Apache Kafka – 360 View

Apache Kafka

A high-throughput distributed messaging system.

Course Outline

- ❑ **Apache Kafka Introduction**
- ❑ Low-Level Architecture
- ❑ Advanced Kafka Producers and Consumers

Compiled by Navaneetha Babu C

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

Compiled by Navaneetha Babu C

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

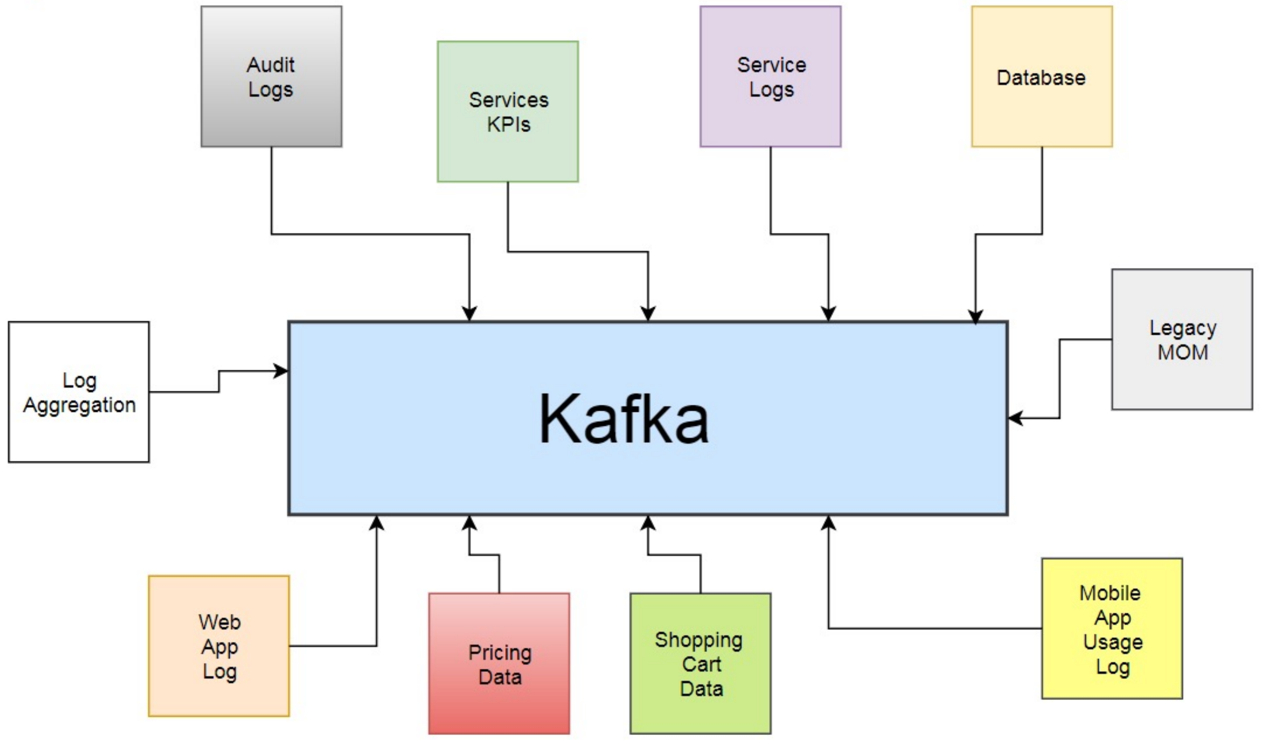
Why Apache Kafka

- ❑ 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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

Why Apache Kafka



Compiled by Navaneetha Babu C

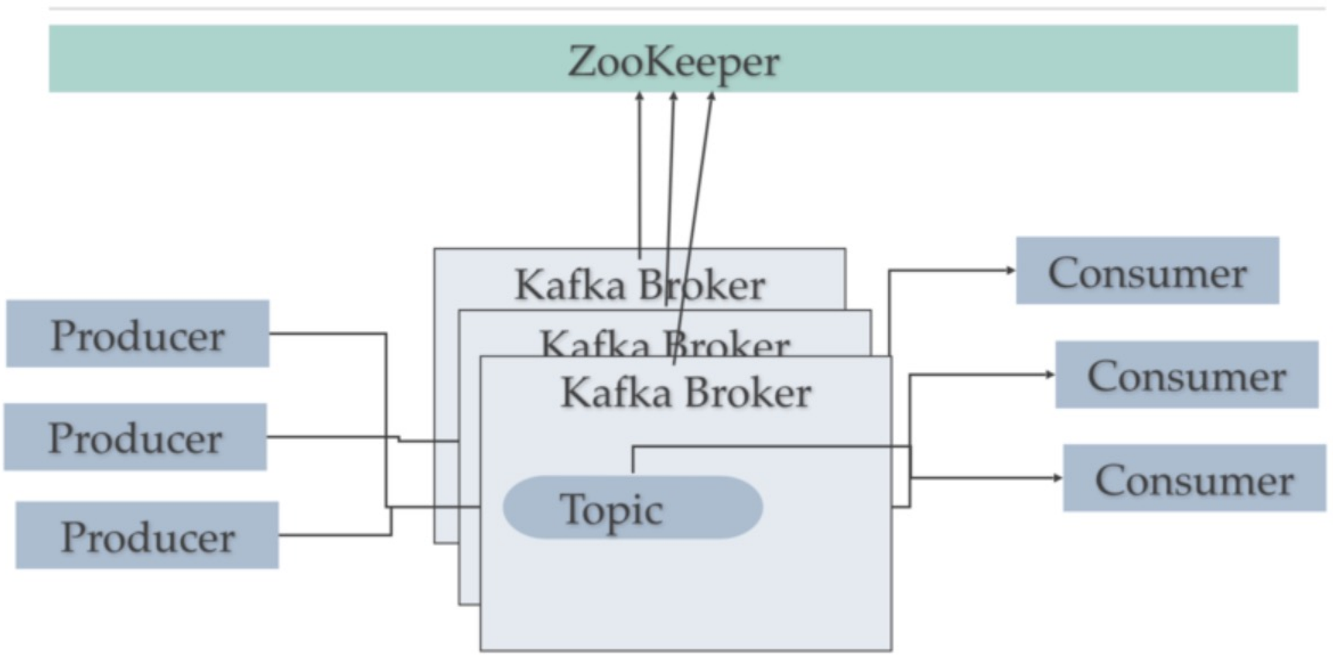
Apache Kafka Introduction

- ✓ 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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

Apache kafka Architecture



Compiled by Navaneetha Babu C

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.

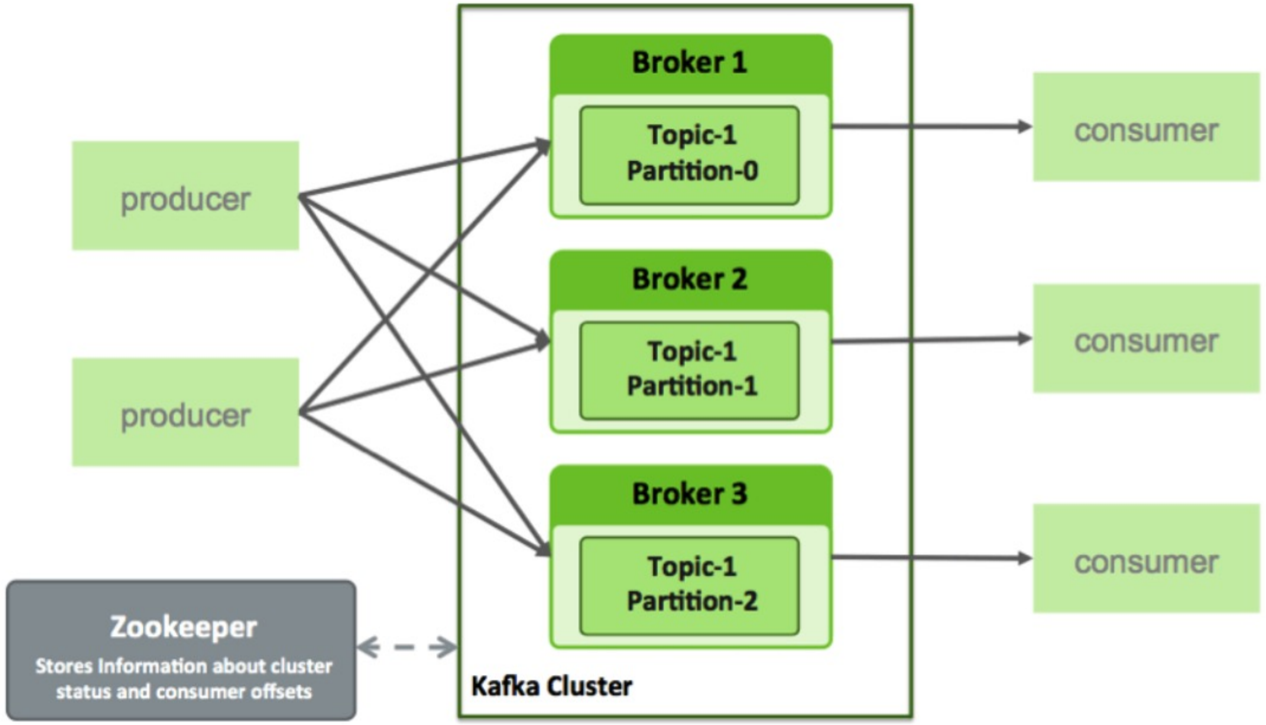
Apache Kafka Introduction

- ✓ 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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

Overview of key Concepts



Compiled by Navaneetha Babu C

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

- ❑ **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

Compiled by Navaneetha Babu C

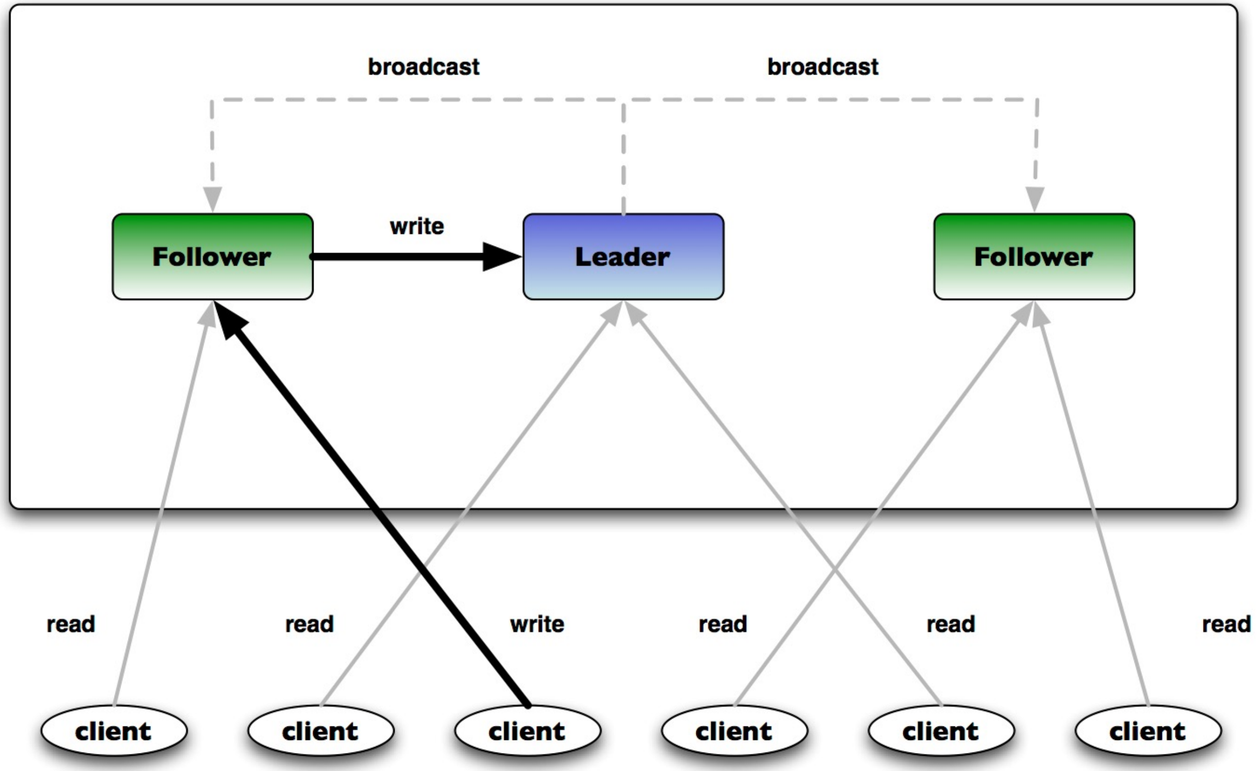
Apache Kafka Introduction

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, kafka, Cassandra and many other cluster. Each client machine communicates to one of the servers to retrieve information.

Apache Kafka Introduction

Apache Zookeeper



Compiled by Navaneetha Babu C

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 Kafka Introduction

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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

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

Compiled by Navaneetha Babu C

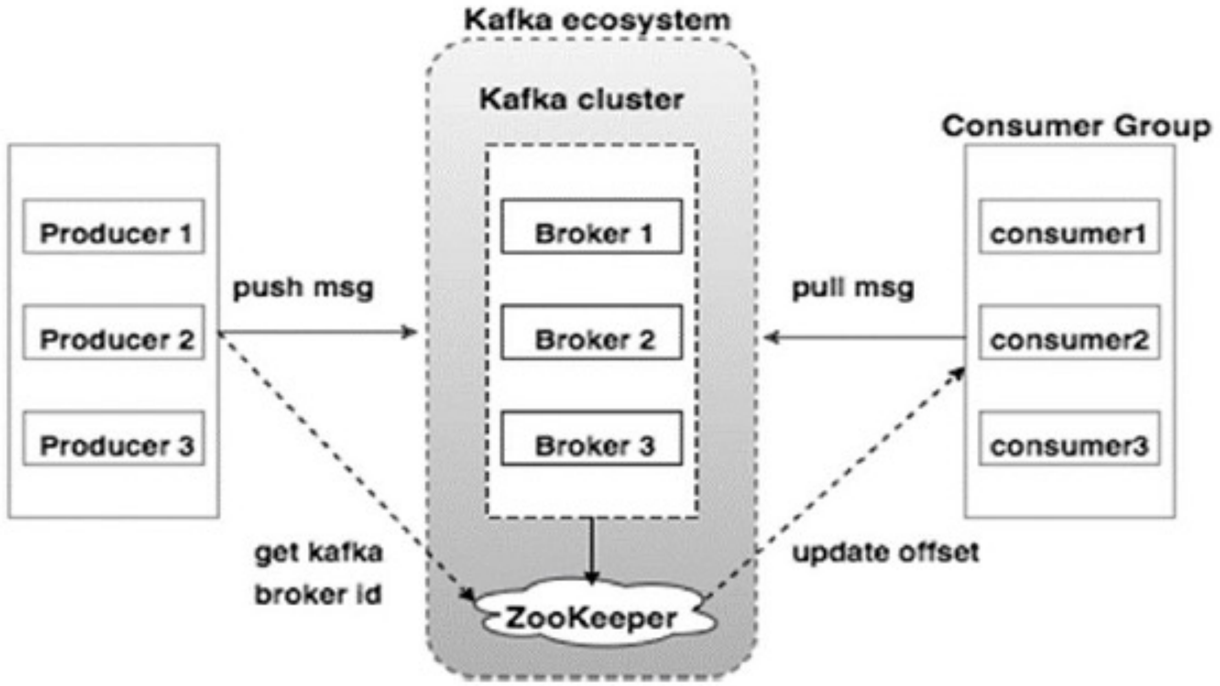
Apache Kafka Introduction

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



Compiled by Navaneetha Babu C

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

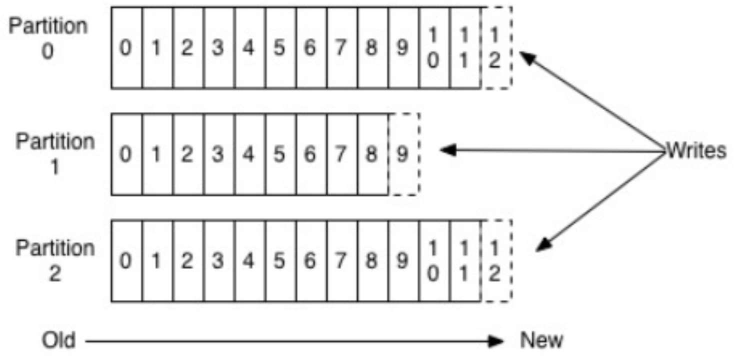
- ✓ Why Apache Kafka
- ✓ Apache Kafka Architecture
- ✓ Overview of Key Concepts
- ✓ Apache Zookeeper
- ✓ Cluster, Nodes and Kafka Brokers
- ❑ **Kafka Topic and Kafka APIs**
- ❑ Kafka partitions, Records and Keys
- ❑ Consumers and Producers
- ❑ Kafka Logs
- ❑ Kafka Partitions for Write Throughput
- ❑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

Compiled by Navaneetha Babu C

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:



Compiled by Navaneetha Babu C

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

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

Compiled by Navaneetha Babu C

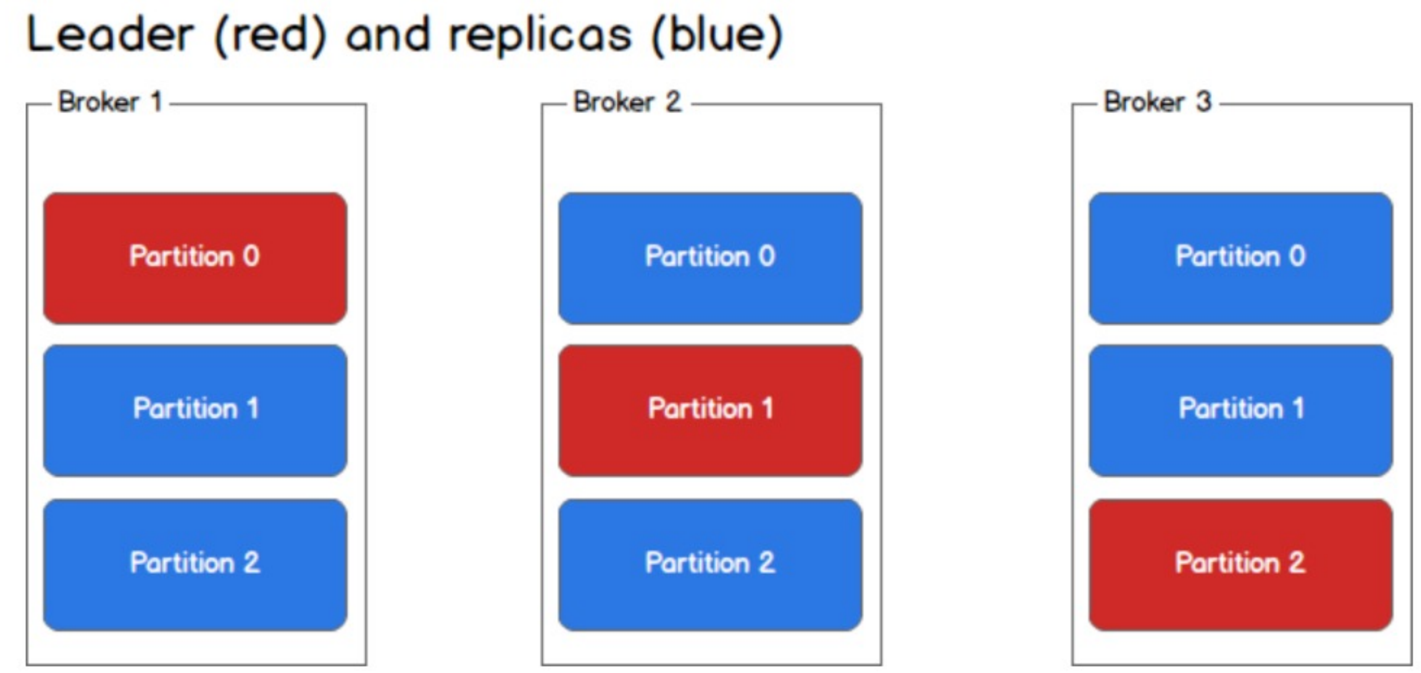
Apache Kafka Introduction

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.

Apache Kafka Introduction

Kafka Partitions



Compiled by Navaneetha Babu C

Apache Kafka Introduction

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.

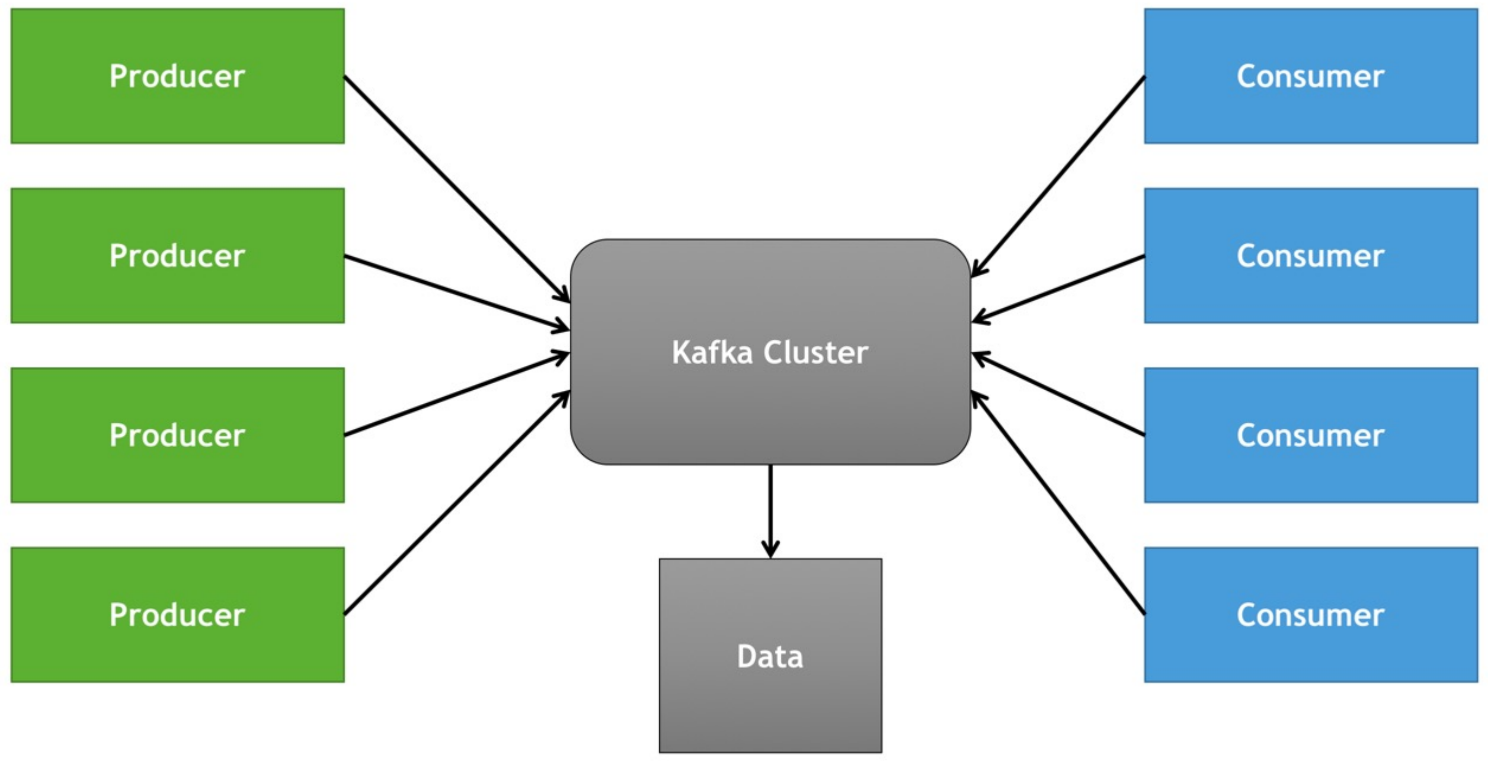
Apache Kafka Introduction

- ✓ 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

Compiled by Navaneetha Babu C

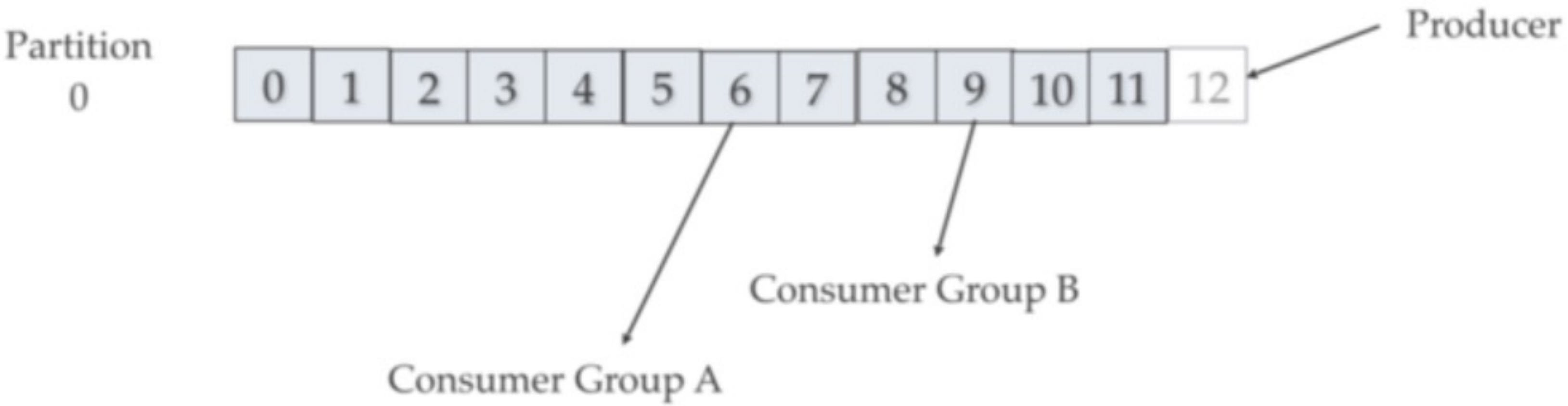
Apache Kafka Introduction

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

Apache Kafka Introduction

Kafka Consumer

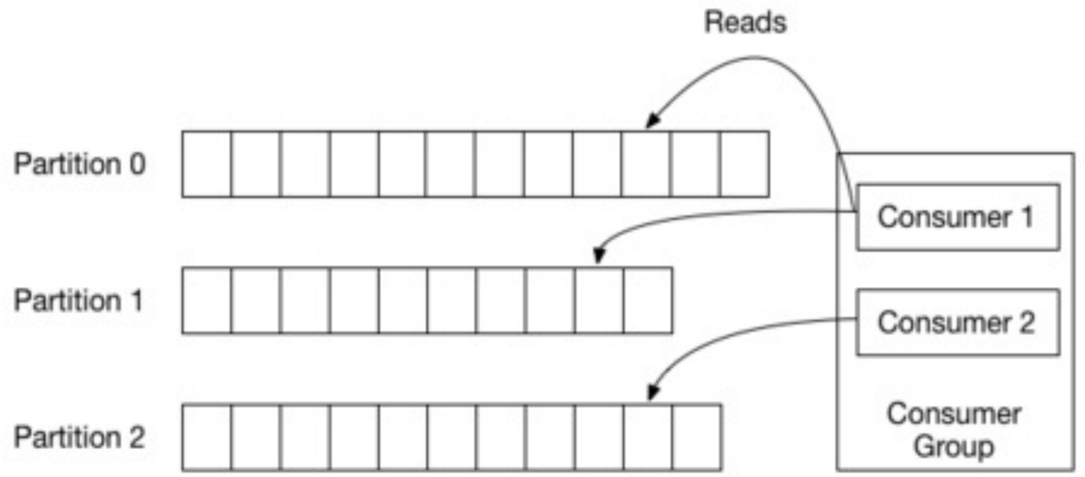


Figure 1: Consumer Group

Compiled by Navaneetha Babu C

Apache Kafka Introduction

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

Compiled by Navaneetha Babu C

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(offset) and partition pair.
- ❑ Each message has its value, offset, timestamp, key, message size, compression codec, checksum, and version of the message format.

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

Compiled by Navaneetha Babu C

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.

Apache Kafka Introduction

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

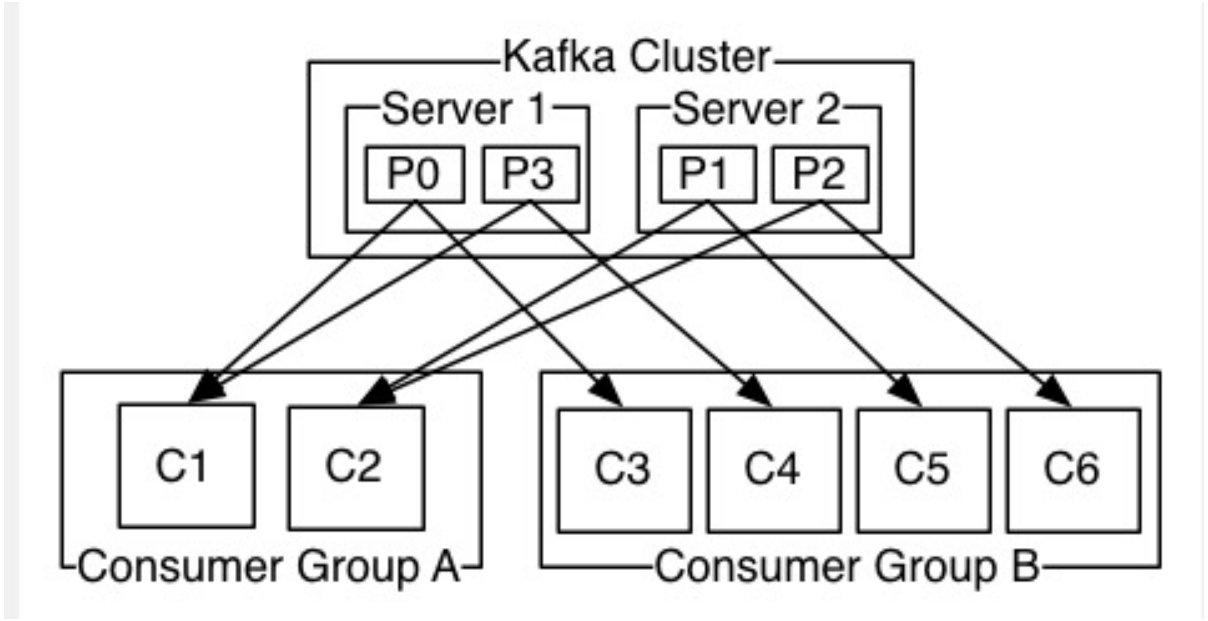
Apache Kafka Introduction

- ✓ 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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

❑ **Partitions for Consumer Parallelism**



Compiled by Navaneetha Babu C

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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

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.

Apache Kafka Introduction

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 Replica
- ❑ 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 failes.

Apache Kafka Introduction

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.

Apache Kafka Introduction

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).

Apache Kafka Introduction

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

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

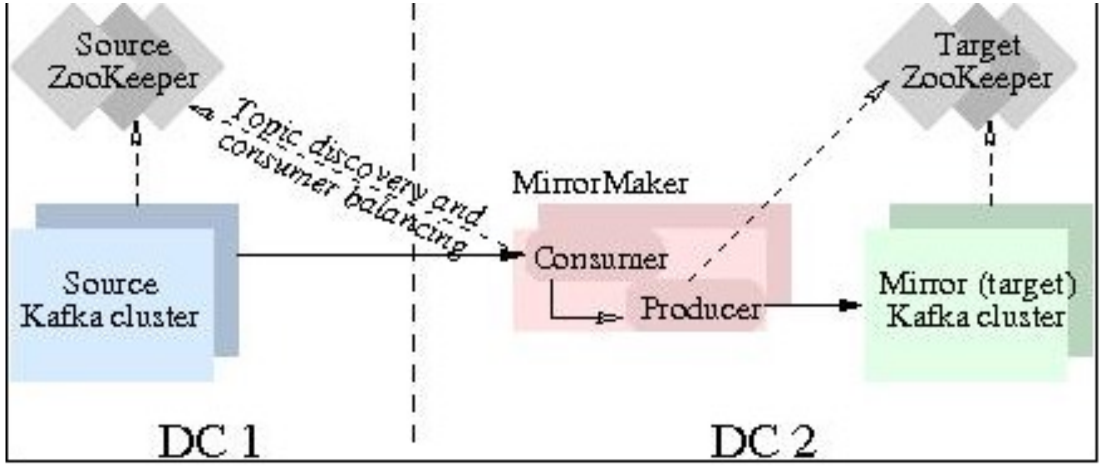
Compiled by Navaneetha Babu C

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.

Apache Kafka Introduction

Disaster Recovery – High Level



Kafka Mirror Maker will be discussed in upcoming chapter

Compiled by Navaneetha Babu C

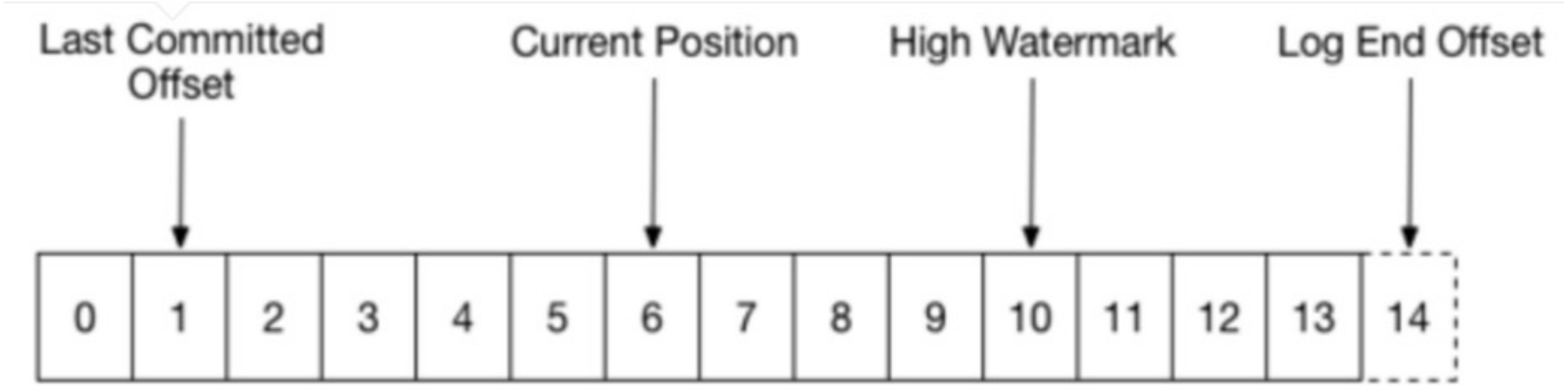
Apache Kafka Introduction

- ✓ 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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

High Water mark



Compiled by Navaneetha Babu C

Apache Kafka Introduction

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

Compiled by Navaneetha Babu C

Apache Kafka Introduction

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

Compiled by Navaneetha Babu C

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

Compiled by Navaneetha Babu C

Course Outline

- [✓] Apache Kafka Introduction
- ❑ **Low-Level Architecture**
- ❑ Advanced Kafka Producers and Consumers

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

Compiled by Navaneetha Babu C

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 no 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.

Low Level Architecture

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.

Low Level Architecture

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.

Compiled by Navaneetha Babu C

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.

Course Outline

- [✓] Apache Kafka Introduction
- [✓] Low-Level Architecture
- [ ] **Advanced Kafka Producers and Consumers**

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

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, 16384 * 4);
```

Advanced Kafka Producers and Consumers

Advanced Producer

- [x] 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

Compiled by Navaneetha Babu C

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);
```

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.

Advanced Kafka Producers and Consumers

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

Compiled by Navaneetha Babu C

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

Advanced Kafka Producers and Consumers

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
```

Compiled by Navaneetha Babu C

Advanced Kafka Producers and Consumers

Advanced Producer

- [x] Batching by Time and Size
- [x] Compression
- [x] Async Producers and Sync Producers
- [x] Default partitioning
- [ ] **Custom Partitioning**

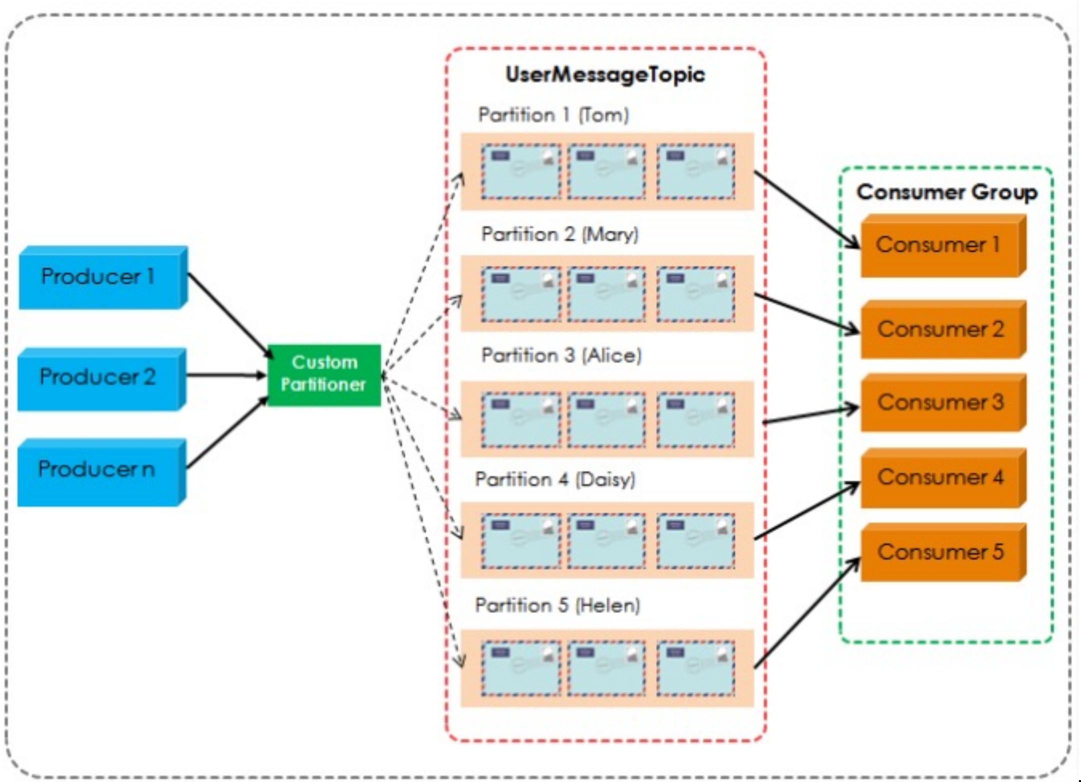
Advanced Consumer

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

Compiled by Navaneetha Babu C

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;
}
```

Compiled by Navaneetha Babu C

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

Advancced Kafka Producer and Consumer

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

Advancced Kafka Producer and Consumer

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

- [x] Batching by Time and Size
- [x] Compression
- [x] Async Producers and Sync Producers
- [x] Default partitioning
- [x] Custom Partitioning

Advanced Consumer

- [x] Consumer Poll
- [ ] **At most once message semantics**
- [ ] At least once message semantics
- [ ] Exactly once message semantics
- [ ] Using ConsumerRebalanceListener

Compiled by Navaneetha Babu C

Advancced Kafka Producer and Consumer

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

Compiled by Navaneetha Babu C

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

Compiled by Navaneetha Babu C

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**

Compiled by Navaneetha Babu C

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 gets 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.

Advanced Kafka Producer and Consumer

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

Thank you