Kafka Introduction

Apache Zookeeper

- Zookeeper server is replicated over a set of machines
- All machines stores a copy of the data(in-memory)
- A leader is elected on service start-up
- Client connects only to a single zookeeper server and maintains a TCP connection
- Clients can read from any zookeeper server, writes go through the leader and needs majority consensus.
- Read Requests are processed locally at the zookeeper server to which the client is currently connected.
- Write Requests are forwarded to the leader and go through majority consensus before a response is generated

Apache Zookeeper - Overview

- Client
  - One of the nodes in our distributed application cluster, access information from the server. For a particular time interval, every client sends a message to the server to let the sever know that the client is alive.
  - Similarly, the server sends an acknowledgement when a client connects. If there is no response from the connected server, the client automatically redirects the message to another server.

- Server
  - Server, one of the nodes in our ZooKeeper ensemble, provides all the services to clients. Gives acknowledgement to client to inform that the server is alive

- Ensemble
  - Group of ZooKeeper servers. The minimum number of nodes that is required to form an ensemble is 3.

- Leader
  - Server node which performs automatic recovery if any of the connected node failed. Leaders are elected on service startup

- Follower
  - Server node which follows leader instruction.

Apache Zookeeper Use Cases

- Configuration Management
  - Cluster member nodes bootstrapping configuration from a centralized source.
  - Easier, simpler deployment/provisioning
- Distributed Cluster Management
  - Node Join/Leave
  - Node status in real-time
- Naming Services – DNS
- Distributed syncronization – Locks, barriers and queues
- Leader election in Distributed system
- Centralized and highly reliable data registry

Apache Zookeeper Consistency

- Sequential Consistency – Updates are applied in order
- Atomicity – Updates either succeed of fail
- Single System Image – A Client sees the same view of the service regardless of the ZK server it connects to.
- Reliability – Updates persists once applied, till overwritten by some clients
- Timeliness – The client's view of the system is guaranteed to be up-to-date within a certain time bound.

Apache Kafka Introduction

Apache Zookeeper Features

- Naming Services
  - Identifying ZooKeeper attaches a unique identification to every node which is quite similar to the DNA that helps identify it.

- Updating the Node's status
  - Apache Zookeeper is capable of updating every node which allows it to store updated information about each node across the cluster.

- Managing the cluster
  - Able to manage the cluster in such a way that the status of each node is maintained in real-time leaving lesser chances for errors and ambiguity

- Automatic failure recovery
  - Apache ZooKeeper locks the data while modifying which helps the cluster to recover it automatically if a failure occurs in the database.

Apache Kafka Introduction

Apache Zookeeper Features

- Naming Services
  - Identifying ZooKeeper attaches a unique identification to every node which is quite similar to the DNA that helps identify it.

- Updating the Node's status
  - Apache Zookeeper is capable of updating every node which allows it to store updated information about each node across the cluster.

- Managing the cluster
  - Able to manage the cluster in such a way that the status of each node is maintained in real-time leaving lesser chances for errors and ambiguity

- Automatic failure recovery
  - Apache ZooKeeper locks the data while modifying which helps the cluster to recover it automatically if a failure occurs in the database.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ❑ **Cluster, Nodes and Kafka Brokers**
- ❑ Kafka Topic and Kafka APIs
- ❑ Kafka partitions, Records and Keys
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

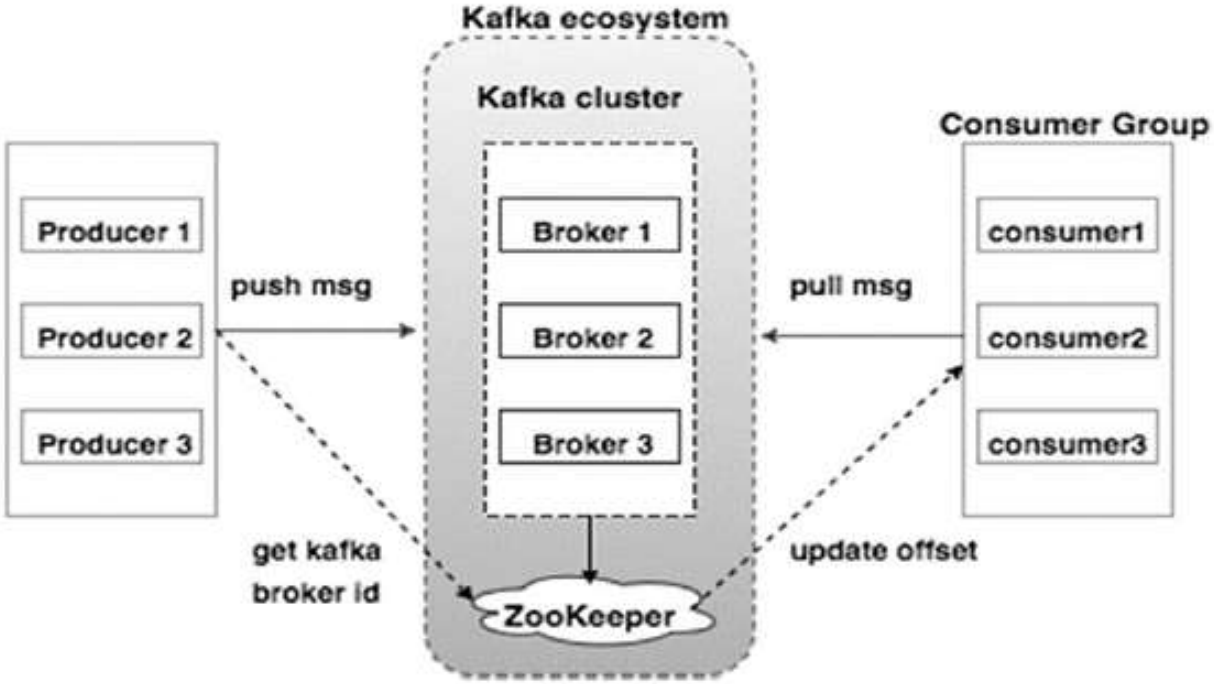
Apache Kafka Introduction

Kafka Cluster

- A collection of Kafka broker forms the cluster.
- One of the brokers in the cluster is designated as a controller, which is responsible for handling the administrative operations as well as assigning the partitions to other brokers.
- The controller also keeps track of broker failures.
- Kafka uses Apache ZooKeeper as the distributed configuration store. It forms the backbone of Kafka cluster that continuously monitors the health of the brokers.
- When new brokers get added to the cluster, ZooKeeper will start utilizing it by creating topics and partitions on it.

Apache Kafka Introduction

Kafka Nodes and Broker



Apache Kafka Introduction

Kafka Nodes and Broker

- A node is a single computer in the Apache Kafka cluster.
- Each node in the cluster is called a Kafka *broker*.
- Kafka cluster typically consists of multiple brokers to maintain load balance.
- Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state.
- One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact.
- Kafka broker leader election can be done by ZooKeeper.
- A Kafka broker receives messages from producers and stores them on disk keyed by unique **offset**.
- A Kafka broker allows consumers to fetch messages by topic, partition and offset

Apache Kafka Introduction

Kafka Nodes and Broker

- Each Kafka Broker has a unique ID (number)
- Kafka Brokers contain topic log partitions
- Connecting to one broker bootstraps a client to the entire Kafka cluster.
- For failover, you want to start with at least three to five brokers.
- A Kafka cluster can have, 10, 100, or 1,000 brokers in a cluster if needed.

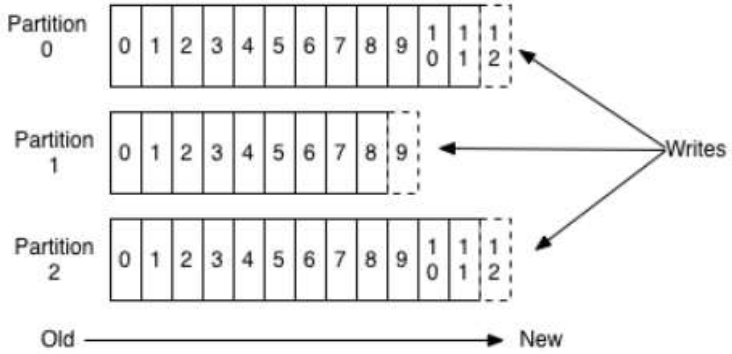
Apache Kafka Introduction

- ✓ Why Apache Kafka
- ✓ Apache Kafka Architecture
- ✓ Overview of Key Concepts
- ✓ Apache Zookeeper
- ✓ Cluster, Nodes and Kafka Brokers
- ❑ **Kafka Topic and Kafka APIs**
- ❑ Kafka partitions, Records and Keys
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Kafka Introduction

Kafka Topic

- Kafka provides for a stream of records—the topic
- A topic is a category or feed name to which records are published.
- Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.
- For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Apache Kafka Introduction

Kafka Topic

- Lets say a Bank with customer 360 capturing events from various side of the business using kafka

  - Create separate topic for the customer personal events
  - Create separate topic for payments related events
  - Create another topic for credit card related events.

- By this way you can isolate the data flow.

- We can also isolate the real-time, batch and near real-time data flows using topics

Apache Kafka Introduction

Kafka APIs

- Kafka Producer API
  - The Producer API allows an application to publish a stream of records to one or more Kafka topics.

- Kafka Consumer API
  - The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.

- Kafka Stream API
  - The Streams API allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

- Kafka Connector API
  - The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table

Apache Kafka Introduction

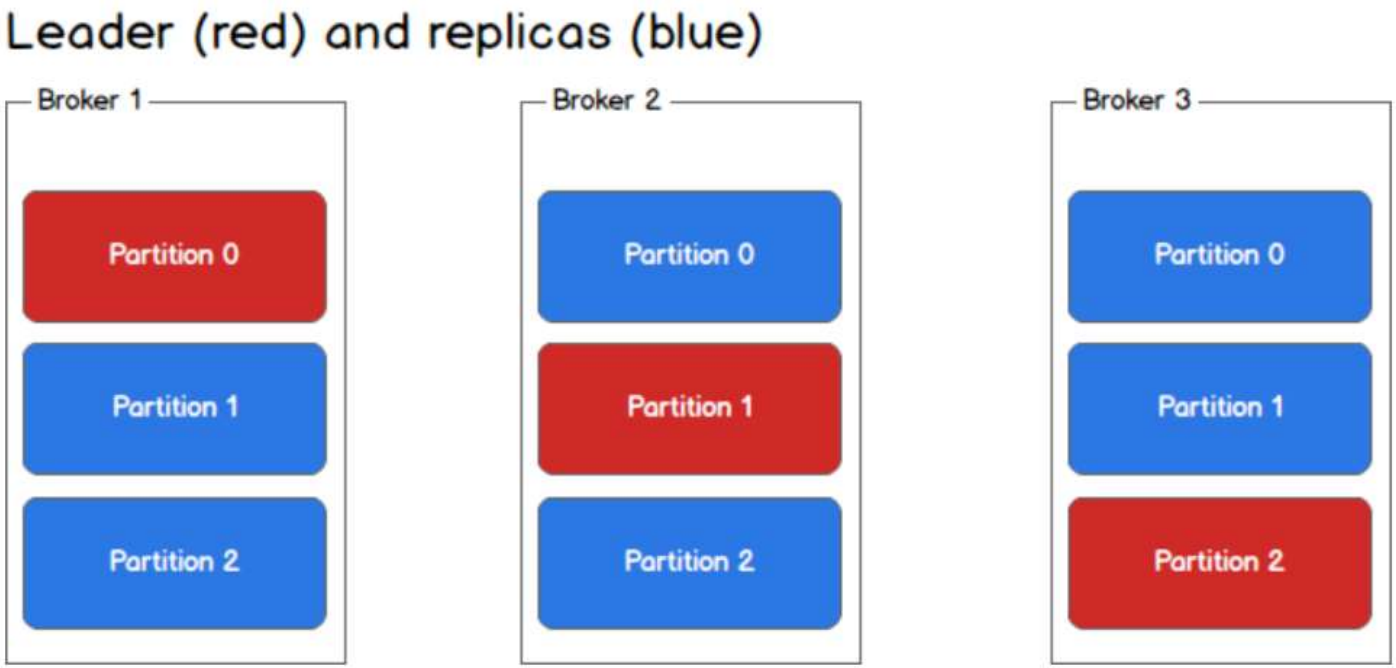
- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ❑ **Kafka partitions, Records and Keys**
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Kafka Introduction

Kafka Partitions

- Kafka can replicate partitions across a configurable number of Kafka servers which is used for fault tolerance.
- Each kafka broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic.
- Each partition has a leader server and zero or more follower servers.
- All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data.
- If a leader fails, a replica takes over as the new leader with the help of zookeeper
- Kafka also uses partitions for parallel consumer handling within a group.
- Kafka distributes topic log partitions over servers in the Kafka cluster. Each server handles its share of data and requests by sharing partition leadership.

Kafka Partitions



Kafka Record – Key value pair

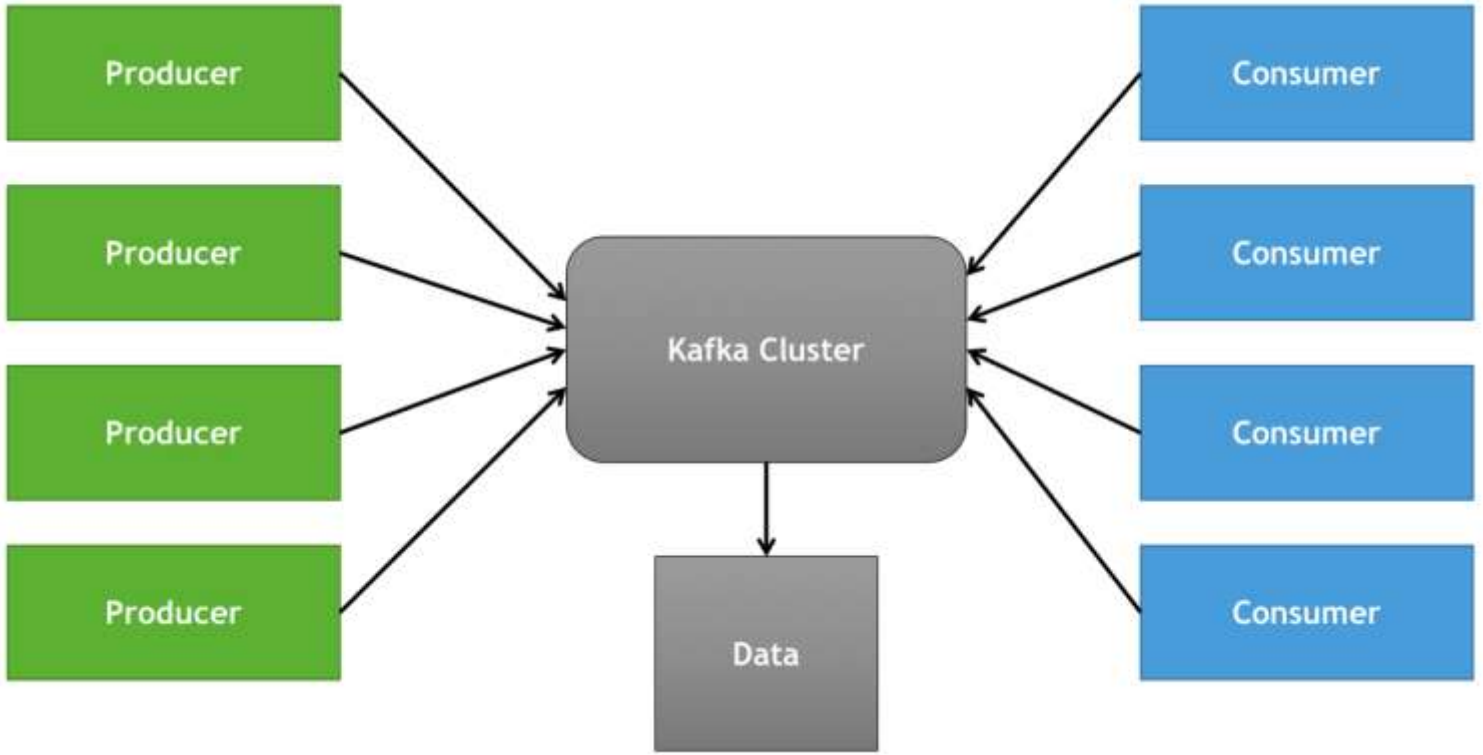
- Kafka record is nothing but a key/value pair to be sent to Kafka
- This consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value.
- If a valid partition number is specified that partition will be used when sending the record.
- If no partition is specified but a key is present a partition will be chosen using a hash of the key
- If neither key nor partition is present a partition will be assigned in a round-robin fashion.
- The record also has an associated timestamp. If the user did not provide a timestamp, the producer will stamp the record with its current time.
- The timestamp eventually used by Kafka depends on the timestamp type configured for the topic.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ❑ **Consumers and Producers**
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

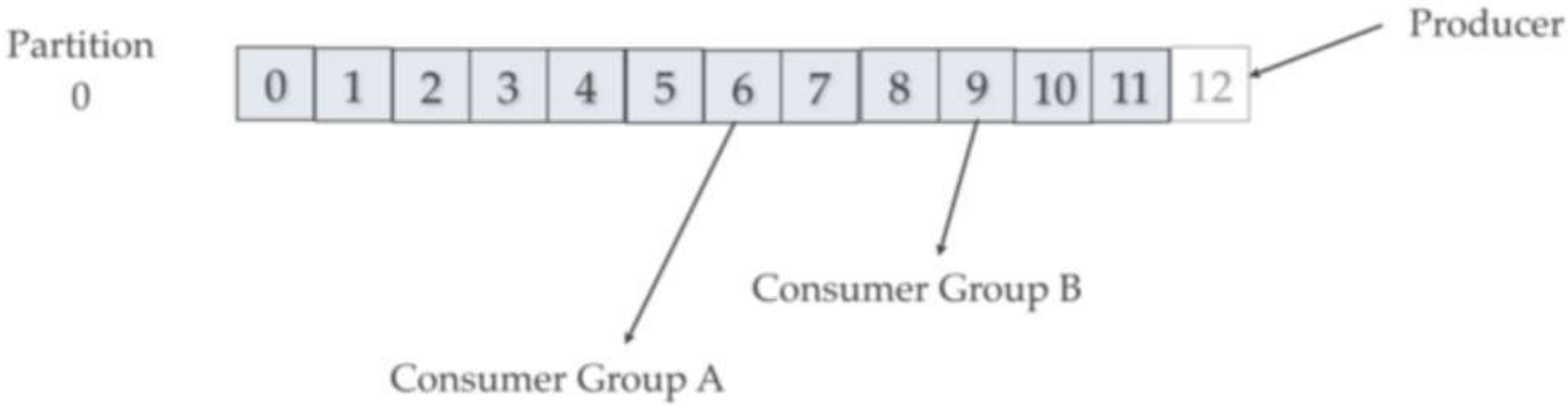
Apache Kafka Introduction

Kafka Producer and Consumer



Apache Kafka Introduction

Kafka Producer



Apache Kafka Introduction

Kafka Producer

- Kafka producers send records to topics.
- The producer picks which partition to send a record to per topic.
- The producer can send records round-robin
- The producer could implement priority systems based on sending records to certain partitions based on the priority of the record.
- Producers send records to a partition based on the record's key.
- The default partitioner for Java uses a hash of the record's key to choose the partition or uses a round-robin strategy if the record has no key.
- Producers write at their cadence so the order of Records cannot be guaranteed across partitions

Kafka Consumer

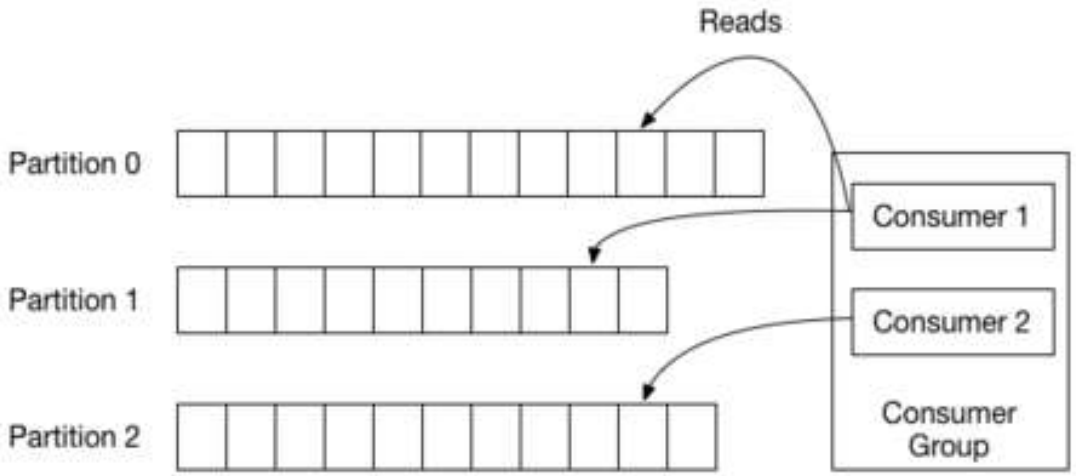


Figure 1: Consumer Group

Kafka Consumer

- Kafka Consumers are typicaclly part of kafka consumer groups
- Consumer group is a multi-threaded or multi-machine consumption from Kafka topics
- The maximum parallelism of a group is that the number of consumers in the group - no of partitions
- Kafka assigns the partitions of a topic to the consumer in a group, so that each partition is consumed by exactly one consumer in the group.
- Kafka guarantees that a message is only ever read by a single consumer in the group.
- Consumers can see the message in the order they were stored in the log.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ❑ **Kafka Logs**
- ❑ Kafka Partitions for Write Throughput
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Kafka Introduction

Kafka Logs

- A log for a topic named "my_topic" with two partitions consists of two directories (namely my_topic_0 and my_topic_1) populated with data files containing the messages for that topic.
- The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer *N* storing the message length which is followed by the *N* message bytes.
- Each message is uniquely identified by a 64-bit integer *offset* giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition.
- A GUID will be generated by the producer for the uniqueness and the consumers will read the logs based upon the GUID
- Each message has its value, offset, timestamp, key, message size, compression codec, checksum, and version of the message format.

Apache Kafka Introduction

Exericese

- ❑ Create a Kafka Multi broker cluster
  - You will be given with 3 machines. Name those as broker-1, broker-2 and broker-3
  - Install the required pre-requisites for installing kafka.
  - Download and Install Apache Zookeeper
  - Download and Install Apache Kafka

- ❑ Transfer data between producer and consumer
  - Create a sample topic
  - Check the topic partitions
  - Pass the message between Producer and Consumer

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ❑ **Kafka Partitions for Write Throughput**
- ❑ Partitions for Consumer Parallelism
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Kafka Partitions for Write Throughput

- Kafka always write data to files immediately.
- Recommend using multiple drives to get good throughput. Do not share the same drives with any other application or for kafka application logs.
- Multiple drives can be configured using log.dirs in server.properties. Kafka assigns partitions in round-robin fashion to log.dirs directories.
- If the data is not well balanced among partitions this can lead to load imbalance among the disks. Also kafka currently doesn't good job of distributing data to less occupied disk in terms of space. So users can easily run out of disk space on 1 disk and other drives have free disk space and which itself can bring the Kafka down.

Kafka Partitions for Write Throughput

- RAID can potentially do better load balancing among the disks. But RAID can cause performance bottleneck due to slower writes and reduces available disk space.
- RAID can tolerate disk failures but rebuilding RAID array is I/O intensive that effectively disables the server. So RAID doesn't provide much real availability improvement.
- Recommended settings –

File Descriptors limits: Kafka needs open file descriptors for files and network connections . We recommend at least 128000 allowed for file descriptors.

Max socket buffer size , can be increased to enable high-performance data transfer.

Kafka Partitions for Write Throughput

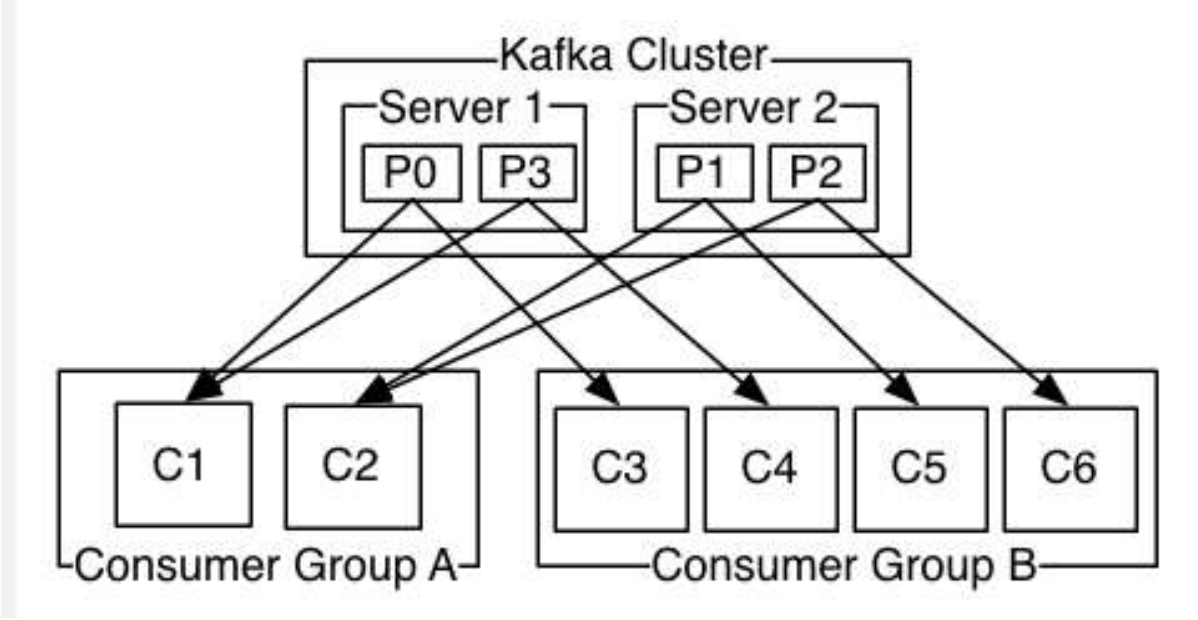
- Kafka uses regular files on disk, and such it has no hard dependency on a specific file system
- Recommend EXT4 or XFS. Recent improvements to the XFS file system have shown it to have the better performance characteristics for Kafka's workload without any compromise in stability.
- Do not use mounted shared drives and any network file systems

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ❑ **Partitions for Consumer Parallelism**
- ❑ Replicas, Followers and Leaders
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Apache Kafka Introduction

- **Partitions for Consumer Parallelism**



Apache Kafka Introduction

❑ Partitions for Consumer Parallelism

- ❑ Each Kafka consumer belongs to a consumer group i.e. it can be thought of as a logical container/namespace for a bunch of consumers
- ❑ A consumer group can choose to receive messages from one or more topics
- ❑ Instances in a consumer group can receive messages from zero, one or more partitions within each topic (depending on the number of partitions and consumer instances)
- ❑ Kafka makes sure that there is no overlap as far as message consumption is concerned i.e. a consumer (in a group) receives messages from exactly one partition of a specific topic
- ❑ The partition to consumer instance assignment is internally taken care of by Kafka and this process is dynamic in nature

Partitions for Consumer Parallelism

- Scaling In and scale out of Consumer groups are possible
- Scalability is the ability to consume messages which are both high volume and velocity.
- Kafka transparently load balances traffic from all partitions amongst a bunch of consumers in a group which means that a consuming application can respond to higher performance and throughput
- If Consumer to Partition ratio is equal to 1 in which each consumer will receive messages from exactly one partition i.e. one-to-one co-relation
- If Consumer to Partition ratio is less than 1 some consumers might receive from more than 1 partition
- If consumer to partition ratio is more than 1 some consumers will remain idle

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism

- ❑ **Replicas, Followers and Leaders**
- ❑ Disaster Recovery – High Level
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Kafka Replicatiton

- The broker is responsible for below things in Kafka
  - Maintaining high availability and consistency in the cluster.
  - Handling requests from producers and consumers.
  - Storing the messages in Kafka
- Same like HDFS, For Maintaining high availability, Kafka is also totally depended upon the replication.
- We know that the Kafka cluster stores streams of records in categories called topics and topics stores the data in partitions.
- Partitions are also the way that Kafka provides redundancy and scalability.
- If we want to achieve high availability in Kafka we need the redundant copy of each partition across the clusters.

Kafka Replicatiton

- Kafka will always replicate partitions for maintaining high availability of a topic. each copy of the partition is called as replicas.
- There are two types of Replica
  - Leader Replica and
  - Follower Replicac
- The leader always handles all read and write requests for any particular partition. \
- Leader handles the write request from producer and writes messages to the Kafka cluster.
- After every 500 milliseconds, followers connects to the leader and fetches the new message and keeps himself ready for the event of leader failers.

Leader Replica

- If replication factor is set to 3, each partition will have 3 replicas which will be stored across the nodes of a cluster and not on the same node.
- Out of this 3 replicas, consumer and producer will always use a specific replica for both producing and consuming messages. i.e all read and write request for a particular partition will always be served by a specific replica and those replicas are called as Leader replicas.
- Each partition has a single replica designated as the leader.
- Leader replica is responsible for reading and writing a data in particular partition.
- Without leader replica, you can not read or write a data to or from the partition.

Follower Replica

- All replicas which are not leader replicas are called as a follower replicas.
- The only job follower replica does is keeping himself up to date with leader replica by fetching a message from leader replica so that in the event of leader failures any follower replicas can act as a leader replica.
- A leader is elected while starting up of the kafka broker by IDs assigned to it in incremental style
- Once the leader is not available, the ID with next increment will be the leader.

Leader and Follower

- Apart from serving read and write request of producer and consumers, Leader is also responsible for keeping a track of replicas which are up to date (In sync) with the leader.
- In order to keep himself in sync with leader, followers always connect to the leader after every 500 milliseconds (replica.fetch.wait.max.ms) and fire a fetch request to the leader.
- In every fetch request, follower sends the offset number that he wants to fetch from the leader.
- The rule is follower can't request any random offset from the leader, i.e. in order to request offset 4 follower must have offsets till 3. This is how leader comes to know which follower is at what offset.
- The followers which are not having the previous offset within 10 seconds of time are termed as out of sync follower
- Replicas who are continuously in sync with leader are called as In sync replicas (ISR).

Controller

- Till now we have understood that leader is the main component using which read and writes to Kafka cluster happens.
- In order to the continuous functionality of Kafka cluster, we need redundancy in the leader of each partition.
- To achieve this redundancy, the broker needs to perform one more role i.e the role of controller.
- The controller is one of the broker node which is responsible for the leader election of each partition.
- The very first broker which we start in Kafka cluster will always be a controller of Kafka cluster.
- Same like how broker creates ephemeral znode in zookeeper while starting himself in /brokers/ids it also tries to create ephemeral znode like /controller and whoever is able to create ephemeral znode acts as a controller.

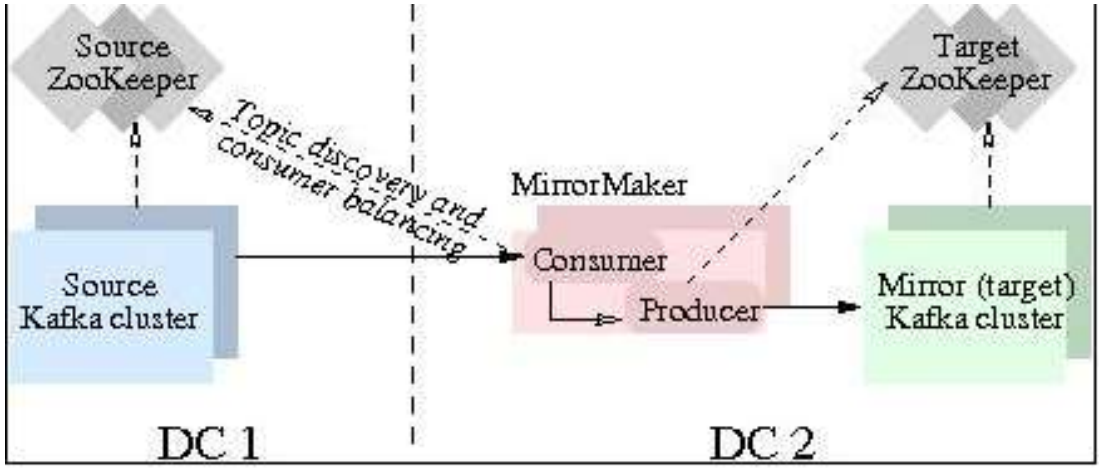
Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism
- ☑ Replicas, Followers and Leaders
- ❑ **Disaster Recovery – High Level**
- ❑ High Water Mark
- ❑ Consumer Load Balancing
- ❑ Fail-over

Disaster Recovery – High Level

- Datacenter downtime and data loss can result in businesses losing a vast amount of revenue or entirely halting operations.
- To minimize the downtime and data loss resulting from a disaster, enterprises create business continuity plans and disaster recovery strategies.
- If disaster strikes—catastrophic hardware failure, software failure, power outage, denial of service attack, or any other event that causes one datacenter to completely fail—Kafka continues running in another datacenter until service is restored.
- Kafka's mirroring feature makes it possible to maintain a replica of an existing Kafka cluster.

Disaster Recovery – High Level

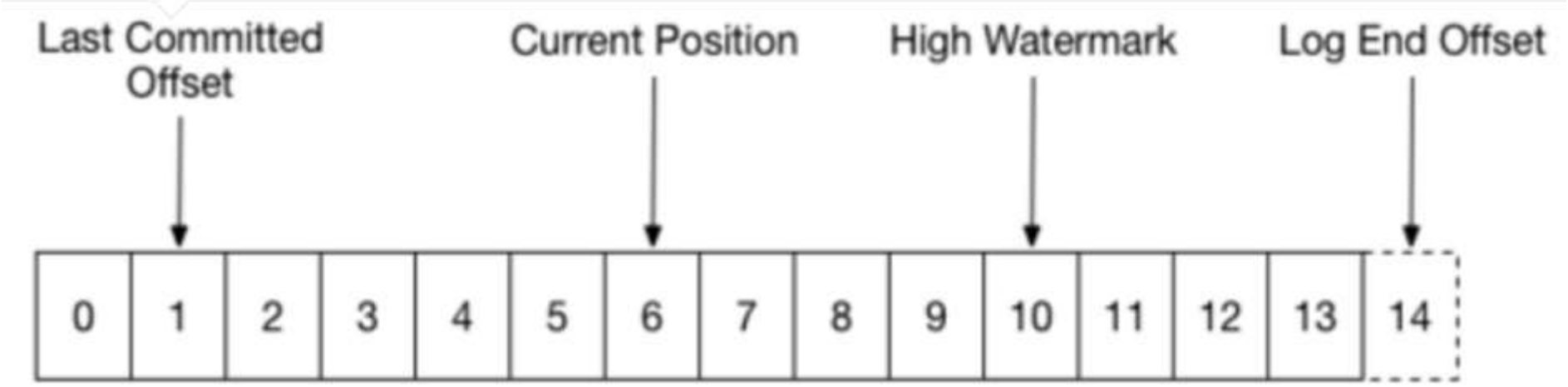


Kafka Mirror Maker will be discussed in upcoming chapter

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism
- ☑ Replicas, Followers and Leaders
- ☑ Disaster Recovery – High Level
- ❑ **High Water Mark**
- ❑ Consumer Load Balancing
- ❑ Fail-over
- ❑ Working on Partitions for parallel processing and resiliency

High Water mark



High Water mark

- The high watermark indicated the offset of messages that are fully replicated, while the end-of-log offset might be larger if there are newly appended records to the leader partition which are not replicated yet.
- Consumers can only consume messages up to the high watermark.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism
- ☑ Replicas, Followers and Leaders
- ☑ Disaster Recovery – High Level
- ☑ High Water Mark
- ❑ **Consumer Load Balancing**
- ❑ Fail-over

Consumer Rebalancing

Recap

- Kafka consumers are part of consumer groups.
- A group has one or more consumers in it. Each partition gets assigned to one consumer.
- Partitions are how Kafka scales out
- If you have more consumers than partitions, then some of your consumers will be idle.
- If you have more partitions than consumers, more than one partition may get assigned to a single consumer.

Consumer Rebalancing

- When a new consumer joins, a rebalance occurs, and the new consumer is assigned some partitions previously assigned to other consumers.
- If there were 10 partitions all being consumed by one consumer, and another consumer joins, there'll be a rebalance, and afterwards, there'll be (typically) five partitions per consumer.
- It's worth noting that during a rebalance, the consumer group "pauses". A similar thing happens when consumers gracefully leave, or the leader detects that a consumer has left.

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism
- ☑ Replicas, Followers and Leaders
- ☑ Disaster Recovery – High Level
- ☑ High Water Mark
- ☑ Consumer Load Balancing
- ❑ **Fail-over**
- ❑ Working on Partitions for parallel processing and resiliency

Kafka failover will be demostrated in the exercise. Learn by practice

Apache Kafka Introduction

- ☑ Why Apache Kafka
- ☑ Apache Kafka Architecture
- ☑ Overview of Key Concepts
- ☑ Apache Zookeeper
- ☑ Cluster, Nodes and Kafka Brokers
- ☑ Kafka Topic and Kafka APIs
- ☑ Kafka partitions, Records and Keys
- ☑ Consumers and Producers
- ☑ Kafka Logs
- ☑ Kafka Partitions for Write Throughput
- ☑ Partitions for Consumer Parallelism
- ☑ Replicas, Followers and Leaders
- ☑ Disaster Recovery – High Level
- ☑ High Water Mark
- ☑ Consumer Load Balancing
- ☑ Fail-over

Course Outline

- ✅ Course Introduction
- ✅ Introduction to Data Ingestion
- ✅ Apache Kafka Introduction
- ❑ **Low-Level Architecture**
- ❑ Advanced Kafka Producers and Consumers
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Low Level Architecture

- ❏ **Kafka Design Motivatiton**
- ❏ Kafka persistance
- ❏ Kafka Producer load Balancing
- ❏ Kafka Producer Record Batching
- ❏ Kafka Compression
- ❏ Pull vs Push
- ❏ Kafka Consumer message state Tracking
- ❏ Message Delivery Semantics
- ❏ Kafka Producer Durability and Acknowledgement
- ❏ Producer Durability
- ❏ Kafka Producer Atomic Log writes
- ❏ Kafka Broker Failover
- ❏ Replicated Log partitions
- ❏ Kafka and Quorum
- ❏ Quotas

Low Level Architecture

Kafka Design Motivation

- Kafka was designed to feed analytics system that did real-time processing of streams.
- The goal behind Kafka, build a high-throughput streaming data platform that supports high-volume event streams like log aggregation, user activity, etc.
- Kafka was also designed to handle periodic large data loads from offline systems as well as traditional messaging use-cases, low-latency.
- MOM is message oriented middleware think IBM MQSeries, JMS, ActiveMQ, and RabbitMQ.
- Like many MOMs, Kafka is fault-tolerance for node failures through replication and leadership election.
- However, the design of Kafka is more like a distributed database transaction log than a traditional messaging system. Unlike many MOMs, Kafka replication was built into the low-level design and is not an afterthought.

Low Level Architecture

Kafka Persistance

- Kafka relies on the file system for storing and caching records.
- The disk performance of hard drives performance of sequential writes is fast.
- JBOD is just a bunch of disk drives. JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec.
- Like Cassandra tables, Kafka logs are write only structures, meaning, data gets appended to the end of the log.
- When using HDD, sequential reads and writes are fast, predictable, and heavily optimized by operating systems. Using HDD, sequential disk access can be faster than random memory access and SSD.

Low Level Architecture

Kafka Persistance

- While JVM GC overhead can be high, Kafka leans on the OS a lot for caching, which is big, fast and rock solid cache.
- OS file caches are almost free and don’t have the overhead of the OS. Implementing cache coherency is challenging to get right, but Kafka relies on the rock solid OS for cache coherence.
- Using the OS for cache also reduces the number of buffer copies. Since Kafka disk usage tends to do sequential reads, the OS read-ahead cache is impressive.

Low Level Architecture

Kafka Producer load Balancing

- The producer asks the Kafka broker for metadata about which Kafka broker has which topic partitions leaders thus no routing layer needed.
- This leadership data allows the producer to send records directly to Kafka broker partition leader.
- The Producer client controls which partition it publishes messages to, and can pick a partition based on some application logic.
- Producers can partition records by key, round-robin or use a custom application-specific partitioner logic.

Low Level Architecture

Kafka Producer Record Batching

- Kafka producers support record batching. Batching can be configured by the size of records in bytes in batch. Batches can be auto-flushed based on time.
- Batching is good for network IO throughput.
- Batching speeds up throughput drastically.
- Buffering is configurable and lets you make a tradeoff between additional latency for better throughput.
- Batching allows accumulation of more bytes to send, which equate to few larger I/O operations on Kafka Brokers and increase compression efficiency.

Low Level Architecture

Kafka Compression

- In large streaming platforms, the bottleneck is not always CPU or disk but often network bandwidth.
- Batching is beneficial for efficient compression and network IO throughput.
- Kafka provides end-to-end batch compression instead of compressing a record at a time, Kafka efficiently compresses a whole batch of records.
- The same message batch can be compressed and sent to Kafka broker/server in one go and written in compressed form into the log partition.
- We can configure the compression so that no decompression happens until the Kafka broker delivers the compressed records to the consumer.
- Kafka supports GZIP, Snappy and LZ4 compression protocols

Low Level Architecture

Pull vs Push

- Messaging is usually a pull-based system
- With the pull-based system, if a consumer falls behind, it catches up later when it can.
- Since Kafka is pull-based, it implements aggressive batching of data.
- A long poll keeps a connection open after a request for a period and waits for a response.
- Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka)
- Push-based or streaming systems have problems dealing with slow or dead consumers.
- It is possible for a push system consumer to get overwhelmed when its rate of consumption falls below the rate of production.

Low Level Architecture

Pull vs Push

- Push-based or streaming systems can send a request immediately or accumulate requests and send in batches.
- The consumer can accumulate messages while it is processing data already sent which is an advantage to reduce the latency of message processing.

Low Level Architecture

Kafka Consumer message state Tracking

- Message tracking is not an easy task. As consumer consumes messages, the broker keeps track of the state.
- Remember that Kafka topics get divided into ordered partitions. Each message has an offset in this ordered partition. Each topic partition is consumed by exactly one consumer per consumer group at a time.
- This partition layout means, the Broker tracks the offset data not tracked per message like MOM, but only needs the offset of each consumer group, partition offset pair stored. This offset tracking equates to a lot fewer data to track.
- The consumer sends location data periodically (consumer group, partition offset pair) to the Kafka broker, and the broker stores this offset data into an offset topic.
- The offset style message acknowledgment is much cheaper compared to MOM.

Low Level Architecture

Message Delivery Semantics

- There are three message delivery semantics: at most once, at least once and exactly once.
- “At most once” is messages may be lost but are never redelivered.
- “At least once” is messages are never lost but may be redelivered.
- “Exactly once” is each message is delivered once and only once. Exactly once is preferred but more expensive, and requires more bookkeeping for the producer and consumer.
- Recall that all replicas have exactly the same log partitions with the same offsets and the consumer groups maintain its position in the log per topic partition.

Low Level Architecture

Message Delivery Semantics - at most once

- To implement “at-most-once” consumer reads a message, then saves its offset in the partition by sending it to the broker, and finally process the message.
- The issue with “at-most-once” is a consumer could die after saving its position but before processing the message.
- Then the consumer that takes over or gets restarted would leave off at the last position and message in question is never processed.

Message Delivery Semantics – at least once

- To implement “at-least-once” the consumer reads a message, process messages, and finally saves offset to the broker.
- The issue with “at-least-once” is a consumer could crash after processing a message but before saving last offset position.
- If the consumer is restarted or another consumer takes over, the consumer could receive the message that was already processed.
- The “at-least-once” is the most common set up for messaging, and it is your responsibility to make the messages idempotent, which means getting the same message twice will not cause a problem.

Message Delivery Semantics – Exactly once

- To implement “exactly once” on the consumer side, the consumer would need a two-phase commit between storage for the consumer position, and storage of the consumer’s message process output. Or, the consumer could store the message process output in the same location as the last offset.

- Kafka offers the first two, and it up to you to implement the third from the consumer perspective.

Low Level Architecture

Kafka Producer Durability and Acknowledgement

- Kafka's offers operational predictability semantics for durability.
- When publishing a message, a message gets "committed" to the log which means all ISRs accepted the message.
- This commit strategy works out well for durability as long as at least one replica lives.
- The producer connection could go down in middle of send, and producer may not be sure if a message it sent went through, and then the producer resends the message.
- This resend-logic is why it is important to use message keys and use idempotent messages
- The producer can resend a message until it receives confirmation, i.e., acknowledgment received.
- The producer resending the message without knowing if the other message it sent made it or not, negates "exactly once" and "at-most-once" message delivery semantics.

Low Level Architecture

Producer Durability

- The producer can specify durability level.
- The producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message.
- The producer can send with no acknowledgments (0)
- The producer can send with just get one acknowledgment from the partition leader (1).
- The producer can send and wait on acknowledgments from all replicas (-1), which is the default.

Low Level Architecture

Kafka Producer Atomic Log writes

- Kafka producers having atomic write across partitions.
- The atomic writes mean Kafka consumers can only see committed logs which can be configurable.
- Kafka has a coordinator that writes a marker to the topic log to signify what has been successfully transacted.
- The transaction coordinator and transaction log maintain the state of the atomic writes.

Low Level Architecture

Kafka Broker Failover

- Kafka keeps track of which Kafka brokers are alive.
- To be alive, a Kafka Broker must maintain a ZooKeeper session using Zookeeper's heartbeat mechanism, and must have all of its followers in-sync with the leaders and not fall too far behind.
- Both the ZooKeeper session and being in-sync is needed for broker live ness which is referred to as being in-sync.
- In-sync replica is called an ISR. Each leader keeps track of a set of “in sync replicas”.
- If ISR/follower dies, falls behind, then the leader will remove the follower from the set of ISRs. Falling behind is when a replica is not in-sync after replica.lag.time.max.ms period.
- A message is considered “committed” when all ISRs have applied the message to their log. Consumers only see committed messages.
- Kafka guarantee: committed message will not be lost, as long as there is at least one ISR.

Low Level Architecture

Replicated Log partitions

- A Kafka partition is a replicated log
- A replicated log is a distributed data system primitive.
- A replicated log is useful for implementing other distributed systems using state machines.
- While a leader stays alive, all followers just need to copy values and ordering from their leader.
- If the leader does die, Kafka chooses a new leader from its followers which are in-sync.
- If a producer is told a message is committed, and then the leader fails, then the newly elected leader must have that committed message.
- The more ISRs you have; the more there are to elect during a leadership failure.

Low Level Architecture

Kafka and Quorum

- Quorum is the number of acknowledgments required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap for availability. Most systems use a majority vote, Kafka does not use a simple majority vote to improve availability.
- In Kafka, leaders are selected based on having a complete log. If we have a replication factor of 3, then at least two ISRs must be in-sync before the leader declares a sent message committed.
- If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages.
- Among the followers there must be at least one replica that contains all committed messages. Problem with majority vote Quorum is it does not take many failures to have inoperable cluster.

Low Level Architecture

Kafka and Quorum

- Kafka maintains a set of ISRs per leader.
- Only members in this set of ISRs are eligible for leadership election.
- Kafka's guarantee about data loss is only valid if at least one replica is in-sync.
- If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid.
- If all replicas are down for a partition, Kafka, by default, chooses first replica.

Low Level Architecture

Quotas

- Kafka has quotas for consumers and producers to limits bandwidth they are allowed to consume.
- These quotas prevent consumers or producers from hogging up all the Kafka broker resources.
- The quota is by client id or user.
- The quota data is stored in ZooKeeper, so changes do not necessitate restarting Kafka brokers.

Low Level Architecture

Exercise

- Use kafka consumer and producer API for capturing data
  - Write Java based Producer API
  - Create a topic with replications and partitions
  - Write Java Based Consumer API
  - Pass Message between producer and consumer

Course Outline

- ✅ Course Introduction
- ✅ Introduction to Data Ingestion
- ✅ Apache Kafka Introduction
- ✅ Low-Level Architecture
- ❑ **Advanced Kafka Producers and Consumers**
- ❑ Schema management in kafka
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Advanced Kafka Producers and Consumers

Advanced Producer

- ❏ **Batching by Time and Size**
- ❏ Compression
- ❏ Async Producers and Sync Producers
- ❏ Default partitioning
- ❏ Custom Partitioning

Advanced Consumer

- ❏ Consumer Poll
- ❏ At most once message semantics
- ❏ At least once message semantics
- ❏ Exactly once message semantics
- ❏ Using ConsumerRebalanceListener

Advanced Kafka Producers and Consumers

Buffering and Batching

- The Producer has buffers of unsent records per topic partition (sized at **batch.size**)
- The Kafka Producer buffers are available to send immediately.
- To reduce requests count and increase throughput, set linger.ms > 0.
- This setting forces the Producer to wait up to **linger.ms** before sending contents of buffer or until batch fills up whichever comes first.
- Under heavy load **linger.ms** not met as the buffer fills up before the linger.ms duration completes.
- Under lighter load, the producer can use to linger to increase broker IO throughput and increase compression.

Advanced Kafka Producers and Consumers

Buffering and Batching

- The buffer.memory controls total memory available to a producer for buffering.
- If records get sent faster than they can be transmitted to Kafka then and this buffer will get exceeded then additional send calls block up to max.block.ms after then Producer throws a TimeoutException.
- You can also set the producer config property buffer.memory which default 32 MB of memory.
- This denotes the total memory (in bytes) that the producer can use to buffer records to be sent to the broker.
- If the Producer is sending records faster than the broker can receive records, an exception is thrown.

Batching by Time

- The producer config property linger.ms defaults to 0.
- You can set this so that the Producer will wait this long before sending if batch size not exceeded.
- This setting allows the Producer to group together any records that arrive before they can be sent into a batch.
- Setting this value to 5 ms or greater is good if records arrive faster than they can be sent out.
- The producer can reduce requests count even under moderate load using linger.ms.
- Kafka may send batches before this limit is reached

Batching by Size

- The linger.ms setting adds a delay to wait for more records to build up, so larger batches get sent.
- Increase linger.ms to increase brokers throughput at the cost of producer latency.
- If the producer gets records whose size is batch.size or more for a broker's leader partitions, then it is sent right away.
- If Producers gets less than batch.size but linger.ms interval has passed, then records for that partition are sent.
- Increase linger.ms to improve the throughput of Brokers and reduce broker load
- Normally the producer will not wait at all, and simply send all the messages that accumulated while the previous send was in progress

Code Snippets - Batching by time and Size

```
/Linger up to 100 ms before sending batch if size not met
props.put(ProducerConfig.LINGER_MS_CONFIG, 100);

//Batch up to 64K buffer sizes.
props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ❑ **Compression**
- ❑ Async Producers and Sync Producers
- ❑ Default partitioning
- ❑ Custom Partitioning

Advanced Consumer

- ❑ Consumer Poll
- ❑ At most once message semantics
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Advanced Kafka Producers and Consumers

Compression

- The producer config property **compression.type** defaults to none
- Setting this allows the producer to compresses request data.
- By default, the producer does not compress request data.
- This setting can be set to none, gzip, snappy, lz4 or custom.
- The compression is by batch and improves with larger batch sizes.
- End to end compression is possible if the Kafka Broker config “compression.type” set to “producer”.

Advanced Kafka Producers and Consumers

Compression

- The compressed data can be sent from a producer, then written to the topic log and forwarded to a consumer by broker using the same compressed format.
- End to end compression is efficient as compression only happens once and is reused by the broker and consumer. End to end compression takes the load off of the broker.

Code Snippet – Compression

```
//Use Snappy compression for batch compression.
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ✓ Batching by Time and Size
- ✓ Compression
- ❑ **Async Producers and Sync Producers**
- ❑ Default partitioning
- ❑ Custom Partitioning

Advanced Consumer

- ❑ Consumer Poll
- ❑ At most once message semantics
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Advanced Kafka Producers and Consumers

Synchronous Producers

- Kafka provides a synchronous send method to send a record to a topic.
- RecordMetadata has "partition" where the record was written and the ‘offset’ of the record in that partition.
- It wait for acknowledgement
- Message is sent only after the acknowledgement is received.
- In case of exception, it stop sending the messages after the exception occurs.

Advanced Kafka Producers and Consumers

Synchronous Producers

Code Snippet

```
public class KafkaProducerExample {

...
static void runProducer(final int sendMessageCount) throws Exception {
        final Producer<Long, String> producer = createProducer();
        long time = System.currentTimeMillis();
…
RecordMetadata metadata = producer.send(record).get();

long elapsedTime = System.currentTimeMillis() - time;
System.out.printf("sent record(key=%s value=%s) " +
                  "meta(partition=%d, offset=%d) time=%d\n",
        record.key(), record.value(), metadata.partition(), \
        metadata.offset(), elapsedTime);
```

Advanced Kafka Producers and Consumers

Asynchronous Producers

- Kafka provides asynchronous send method to send a record to a topic.
- Since the send call is asynchronous it returns a Future for the RecordMetadata that will be assigned to this record.
- It wont wait for the acknowledgement
- Some messages will be sent before that something is wrong and perform some actions
- in asynchronous approach the number of messages which are "in flight" is controlled by max.in.flight.requests.per.connection parameter.
- The callback ensures that the message request is completed or not.

Asynchronous Producers

Code Snippet

```
public void send() {
                aProducer.send(message, new Callback() {
                                public void onCompletion(RecordMetadata metadata, Exception exception) {
                                                if (exception != null) {
                                                                // How do I find get the original message so that I can do something with it if
needed?
                                                                throw new KafkaException("Asynchronous send failure: ", exception);
                                                } else {
                                                                //NoOp
                                                }
                }
}
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ❑ **Default partitioning**
- ❑ Custom Partitioning

Advanced Consumer

- ❑ Consumer Poll
- ❑ At most once message semantics
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Default Partition – Round Robin

- Let's consider, we have a TopicA in Kafka which has partitions count 5 and replication factor 3 and we want to distribute data uniformly between all the partitions so that all the partitions contains same data size.

- The Kafka uses the default partition mechanism to distribute data between partitions, but in case of default partition mechanism it might be possible that our some partitions size larger than others.

- Suppose we have inserted 40 GB data into Kafka, then the data size of each partition may look like:

Without Partitioner	With Partitioner
Partition0 - 10 GB	Partition0 – 8GB
Partition1 - 8 GB	Partition1 – 8 GB
Partition2 - 6 GB	Partition2 – 8 GB
Partition3 - 9 GB	Partition3 – 8 GB
Partition4 - 11 GB	Partition4 – 8 GB

- Ensuring the fair share of the data to the partitions

Default Partition – Round Robin

- Round Robin is the default partition strategy for producer.

Code Snippet

```
class RRPartitioner():
        def __init__():
                # Using topic metadata get total number of partitions
                self.total_partitions = client[topic].get_number_partitions()
                self.part_offset = 0

        def partitioner(self, key, msg):
                if self.part_offset > self.total_partitions:
                        self.part_offset = 0
                        return self.part_offset
                else:
                        self.part_offset += 1
                        return self.part_offset
```

Advanced Kafka Producers and Consumers

Advanced Producer

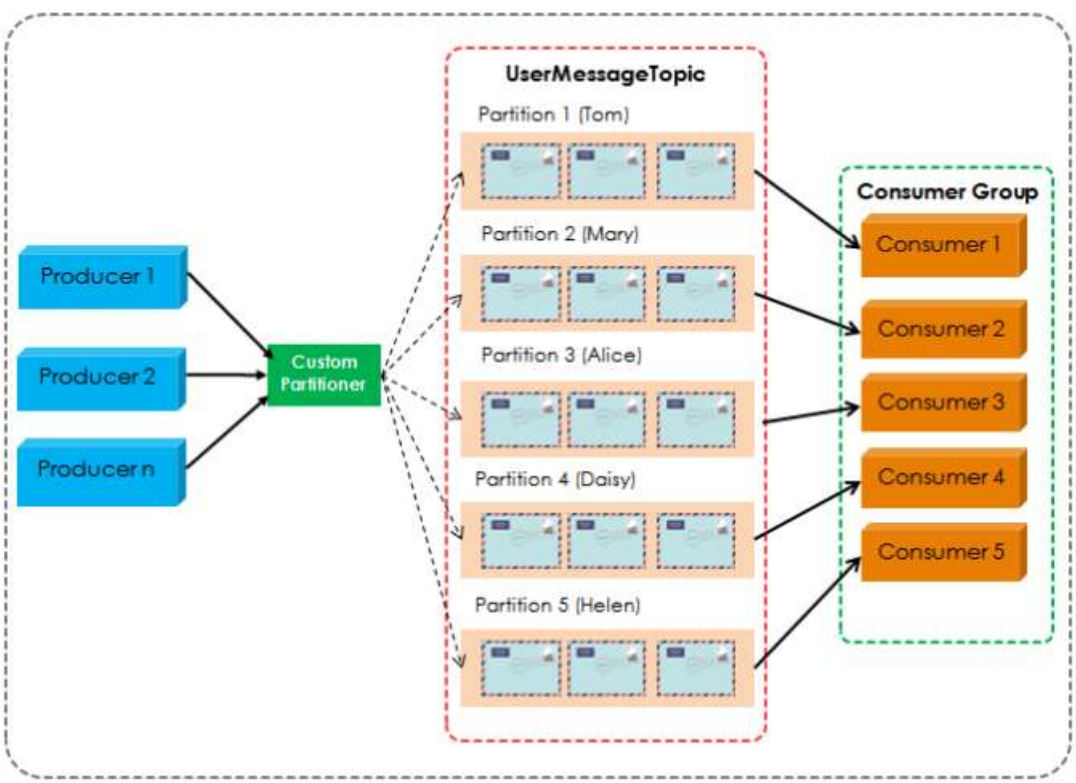
- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ❑ **Custom Partitioning**

Advanced Consumer

- ❑ Consumer Poll
- ❑ At most once message semantics
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Advancced Kafka Producer and Consumer

Custom Partitioning



Advancced Kafka Producer and Consumer

Custom Partitioning

- By default, Apache Kafka producer will distribute the messages to different partitions by round-robin fashion.
- Writing a custom partition will split the data to the partition based upon the partition logic

Code snippet

```
public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes,
                     Cluster cluster) {

                    int partition = 0;
                    String userName = (String) key;
                    // Find the id of current user based on the username
                    Integer userId = userService.findUserId(userName);
                    // If the userId not found, default partition is 0
                    if (userId != null) {
                    partition = userId;
                    }
            return partition;
}
```

Advancced Kafka Producer and Consumer

Advancecd Producer Exercise

- ❑ Exercise on Message Batching and Compression
- ❑ Exercise – Round Robin Partition
- ❑ Exercise Custom Partition

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ❑ **Consumer Poll**
- ❑ At most once message semantics
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Advancced Kafka Producer and Consumer

Consumer Poll

- Poll the kafka broker to obtain new messages
- Once the record is polled, Consumer protocol process the messages and Commit the update consumer position back to the kafka broker.
- If the application running this loop dies, it will start consuming at the last committed consumer position.
- Effectively, this guarantees you that you will process each message *at least once*. It can very well happen that the same message is processed multiple times.
- Consumer Poll has no Timeout

Consumer Poll

- poll() function actually has a parameter called timeout.
- It's important to realize that this timeout only applies to part of what the poll() function does internally.
- The timeout parameter is the number of milliseconds that the network client inside the kafka consumer will wait for sufficient data to arrive from the network to fill the buffer.
- If no data is sent to the consumer, the poll() function will take at least this long. If data is available for the consumer, poll() might be shorter.
- Before it gets to that part of the poll() function, the consumer will also do a check to ensure that the broker is available.
- That part does not respect the timeout. It will try infinitely long to fetch metadata from the cluster

Consumer Poll

- If the processing of messages is expensive (e.g. complex calculations, or long blocking I/O), you may run into a CommitFailedException.
- The reason for this is that the consumer is expected to send a heartbeat to the broker every so often.
- This heartbeat informs the broker that the consumer is still alive. When the heartbeat doesn't arrive in time, the broker will mark the consumer as dead and kick it from the consumer group.
- The time is defined by the session.timeout.ms configuration of the broker (default is 30 seconds).
- Both the poll() and commitSync() functions send this heartbeat. However, if the time between the two function calls is 30 seconds, then by the time commitSync() is called, the broker will already have marked the consumer as dead. As a result you get a CommitFailedException.

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ☑ Consumer Poll
- ❑ **At most once message semantics**
- ❑ At least once message semantics
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

At most once message semantics

- if the producer does not retry when an ack times out or returns an error, then the message might end up not being written to the Kafka topic, and hence not delivered to the consumer.
- In most cases it will be, but in order to avoid the possibility of duplication, we accept that sometimes messages will not get through.

Advancced Kafka Producer and Consumer

At most once message semantics

- To configure this type of consumer:
  -- Set '**enable.auto.commit'** to **true** or
  -- Set **'auto.commit.interval.ms'** to a lower timeframer.
  -- And do not make call to **consumer.commitSync();** from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

Code Snippet

```
// Set this property, if auto commit should happen.
props.put("enable.auto.commit", "true");
// Auto commit interval, kafka would commit offset at this interval.
props.put("auto.commit.interval.ms", "101");
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ☑ Consumer Poll
- ☑ At most once message semantics
- ❑ **At least once message semantics**
- ❑ Exactly once message semantics
- ❑ Using ConsumerRebalanceListener

Advanced Kafka Producer and Consumer

At least once message semantics

- if the producer receives an acknowledgement (ack) from the Kafka broker and acks=all, it means that the message has been written exactly once to the Kafka topic.
- If a producer ack times out or receives an error, it might retry sending the message assuming that the message was not written to the Kafka topic.
- If the broker had failed right before it sent the ack but after the message was successfully written to the Kafka topic, this retry leads to the message being written twice and hence delivered more than once to the end consumer.
- This approach can lead to duplicated work and incorrect results.

Advanced Kafka Producer and Consumer

At least once message semantics

- To configure this type of consumer:
  -- Set '**enable.auto.commit'** to **false or**
  -- Set '**enable.auto.commit'** to **true** with **'auto.commit.interval.ms'** to a higher number.
  -- Consumer should now then take control of the message offset commits to Kafka by making the following call **consumer.commitSync()**;

Code Snippet

```
// Set this property, if auto commit should happen.
props.put("enable.auto.commit", "true");
// Make Auto commit interval to a big number so that auto commit does not happen,
// we are going to control the offset commit via consumer.commitSync(); after processing
// message.
props.put("auto.commit.interval.ms", "999999999999");
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ☑ Consumer Poll
- ☑ At most once message semantics
- ☑ At least once message semantics
- ❑ **Exactly once message semantics**
- ❑ Using ConsumerRebalanceListener

Advanced Kafka Producer and Consumer

Exactly once message semantics

- Even if a producer retries sending a message, it leads to the message being delivered exactly once to the end consumer.

- Exactly-once semantics is the most desirable guarantee, but also a poorly understood one. This is because it requires a cooperation between the messaging system itself and the application producing and consuming the messages.

- To configure this type of consumer
  -- Set **enable.auto.commit = false.**
  -- **Do not** make call to **consumer.commitSync();** after processing message.
  -- Register consumer to a topic by making a '**subscribe**' call. Subscribe call behavior is explained earlier in the article.
  -- Implement a **ConsumerRebalanceListener** and within the listener perform **consumer.seek(topicPartition,offset);** to start reading from a specific offset of that topic/partition.

Advanced Kafka Producer and Consumer

Exactly once message semantics

-- While processing the messages, get hold of the offset of each message. Store the processed message's offset in an atomic way along with the processed message using atomic-transaction. When data is stored in relational database atomicity is easier to implement.

-- Implement idempotent as a safety net.

Code Snippets

```
// Below is a key setting to turn off the auto commit.
props.put("enable.auto.commit", "false");
props.put("heartbeat.interval.ms", "2000");
props.put("session.timeout.ms", "6001");
// Control maximum data on each poll, make sure this value is bigger than the maximum //
single message size
props.put("max.partition.fetch.bytes", "140");
…
// Save processed offset in external storage.
offsetManager.saveOffsetInExternalStore(record.topic(), record.partition(),
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ☑ Consumer Poll
- ☑ At most once message semantics
- ☑ At least once message semantics
- ☑ Exactly once message semantics
- ☐ **Using ConsumerRebalanceListener**

Advanced Kafka Producer and Consumer

ConsumerRebalanceListener

- When the situation arises to adjust the number of partitions, rebalance will be triggered.
- When Kafka is managing the group membership, a partition re-assignment will be triggered any time the members of the group change or the subscription of the members changes.
- This can occur when processes die, new process instances are added or old instances come back to life after failure.
- There are many uses for this functionality. One common use is saving offsets in a custom store. By saving offsets in the onPartitionsRevoked(Collection) call we can ensure that any time partition assignment changes the offset getss saved.
- Another use is flushing out any kind of cache of intermediate results the consumer may be keeping.
- Callback will execute in the user thread as part of the poll(long) call whenever partition assignment changes.

ConsumerRebalanceListener

Code Snippet

```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
private Consumer<?,?> consumer;

public SaveOffsetsOnRebalance(Consumer<?,?> consumer) {
                this.consumer = consumer;
}

public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                // save the offsets in an external store using some custom code not described here
                for(TopicPartition partition: partitions)
saveOffsetInExternalStore(consumer.position(partition));
}

public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                // read the offsets from an external store using some custom code not described here
                for(TopicPartition partition: partitions)
                consumer.seek(partition, readOffsetFromExternalStore(partition));
                }
}
```

Advanced Kafka Producers and Consumers

Advanced Producer

- ☑ Batching by Time and Size
- ☑ Compression
- ☑ Async Producers and Sync Producers
- ☑ Default partitioning
- ☑ Custom Partitioning

Advanced Consumer

- ☑ Consumer Poll
- ☑ At most once message semantics
- ☑ At least once message semantics
- ☑ Exactly once message semantics
- ☑ Using ConsumerRebalanceListener

Course Outline

- ✅ Course Introduction
- ✅ Introduction to Data Ingestion
- ✅ Apache Kafka Introduction
- ✅ Low-Level Architecture
- ✅ Advanced Kafka Producers and Consumers
- ❑ **Schema management in kafka**
- ❑ Kafka Security
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Advanced Kafka Producers and Consumers

Schema Management in kafka

- ❏ **Avro File Formation**
- ❏ Schema Evolution
- ❏ Schema Registry

Schema Management in Kafka

Introduction to Avro File Format

- Avro Data File Format is one part of the Avro project
- Avro is an effective data serialisation framework. Widely supported throughout Hadoop ecosystem.
- Offers compatibility with sacrificing performance
  - Data is serialized according to a schema you define
  - Read/write data In java, C, C++, C#, Python, PHP and other languages
  - Serializes data using a highly-optimized binary coding
  - Specifies rules for evolving your schema over time.
- Avro also supports Remote Procedure Calls(RPC)
  - Can be used for building custom network protocols

Schema Management in Kafka

Introduction to Avro File Format

- Supported Types in Avro Schema

Name	Description	Java Equivalent
`null`	An absence of a value	`null`
`boolean`	A binary value	`boolean`
`int`	32-bit signed integer	`int`
`long`	64-bit signed integer	`long`
`float`	Single-precision floating point value	`float`
`double`	Double-precision floating point value	`double`
`bytes`	Sequence of 8-bit unsigned bytes	`java.nio.ByteBuffer`
`string`	Sequence of Unicode characters	`java.lang.CharSequence`

Introduction to Avro File Format

- Avro also supports complex types

Name	Description
`record`	A user-defined type composed of one or more named fields
`enum`	A specified set of values
`array`	Zero or more values of the same type
`map`	Set of key-value pairs; key is string while value is of specified type
`union`	Exactly one value matching a specified set of types

- The record type is the most important

Introduction to Avro File Format

- Sql Create Schema

```
CREATE TABLE employees
   (id INT, name STRING, title STRING, bonus INT)
```

- Equivalent Avro Schema

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Employee",
 "fields":[
     {"name":"id", "type":"int"},
     {"name":"name", "type":"string"},
     {"name":"title", "type":"string"},
     {"name":"bonus", "type":"int"}]
}
```

Introduction to Avro File Format

- Avro also supports setting a default value in the schema
  - Used when no value was explicitly set for a field
  - Similar to Sql

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Invoice",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"taxcode", "type":"int", "default":"39"},
    {"name":"lang", "type":"string", "default":"EN_US"}]
}
```

The **taxcode** and **lang** fields have default values

Introduction to Avro File Format

- Avro checks for null values when serializing the data
- Null values are only allowed when explicitly specified in the schema

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Employee",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"name", "type":"string"},
    {"name":"title", "type":["null","string"]},
    {"name":"bonus", "type":["null","int"]}]
}
```

The **title** and **bonus** fields allow null values

Schema Management in Kafka

Avro Container Format

- Avro also defines a container file format for storing avro records
  - Also known as " Avro Data File Format"
  - Cross language support for reading and writing data
- Supports compressing blocks(groups) of records
  - It is splittable for efficient processing
- This format is self describing
  - Each file contains a copy of the schema used to write its data.
  - All records in a file must use the same schema

Advanced Kafka Producers and Consumers

Schema Management in kafka

- ☑ Avro File Formation
- ❑ **Schema Evolution**
- ❑ Schema Registry

Schema Management in Kafka

Avro Schema Evolution

- The structure of the data will change over time
  - Fields may added, removed, changed or renamed.
- These changes can break compatibility with many formats.
- Data written to avro data files is always readable.
  - The schema used to write the data is embedded in the file itself.
  - However an application reading data might expect the new structure
- Avro has a unique approach to maintaining the forward compatibility
  - A Reader can use a different schema than the writer.

Avro Schema Evolution

- Imagine we have written million of records with this schema.

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name", "type": "string"},
     {"name": "faxNumber", "type": "string"}
]}
```

Avro Schema Evolution

- We would like to modernize this based on the schema below
  - Rename ID field to customerID and change the type from int to long
  - Remove faxNumber field
  - Add prefLang field
  - Add email field

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name": "customerId", "type": "long"},
     {"name": "name", "type": "string"},
     {"name": "prefLang", "type": "string"},
     {"name": "email", "type": "string"}
]}
```

Avro Schema Evolution

- We could use the new schema to write new data
- Applications that use the new schema could read the new data
- Unfortunately new applications wouldn't be able to read the old data.
- We must take a few schema changes to improve the compatibility

Avro Schema Evolution

- If you rename a field, you must specify an alias for the old name(s)
- Here we map the old ID field to the new customerID field.

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name":"customerId", "type":"long",
          "aliases":["id"]},
     {"name":"name", "type":"string"},
     {"name":"prefLang", "type":"string"},
     {"name":"email", "type":"string"}
]}
```

Schema Management in Kafka

Avro Schema Evolution

- Newly-added fields will lack values for the records previously written. You must specify a default value.

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
    {"name":"customerId", "type":"long",
         "aliases":["id"]},
    {"name":"name", "type":"string"},
    {"name":"prefLang", "type":"string",
         "default":"en_US"},
    {"name":"email",
         "type":["null","string"],"default":null}
]}
```

Default value for **prefLang** is **en_US**

email is nullable so **null** can be the default

Schema Management in Kafka

Avro Schema Evolution

- The following changes will not affect the existing readers.
  - Adding, changing or removing a doc attribute
  - Changing the field's default value
  - Adding a new field with a default value
  - Removing a field that specified a default value
  - Promoting a field to a wider type(eg : Int to Long)
  - Adding alias for a field

Schema Management in Kafka

Avro Schema Evolution

- The following changes some changes that might break compatibility
  - Changing the record's name of namespace
  - Adding a new field without a default value
  - Removing a symbol from an enum
  - Removing a type from a union
  - Modifying a field's type to one that could result in truncation.

- To handle with these compatibilities
  - Read your old data(using the original schema)
  - Modify data as needed in your application
  - Write the new data(using the new schema)

- Existing readers/writers may need to be updated to use new schema

Advanced Kafka Producers and Consumers

Schema Management in kafka

- ☑ Avro File Formation
- ☑ Schema Evolution
- ❑ **Schema Registry**

Schema Management in Kafka

Schema Registry

- Kafka Avro serialization project provides serializers.
- Kafka Producers and Consumers that use Kafka Avro serialization handle schema management and serialization of records using Avro and the Schema Registry.
- When using the Confluent Schema Registry, Producers don't have to send schema just the schema id which is unique.
- The consumer uses the schema id to look up the full schema from the Confluent Schema Registry if not already cached.
- Since you don't have to send the schema with each set of records, this saves time. Not sending the schema with each record or batch of records, speeds up the serialization as only the id of the schema is sent.

Schema Management in Kafka

Schema Registry

- The Kafka Producer creates a record/message, which is an Avro record.
- The record contains a schema id and data.
- With Kafka Avro Serializer, the schema is registered if needed and then it serializes the data and schema id.
- The Kafka Avro Serializer keeps a cache of registered schemas from Schema Registry their schema ids.
- Consumers receive payloads and deserialize them with Kafka Avro Deserializers which use the Confluent Schema Registry.
- Deserializer looks up the full schema from cache or Schema Registry based on id.

Schema Management in Kafka

Schema Registry

- The Kafka Producer creates a record/message, which is an Avro record.
- The record contains a schema id and data.
- With Kafka Avro Serializer, the schema is registered if needed and then it serializes the data and schema id.
- The Kafka Avro Serializer keeps a cache of registered schemas from Schema Registry their schema ids.
- Consumers receive payloads and deserialize them with Kafka Avro Deserializers which use the Confluent Schema Registry.
- Deserializer looks up the full schema from cache or Schema Registry based on id.

Schema Management in Kafka

Schema Registry

- Consumer has its schema which could be different than the producers.
- The consumer schema is the schema the consumer is expecting the record/message to conform to.
- With the Schema Registry a compatibility check is performed and if the two schemas don't match but are compatible, then the payload transformation happens via Avro Schema Evolution.
- Kafka records can have a Key and a Value and both can have a schema.

Schema Management in Kafka

Schema Registry

- The Schema Registry can store schemas for keys and values of Kafka records.
- It can also list schemas by subject.
- It can list all versions of a subject (schema).
- It can retrieve a schema by version or id.
- It can get the latest version of a schema.
- Importantly, the Schema Registry can check to see if schema is compatible with a certain version.
- There is a compatibility level (BACKWARDS, FORWARDS, FULL, NONE) setting for the Schema Registry and an individual subject.
- You can manage schemas via a REST API with the Schema registry.

Schema Management in Kafka

Schema Registry

- Backward compatibility means data written with older schema is readable with a newer schema.
- Forward compatibility means data written with newer schema is readable with old schemas.
- Full compatibility means a new version of a schema is backward and forward compatible.
- None disables schema validation and it not recommended. If you set the level to none then Schema Registry just stores the schema and Schema will not be validated for compatibility at all.

Schema Management in Kafka

Schema Registry

- From Kafka perspective, Schema evolution happens only during deserialization at Consumer (read).
- If Consumer's schema is different from Producer's schema, then value or key is automatically modified during deserialization to conform to consumers reader schema if possible
- Avro schema evolution is an automatic transformation of Avro schema between the consumer schema version and what the schema the producer put into the Kafka log.
- When Consumer schema is not identical to the Producer schema used to serialize the Kafka Record, then a data transformation is performed on the Kafka record's key or value.
- If the schemas match then no need to do a transformation

Schema Management in Kafka

Schema Registry

- The Schema Registry allows you to manage schemas using the following operations:

  - store schemas for keys and values of Kafka records
  - List schemas by subject.
  - list all versions of a subject (schema).
  - Retrieves a schema by version
  - Retrieves a schema by id
  - Retrieve the latest version of a schema
  - Perform compatibility checks
  - Set compatibility level globally
  - Set compatibility level globally

- All of these are available via a REST API with the Schema Registry.

Schema Management in Kafka

Schema Registry

- Confluent Schema Registry are available with Confluent platform Open source and Enterprise package.
- Cloudera and hortonworks integrate this in their package.
- It will not support apache Kafka. It need confluent way of installing the components.

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ❑ **Kafka Security**
- ❑ Kafka Disaster Recovery
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Schema Management in Kafka

Kafka Security

- Apache Kafka is an internal middle layer enabling your back-end systems to share real-time data feeds with each other through Kafka topics.
- With a standard Kafka setup, any user or application can write any messages to any topic, as well as read data from any topics.
- As your company moves towards a shared tenancy model where multiple teams and applications use the same Kafka Cluster, or your Kafka Cluster starts on boarding some critical and confidential information, you need to implement security.

Kafka Security

- Kafka Security has three components:

  - Encryption of data in-flight using SSL / TLS

  - Authentication using SSL or SASL

  - Authorization using ACLs

Schema Management in Kafka

Kafka Security

- ❑ Encryption of data in-flight using SSL/TLS

  - This allows your data to be encrypted between your producers and Kafka and your consumers and Kafka.

  - This is a very common pattern everyone has used when going on the web. That's the "S" of HTTPS

- ❑ Authentication using SSL or SASL

  - This allows your producers and your consumers to authenticate to your Kafka cluster, which verifies their identity.

  - It's also a secure way to enable your clients to endorse an identity.

Schema Management in Kafka

Kafka Security

- Authorization using ACLs

  - Once your clients are authenticated, your Kafka brokers can run them against access control lists (ACL) to determine whether or not a particular client would be authorised to write or read to some topic.

Schema Management in Kafka

Encryption(SSL)

- Encryption solves the problem of the man in the middle (MITM) attack.
- That's because your packets, while being routed to your Kafka cluster, travel your network and hop from machines to machines.
- If your data is PLAINTEXT (by default in Kafka), any of these routers could read the content of the data you're sending
- Now with Encryption enabled and carefully setup SSL certificates, your data is now encrypted and securely transmitted over the network.
- With SSL, only the first and the final machine possess the ability to decrypt the packet being sent.

Schema Management in Kafka

Encryption(SSL)

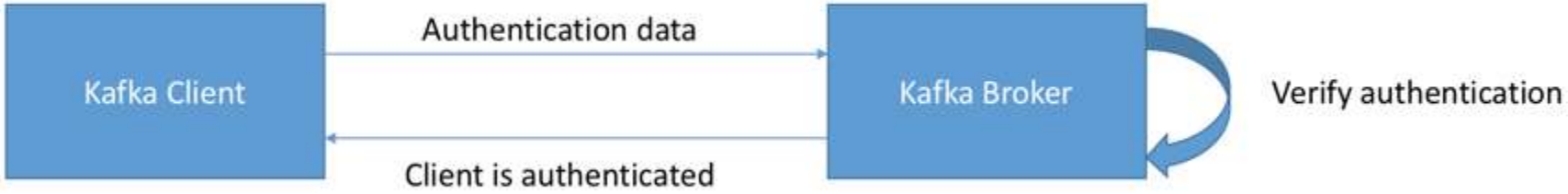
- This encryption comes at a cost
- CPU is now leveraged for both the Kafka Clients and the Kafka Brokers in order to encrypt and decrypt packets.
- Note the encryption is only in-flight and the data still sits un-encrypted on your broker's disk.

Schema Management in Kafka

Authentication (SSL & SASL)

- There are two ways to authenticate your Kafka clients to your brokers:

  -- SSL and
  -- SASL



Schema Management in Kafka

Authentication (SSL & SASL)

- ❑ SSL Authentication

  - SSL Auth is basically leveraging a capability from SSL called two ways authentication.

  - The idea is to also issue certificates to your clients, signed by a certificate authority, which will allow your Kafka brokers to verify the identity of the clients.

- ❑ SASL Authentication

  - SASL stands for Simple Authorization Service Layer and trust me, the name is deceptive, things are not simple.

  - Basically, the idea is that the authentication mechanism is separated from the Kafka protocol

  - It’s very popular with Big Data systems and most likely your Hadoop setup already leverages that

Authentication (SSL & SASL)

- SASL PLAINTEXT

  - This is a classic username/password combination.

  - These usernames and passwords have to be stored on the Kafka brokers in advance and each change needs to trigger a rolling restart.

  - It’s very annoying and not the recommended kind of security.

  - If you use SASL/PLAINTEXT, make sure to enable SSL encryption so that credentials aren’t sent as PLAINTEXT on the network

Authentication (SSL & SASL)

- SASL SCRAM

  - This is a username/password combination alongside a challenge (salt), which makes it more secure.

  - On top of this, username and password hashes are stored in Zookeeper, which allows you to scale security without rebooting brokers.

  - If you use SASL/SCRAM, make sure to enable SSL encryption so that credentials aren't sent as PLAINTEXT on the network

Schema Management in Kafka

Authentication (SSL & SASL)

- SASL GSSAPI(Kerberos)

  - This is based on Kerberos ticket mechanism, a very secure way of providing authentication.

  - LDAP and Microsoft Active Directory are the most common implementation of Kerberos.

  - SASL/GSSAPI is a great choice for big enterprises as it allows the companies to manage security from within their Kerberos Server.

  - Additionally, communications are encrypted to SSL encryption is optional with SASL/GSSAPI.

  - Setting up Kafka with Kerberos is the most difficult option. Cloudera. Hortonworks and Confllluent made the Kerberos installation simple.

Schema Management in Kafka

Authorization

- Once your Kafka clients are authenticated, Kafka needs to be able to decide what they can and cannot do.
- This is where Authorization comes in, controlled by Access Control Lists (ACL).
- ACL are what you expect them to be: User A can(‘t) do Operation B on Resource C from Host D.
- Please note that currently with the packaged SimpleAclAuthorizer coming with Kafka, ACL are not implemented to have Groups rules or Regex-based rules.
- ACL are great because they can help you prevent disasters.

```
kafka-acl.sh --topic first --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:navaneeth
```

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ❑ **Kafka Disaster Recovery**
- ❑ Kafka Cluster Administration and Operations
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Kafka Disaster Recovery

Mirroring and Mirror Maker

- Mirroring is replication between clusters. The name mirroring termed here to not confuse with replication

  - Replication uses cluster involving brokers, partition leaders, partition followers, ISRs and Zookeeper
  - Mirroring is just a consumer/producer pair in two clusters

- MirrorMaker is used for replicating one cluster data to another cluster
- MirrorMaker acts like a consumer to a source cluster
- MirrorMaker acts like a producer to a destination cluster
- Data read from source topics in source cluster and written to same named topics in destination cluster
- Source and destination clusters are independent and not coupled. It can have different topics.

Kafka Disaster Recovery

MirrorMaker Use cases

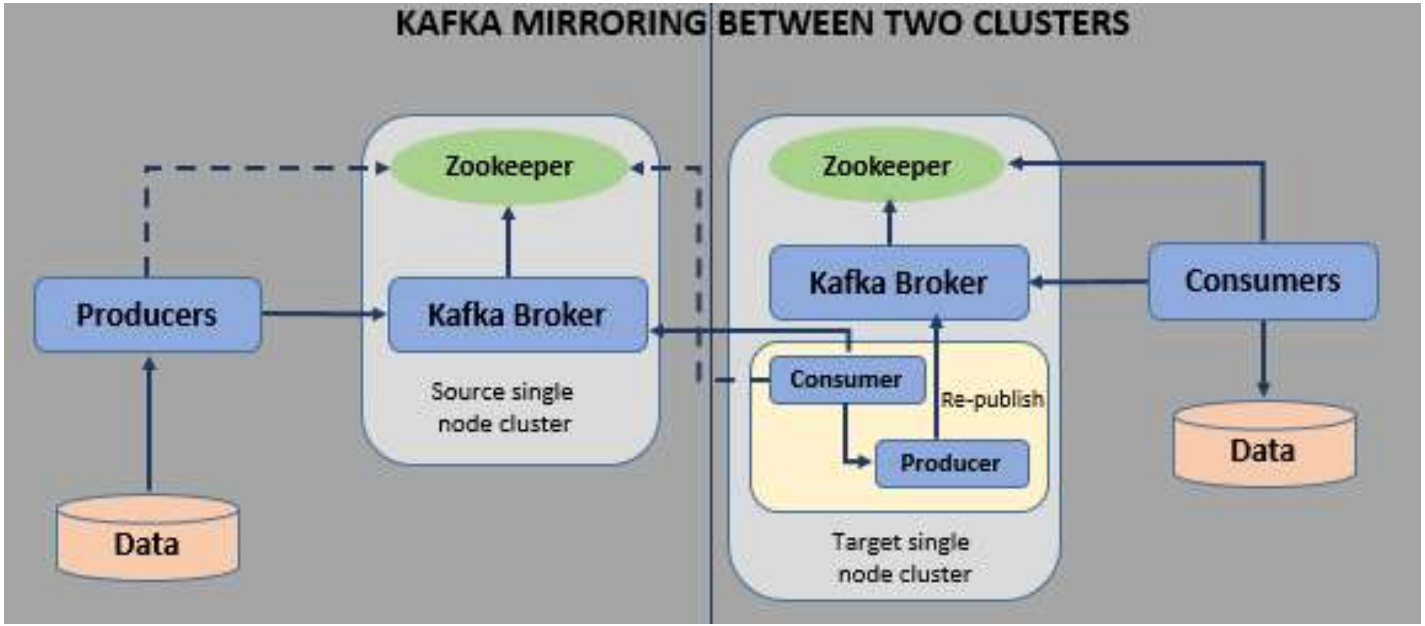
- Provide a replica to another datacenter or AWS region
- Mirroring used for disaster recovery

  - Datacenter or region goes down
  - Cluster is used for normal fault-tolerance

- Mirroring can also be used for increased throughput

  - Scale consumers
  - scales reads

MirrorMaker Use cases



Mirror Maker Command line

- The binary to execute Mirror maker is
  kafka-mirror-maker.sh

- Options

  --whitelist specifies regex for topics to mirror
  --blacklist denotes the whitelist regex for the topics to exclude

- Using mirroring with broker config auto.create.topics.enable=true on destination cluster makes auto replication with no config possible

Mirror Maker Command line

- Mirror maker command line

```
bin/kafka-mirror-maker.sh –consumer-config consumer-config.properties \
                          --producer.config producer-config.properties \
                          --whitelist “.*”
```

- Pass consumer properties to read from 1st cluster
- Pass producer properties to write to 2nd cluster
- Specify that you want to replicate all topics cia whitelist regex

Apache Kafka – 360 View

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ☑ Kafka Disaster Recovery
- ❑ **Kafka Cluster Administration and Operations**
- ❑ Kafka REST Proxy
- ❑ Kafka Connect
- ❑ Kafka Streams

Kafka Cluster Administration and Operations

Kafka Pre-requisites

Supported operating Systems

- Red Hat Enterprise Linux / CentOS 6 (64-bit)
- Red Hat Enterprise Linux / CentOS 7 (64-bit)
- Ubuntu Precise (12.04) (64-bit)
- Ubuntu Trusty (14.04) (64-bit)
- Debian 7
- SUSE Linux Enterprise Server (SLES) 11 SP3 (64-bit)

Kafka Pre-requisites

Software Requirements

- yum and rpm (RHEL/CentOS/Oracle Linux)
- zypper and php_curl (SLES)
- apt (Debian/Ubuntu)
- scp, curl, unzip, tar, and wget
- OpenSSL
- Python
- Scala

Kafka Pre-requisites

Software Requirements

- yum and rpm (RHEL/CentOS/Oracle Linux)
- zypper and php_curl (SLES)
- apt (Debian/Ubuntu)
- scp, curl, unzip, tar, and wget
- OpenSSL
- Python
- Scala

Kafka Pre-requisites

Hardware for Kafka

- Kafka is meant to run on industry standard hardware.
- The machine specs for a Kafka broker machine will be similar to that of your Hadoop worker nodes based upon the workload needed.
- The basic configuration for productions are
  - Processor with four 2Ghz cores
  - Siz 7200 RPM SATA drives(JBOD or RAID10)
  - 32GB RAM Minimum
  - 1 GB Ethernet bonded

Kafka Pre-requisites

Cluster Sizing

- The most accurate way to model your use case is to simulate the load you expect on your own hardware, and you can do that using the load-generation tools that ship with Kafka.
- Size the cluster based on the amount of disk space required
- Size the cluster based on your memory requirements.
- Assuming readers and writers are fairly evenly distributed across the brokers in your cluster, you can roughly estimate of memory needs by assuming you want to be able to buffer for at least 30 seconds and compute your memory need as write_throughput*30.

Kafka Cluster Administration and Operations

Monitoring kafka

A Tool to manage Apache Kafka - Yahoo Kafka Manager

- Managing Multiple Clusters
- Easy Inspection of Cluster States (Topics, Consumers, Offsets, Brokers, Replica Distribution, Partition Distribution)
- Replica Election
- Generate Partition assignments with option to select Brokers to use
- Re Assignment of partition (based on generated assignments)
- Topic can be created with optional configs (0.8.1.1 has different configs than 0.8.2+)
- Topic can be deleted (only supported on 0.8.2+ ,set delete.topic.enable=true in broker config)

Kafka Cluster Administration and Operations

Monitoring kafka

A Tool to manage Apache Kafka - Yahoo Kafka Manager

- List Topic indicates Topics marked for deletion (only supported on 0.8.2+)
- Partition Assignments for Multiple topics with an option to select Broker
- Re Assignment of Partition for multiple topics
- Add partitions to existing Topic
- Update Config for existing Topic
- Optionally enable JMX polling for Broker level and Topic level metrics.
- Optionally filter out Consumers that do not have ids/ owners/ & offsets/ directories in Zookeeper.

Monitoring kafka

Pre-requisites of Kafka Manager

- Apache Zookeeper
- Apache Kafka
- Java 8 as root user
- SBT 13+

Monitoring kafka

Benefits of Kafka Manager

- It would be useful in Enterprise Level, where system Engineers don't have to always go through the CLI to complete their task
- Also on the Engineers end, they would be completing 5,000 tasks a day and its just one of them. It enables you to digest the Information pretty quickly.

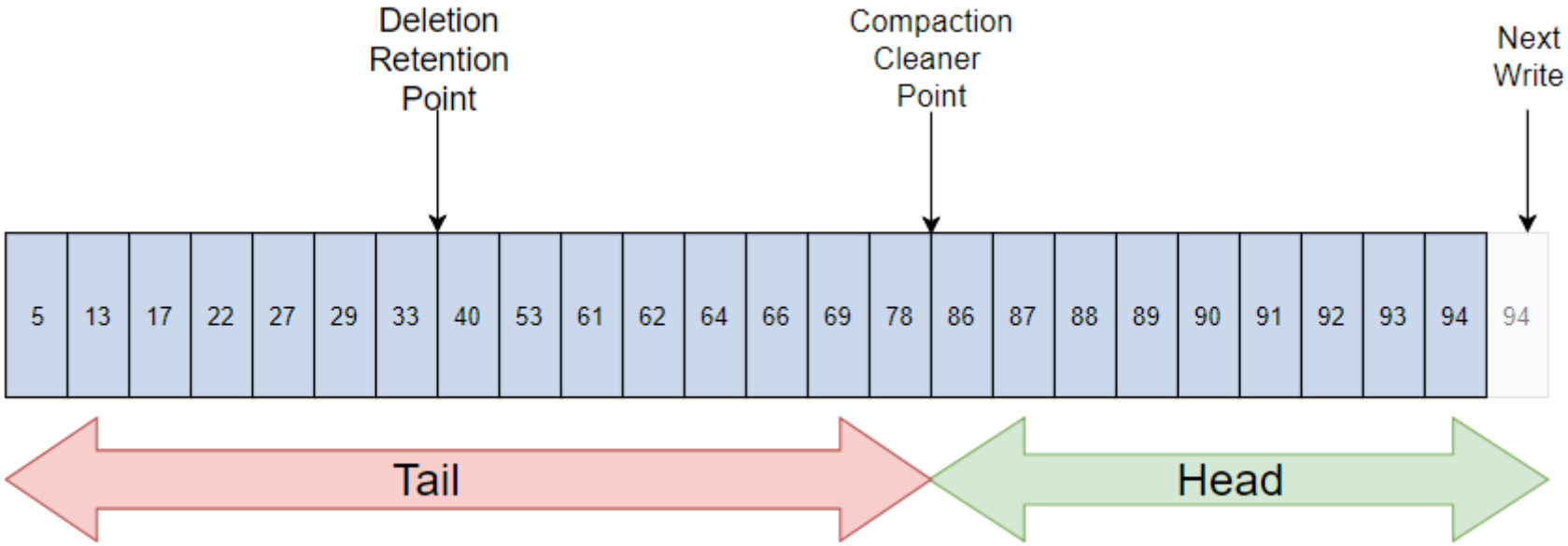
Kafka Log Compaction

- Log compaction retains at least the last known value for each record key for a single topic partition.
- Compacted logs are useful for restoring state after a crash or system failure.
- They are useful for in-memory services, persistent data stores, reloading a cache, etc.
- An important use case of data streams is to log changes to keyed, mutable data changes to a database table or changes to object in in-memory microservice.
- Log compaction is a granular retention mechanism that retains the last update for each key.
- log compacted topic log contains a full snapshot of final record values for every record key not just the recently changed keys.
- Kafka log compaction allows downstream consumers to restore their state from a log compacted topic.

Kafka Log Compaction

- With a compacted log, the log has head and tail.
- The head of the compacted log is identical to a traditional Kafka log.
- New records get appended to the end of the head.
- All log compaction works at the tail of the log
- Only the tail gets compacted
- Records in the tail of the log retain their original offset when written after being rewritten with compaction cleanup.

Kafka Log Compaction



Kafka Log Compaction

- All compacted log offsets remain valid, even if record at offset has been compacted away as a consumer will get the next highest offset.
- Kafka log compaction also allows for deletes.
- A message with a key and a null payload acts like a tombstone, a delete marker for that key.
- Tombstones get cleared after a period.
- Log compaction periodically runs in the background by recopying log segments.
- Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers.

Kafka Log Compaction Cleaning

- Topic config min.compaction.lag.ms gets used to guarantee a minimum period that must pass before a message can be compacted.
- The consumer sees all tombstones as long as the consumer reaches head of a log in a period less than the topic config delete.retention.ms (the default is 24 hours).
- Log compaction will never re-order messages, just remove some
- Partition offset for a message never changes.
- Any consumer reading from the start of the log sees at least final state of all records in the order they were written.

Kafka Log Cleaner

- Recall that a Kafka topic has a log.
- A log is broken up into partitions and partitions are divided into segments which contain records which have keys and values.
- The Kafka Log Cleaner does log compaction.
- The Log cleaner has a pool of background compaction threads.
- These threads recopy log segment files, removing older records whose key reappears recently in the log. Each compaction thread chooses topic log that has the highest ratio of log head to log tail.
- Then the compaction thread recopies the log from start to end removing records whose keys occur later in the log.

Kafka Log Cleaner

- As the log cleaner cleans log partition segments, the segments get swapped into the log partition immediately replacing the older segments.
- This way compaction does not require double the space of the entire partition as additional disk space required is just one additional log partition segment - divide and conquer.

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ☑ Kafka Disaster Recovery
- ☑ Kafka Cluster Administration and Operations
- ❑ **Kafka REST Proxy**
- ❑ Kafka Connect
- ❑ Kafka Streams

Kafka REST Proxy

- Kafka REST Proxy is an HTTP based proxy for your kafka cluster.
- The API supports many interactions with your cluster, including producing and consuming messages and accessing cluster metadata such as the set of topics and mapping of partitions to brokers.
- REST Proxy will be provided to the application team to push the Data to Kafka Topic.
- For the REST proxy the broker host public address should be provided to the External sources
- Ingesting messages into a stream processing framework that doesn’t yet support Kafka.
- REST proxy acts ad Web API layer of Kafka

Kafka REST Proxy

- Confluent Provides Kafka REST Proxy
- Cloudera and Hortonworks does not have Kafka REST.
- We can write our own Kafka REST API for the environment or we can download the reliable REST API from GitHub
- Confluent REST proxy helps in setting up the reliable REST proxy with the hep of confluent schema registry.

Confluent REST Proxy

- The REST Proxy should be able to expose all of the functionality of the Java producers, consumers, and command-line tools.

- Features of Confluent REST Proxy are below

- **Metadata**

  - Most metadata about the cluster – brokers, topics, partitions, and configs – can be read using GETrequests for the corresponding URLs.

- **Producers**

  - Instead of exposing producer objects, the API accepts produce requests targeted at specific topics or partitions and routes them all through a small pool of producers.

Confluent REST Proxy

- **Consumers**
  - The REST Proxy uses either the high level consumer (v1 api) or the new 0.9 consumer (v2 api) to implement consumer-groups that can read from topics.
  - Consumers are stateful and therefore tied to specific REST Proxy instances.
  - Offset commit can be either automatic or explicitly requested by the user. Currently limited to one thread per consumer; use multiple consumers for higher throughput.

- **Data Formats**
  - The REST Proxy can read and write data using JSON, raw bytes encoded with base64 or using JSON-encoded Avro.
  - With Avro, schemas are registered and validated against the Schema Registry

- **REST Proxy Clusters and Load Balancing**
  - The REST Proxy is designed to support multiple instances running together to spread load and can safely be run behind various load balancing mechanisms (e.g. round robin DNS, discovery services, load balancers)

Confluent REST Proxy does not supports

- **Admin operations**
  - Admins have to manually monitor the REST proxy operations and security

- **Multi-topic Produce Requests**
  - Currently each produce request may only address a single topic or topic-partition.
  - Most use cases do not require multi-topic produce requests, they introduce additional complexity into the API, and clients can easily split data across multiple requests if necessary

- **Most Producer/Consumer Overrides in Requests**
  - Only a few key overrides are exposed in the API
  - The reason is two-fold. First, proxies are multi-tenant and therefore most user-requested overrides need additional restrictions to ensure they do not impact other users.
  - Second, tying the API too much to the implementation restricts future API improvements; this is especially important with the new upcoming consumer implementation.

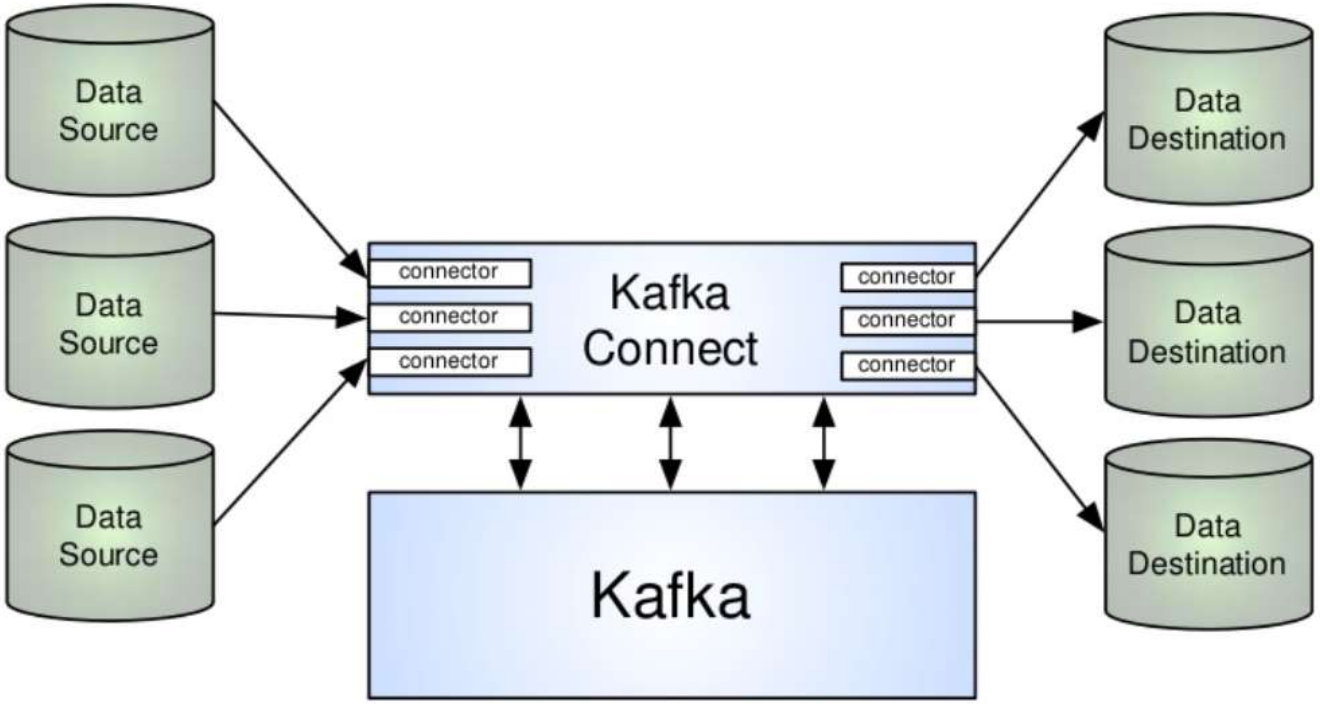
Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ☑ Kafka Disaster Recovery
- ☑ Kafka Cluster Administration and Operations
- ☑ Kafka REST Proxy
- ❑ **Kafka Connect**
- ❑ Kafka Streams

Kafka Connect

- Kafka Connect is a framework included in apache kafka that integrates kafka with other systems
- Its purpose is to make it easy to add new systems to your scalable and secure stream data pipelines.
- To copy data between Kafka and another system, users instantiate Kafka Connectors for the systems they want to pull data from or push data to.
- Source Connectors import data from another system (e.g. a relational database into Kafka) and Sink Connectors export data (e.g. the contents of a Kafka topic to an HDFS file).

Kafka Connect



How Connect differs from normal producer and consumer

- Producer and consumer provide complete flexibility to send any data to kafka pr process it in any way.

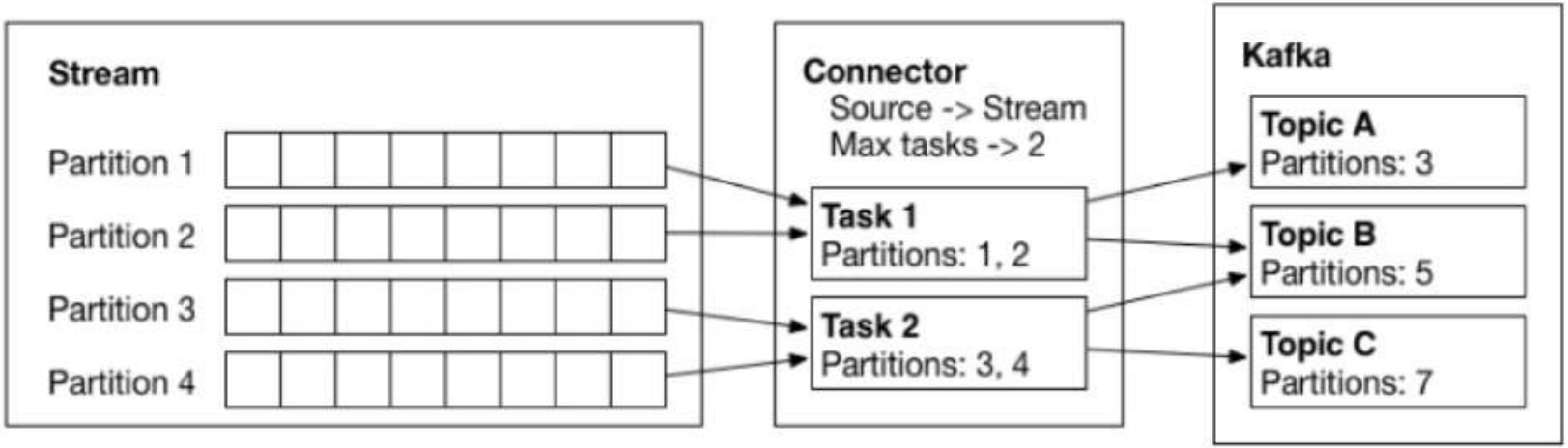
- Kafka connect's framework allows
  - Developers to create connectors that copy data to/from other systems
  - Operators/users to use connectors by writing configuration files and submitting them to connect
  - No Additional code required
  - Community, providers and 3rd party engineers to build reliable plugin's for common data sources and sinks
  - Deployments to deliver fault tolerance and automated load balancing are out of the box

Distributed vs Standalone Mode

- Connect works in two different modes
  - Distributed and Standalone
- Standalone mode runs in a single process – ideal for testing and development.
- Distributed mode runs in multiple processes on the same machine or spread across multiple machines.
- Distributed mode is fault tolerant and scalable

Kafka Connect components

- Connect has 3 different components
  - Connectors
  - Tasks
  - Workers



Source offset Management

- Connect tracks the offset that was last consumed for a source to restart tasks at the correct starting point after the failure. These offsets are different from kafka offset.
- In standalone mode, the source offset is tracked in a local file.
- In distributed mode, the source offset is tracked in a kafka topic

Working with Kafka Connect

- Configure connect-distributed.properties
- Configure Source of sink properties
- Start connect-distributed.sh on all the three brokers
- Verify the creation of topics connect-configs, connect-offsets and connect-status in the cluster

Apache Kafka – 360 View

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ☑ Kafka Disaster Recovery
- ☑ Kafka Cluster Administration and Operations
- ☑ Kafka REST Proxy
- ☑ Kafka Connect
- ❑ **Kafka Streams**

Kafka Streams

Lets work on kafka stream topic as hackathon mode. I will be spawning Hadoop cluster and cassandra cluster. We can establish a kafka connect and stream use cases to load data from logger to HDFS and Cassandra.

Apache Kafka – 360 View

Course Outline

- ☑ Course Introduction
- ☑ Introduction to Data Ingestion
- ☑ Apache Kafka Introduction
- ☑ Low-Level Architecture
- ☑ Advanced Kafka Producers and Consumers
- ☑ Schema management in kafka
- ☑ Kafka Security
- ☑ Kafka Disaster Recovery
- ☑ Kafka Cluster Administration and Operations
- ☑ Kafka REST Proxy
- ☑ Kafka Connect
- ☑ Kafka Streams

Thank you