Getting to Know Kafka

Welcome to **Kafka**, a platform to build real-time streaming applications. **Kafka - Premiera Ola** is the first course in the series of courses covering all the aspects of Kafka.

This course will walk you through the following concepts:

* Overview of Kafka & its **fundamental components**.
* **Publish-Subscribe** messaging workflow
* Kafka as a **messaging** system
* Kafka as a **storage** system
* **Stream Processing** using Kafka
* Final Assessment

##### How Kafka Came to be ..

During the initial days of big data, the focus was mainly on batch processing. In batch processing, applications would be run daily or weekly to load and analyze data from big data stores.

More recently, businesses have an increased need for handling **real-time data feeds**, i.e. analyzing and processing data and events as they happen.

To meet these demands, engineers at **LinkedIn** developed and open sourced **Kafka**, a stream processing platform which scales on commodity hardware.

* Kafka is a publish-subscribe messaging system, written in **Scala** and **Java**, that is fast, distributed and durable.
* Kafka is fault-tolerant and enables you to build distributed applications that scale on commodity hardware.

What is a Messaging System?

Before we explore Kafka, let's understand the types of messaging systems.

A **messaging system** is a medium that allows data transfer from one application to another so that the applications can focus on data without worrying about how to share it.

The two types of messaging patterns are:

* **Point to Point** Messaging System
* **Publish-Subscribe** Messaging System.

In **Point to Point** messaging system, senders send messages to a queue and receivers consume messages from the queue.

But there is a restriction that a particular message can be consumed by a maximum of only one receiver.

The message disappears from the queue, once the message is consumed by the receiver.

n **Publish-Subscribe** messaging system, senders, also known as **publishers**, classifies messages and publish them to a **topic**. Receivers, or **subscribers**, can receive messages only on subscribing to that **topic**.

Unlike point to point, in Publish-Subscribe messaging system,

* A message on a topic is broadcast to all subscribed consumers.
* Consumers can subscribe to multiple topics to receive messages.

**pache Kafka** is a distributed **publish-subscribe** messaging system used for collecting and delivering high volumes of data with low latency, similar to a traditional message broker.

Apache Kafka was originated at LinkedIn and became an open source project in 2011. Scala and Java are used to develop Kafka.

* Kafka stores messages coming from multiple applications called **producers**.
* The messages get partitioned into **partitions** and based on some classification written to different **topics**.
* The messages in a partition are indexed and stored with a **timestamp**.
* Multiple applications called **consumers** polls messages from partitions.
* Kafka runs as a **cluster of servers**.
* Each topic partition is replicated over the nodes of the Kafka Cluster.
* **TCP protocol** is used for communication between clients and Kafka nodes.

We can see this in detail on the upcoming topics.

Kafka is used for :

* Building real-time **streaming pipelines** that move data between different applications.
* Building real-time **streaming applications** that are capable of processing streams of data.
* Building a fault tolerant **storage system** that stores streams of records.

We will be discussing the above points in detail in the upcoming topics.

* **Reliability**: Kafka's distributed design, topic partitioning, and data replication over servers make it reliable.
* **Scalability**: Kafka system exists as a cluster of brokers. The number of brokers can grow over time when more data comes. Any failure of an individual broker in a cluster is handled by the system providing uninterrupted service.
* **Durability**: Disk-based data retention makes Kafka durable. Messages remain on the disk based on the retention rule configured on a per-topic basis. Even if a consumer falls backs due to any reason, the data continue to reside in the Broker till the retention period and is not lost.
* **High-Performance**: All the above features make Kafka a High-Performance messaging system.
* A **Kafka topic** is a category or feed name under which messages are stored.
* A **Kafka producer** publishes messages to a topic, which may be subscribed by zero or more **consumers**.
* As shown in the figure, the Kafka cluster maintains a **partitioned log** for each **topic**. Each of the partitions contains messages or records in an immutable ordered sequence.

Partitions

A **topic partition** is a structured commit log to which the records are continually appended. For each topic, Kafka keeps a minimum of one partition.

* Each record in the partition is assigned a sequential id called as the **offset**, which uniquely identifies each of them within the partition.
* The partitions enable the topic to scale beyond a single server and act as the unit of parallelism.

##### What is a Producer ?

* Kafka producers **publish messages** to one or more Kafka topics.
* Every time a producer sends a message to a broker, the broker **appends them** to the corresponding topic’s partition. Producers can also send messages to a partition of their choice.
* Producers write to a single leader so that each write is served by a separate broker which helps in **load balancing**.

##### What is a Consumer ?

* A consumer subscribes to a topic and consumes published messages by pulling data from the brokers.
* Consumers read from a single partition so that you can scale the throughput of message consumption similar to message production.
* If the number of consumers is more than the number of partitions then some consumers will remain idle as they have no partitions to read from.
* If the number of partitions is greater than the number of consumers, then each consumer will receive messages from multiple partitions.
* If the number of consumers is equal to the number of partitions, then each consumer reads messages in order from exactly one partition.

##### Consumers & Consumer Groups

Consumers can be organized into **consumer groups** for a given topic.

Each message published on a topic will be delivered to one consumer instance within each subscribed consumer group. These consumer instances may either be in separate processes or on separate machines.

* If all the consumer instances are within the same consumer group, then the records will be load balanced over the instances.
* If all the consumer instances are within different consumer groups, then each record will be broadcast to all the consumer processes.
* The **Offset or position of a consumer** in the partition log is the only data retained for that consumer.
* The consumer controls the offset.
* When a consumer reads records, the offset advances linearly along the partition log.
* The consumer can read data from any position in the partition log - it can move back to an older offset to re-read older data or jump ahead to the latest record and start consuming from there.
* Being a **distributed system**, Kafka runs in a **cluster** of machines, where each node in the cluster is called a Kafka **broker**.
* A Kafka cluster is a Kafka distribution with more than one broker.
* A Kafka cluster will expand without downtime.
* Each broker may hold zero or more partitions of a topic. For example, if you have a topic with 24 partitions and a cluster with 3 Kafka brokers, each one will hold 8 partitions of the topic.
* Kafka and Zookeeper will handle the load distributions among these partitions and redistribute them correctly when any broker goes down.

Leader and Followers

Each of the partitions held by a broker is replicated in the Kafka cluster for fault recovery. Each topic partition has one broker acting as a **leader** for that partition.

Nodes following leader instructions are called as followers.

* A leader is the node that handles all read and write requests for a given partition.
* It updates the followers or replicas with new data.
* If a leader fails, a follower takes over as the new leader.

ssume we have a Kafka cluster with three brokers, and topic partitions replicated over them.

As shown in the figure, for partition 0, broker 1 acts as the leader and brokers 2 & 3 are followers (replicas).

The read and write requests for a partition are handled by the **leader** and the **followers** replicate the leader across the nodes of the cluster.

**Each broker in the cluster will be a leader for some of its partitions and a follower for others, to maintain proper load balancing.**

* **Zookeeper** is a distributed centralized service that coordinates/manages large sets of hosts.
* Zookeeper is used to provide a configuration service, naming registry, synchronization, and group services in distributed applications.

##### Role of Zookeeper in Kafka

Kafka uses Zookeeper for the following:

**1. Electing a controller** :

* The controller is one among the many brokers responsible for maintaining the leader/follower relationship for all the partitions.
* When a node crashes or shuts down, the controller tells other replicas to become partition leader replacing the one on the node, that is going away.
* Zookeeper elects a controller, makes sure there is only one, and elects a new one it if it crashes.
* **2. Cluster membership**:
* Zookeeper monitors which brokers are alive and part of the cluster.
* **3. Topic configuration:**
* Zookeeper keeps track of topics, its partitions and replicas, who is the preferred leader and what configuration overrides are set for each topic.
* **4. Quotas:**
* Zookeeper tracks how much data each client is allowed to read and write.
* **5. ACLs:**
* Zookeeper tracks the following: Who is allowed to read and write to which topic, What are the consumer groups which exist, Who are their respective members and What is the latest offset each group received from each partition.

##### Queuing and Publish-Subscribe in Kafka

Kafka acts as both **Publish-Subscribe** and **Queue-based** messaging system.

In both cases, Producers send messages to a topic and Consumers read them from the topic.

Consumers are associated with ConsumerGroup using GroupId.

Consumers having same GroupId belong to a ConsumerGroup.

* If all consumers belong to different consumer groups, then all the Consumer Groups will consume messages (This is a Publish-Subscribe model ).
* If all consumers belong to the same consumer group, then the partitions will be evenly distributed among consumers in the consumer group. (This is a Queuing Model ).

##### Workflow as Publish-Subscribe Messaging System

**1.** Producers send messages to a topic at regular intervals.

**2.** Kafka broker stores all messages to the partitions configured for that topic, such that the messages are equally divided among the partitions of the topic.

**3.** Consumer subscribes to the topic.

**4.** On subscription, Kafka will send the current offset of the topic to the consumer. It then saves a copy of the current offset in Zookeeper ensemble.

**5.** The consumer then requests Kafka for new messages at regular intervals.

ontinues...

**6.** Once received from the producer, Kafka forwards the message to the consumer.

**7.** The consumer receives the message and processes it.

**8.** Once processed, the consumer sends an acknowledgment to Kafka broker.

**9.** On receiving the acknowledgment, Kafka broker changes offset to the new value and updates it in Zookeeper.

\*\*10.\*\*The above flow goes on repeating until the consumer stops the request.

**11.** At any time, the consumer can rewind/skip to the desired offset and get subsequent messages.

##### Workflow as Queue Messaging System

**1.** Producer sends messages to the topic "Topic-01" at regular intervals.

**2.** Broker stores those messages to the partitions configured for the topic "Topic-01", such that the messages are equally divided among the partitions of the topic.

**3.** A consumer with GroupId "Group-01" subscribes to the topic "Topic-01".

**4.** Kafka communicates with the consumer using the same steps as in Publish-Subscribe messaging system.

Workflow as Queue Messaging System Continues...

**5.** Another consumer with the same GroupId Group-01 also subscribes to the same topic Topic-01.

**6.** Now, the data is shared between the two consumers in the ***consumer group***, such that each topic partition is read by a consumer in the ***consumer group***.

**7.** This sharing of partitions go on until the \*number of consumers in the ***consumer group*** grows to the *total number of the topic partitions*.

**8.** Now, if again another consumer with the same GroupId Group-01 subscribes to the same topic Topic-01, it has to wait till any other consumers within the ***consumer group*** unsubscribe.

##### Kafka as a Messaging system

As discussed in the beginning of the course, traditional messaging has two models - **Queuing** and **Publish-Subscribe**.

In Kafka, the consumer group divides processing of messages among its consumer instances, similar to a queue. Again, Kafka broadcasts messages to all subscribing consumer groups, as with Publish-Subscribe.

**Thus, Kafka combines the strength of both these message models, enabling it to easily scale.**

##### Kafka as a Messaging system

* Kafka also provides **better ordering guarantees** than a traditional messaging system using **topic partitions**.
* Kafka assigns topic partitions to each consumer within the consumer group in such a way that each partition is consumed by only one consumer in the group.
* This guarantees that the consumer is the sole reader of that partition, consuming the data in order.

Kafka as a Storage system

Kafka acts as a **efficient storage system** due to the following:

* It stores data in a distributed fashion with high performance.
* It writes the data to disk and replicates the data for fault-tolerance.
* Producers can wait for acknowledgement until the data write operation is complete, i.e. replicated and persisted.

A configurable **retention period** can be set to retain all published records in Kafka irrespective of whether they were consumed or not.

For example, if retention period is set as three days, a record will be available for consumption for three days after it is published. It will be discarded after the retention period.

Kafka Streams API

A stream processor in Kafka reads streams of data from input topics, processes this data and produces continuous streams of data to output topics. For this purpose, Kafka provides a fully integrated **Streams API**.

For example, an application might take in input streams of data and perform computations for handling out-of-order data, reprocess input as code changes etc. and then output a stream of transformed data.

* The input for the Streams API are the producer and consumer APIs.
* It uses Kafka for stateful storage.
* For fault tolerance among stream processor instances, it uses the same group mechanism.

2 of 6

**1. Stream -** Primary abstraction in Kafka Streams, it represents an unbounded and continuously updating data set. A stream contains a sequence of immutable data records which are ordered and fault tolerant.

**2. Stream processing application** - Program using the Kafka Streams Library to implement its computational logic through one or more processor topologies.

**3. Processor topology** - Graph of stream processors (nodes) that are connected by streams (edges).

**4.Stream processor** - Node in the processor topology. It denotes a processing step to operate on input stream data by receiving an input record at a time from its upstream processors in the topology, applying transformations, and consequently producing output records to its downstream processors.

Two special stream processors:

1. **Source Processor** - Does not have any upstream processors. It produces an input stream to its topology by consuming records from one or more Kafka topics and forwards it to downstream processors.
2. **Sink Processor** - Does not have downstream processors. It sends any received records from its upstream processors to a specified Kafka topic.