# Apache Kafka

- Introduction
- What Kafka does?
- How Kafka Works
- Topology Fundamentals
- Cluster Architecture
- Integration with Spark
- Integration with Storm
- Hands on: Kafka Cluster Installation Steps

By**:**
Nimish Joshi

# Kafka

## Introduction:

There are vast volume of data is used in Big Data & there are two main challenges regarding the data. The first challenge is how to collect large volume of data and second challenge is to analyze the collected data. In order to overcome these challenges we must need a messaging system.

Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In compare to other messaging system, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

Apache Kafka was developed by LinkedIn & it was made an open source Apache project in the year 2011. Kafka is written in Scala and Java. It is fast, scalable and distributed by design.

## 1. What is Kafka & What is Does?

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well Apache Storm and Spark for real-time streaming data analysis.

Apache Kafka supports a wide range of use cases as a general-purpose messaging system for scenarios where high throughput, reliable delivery, and horizontal scalability are important.

Common uses include:

- Streaming Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation

Some of the important characteristics or benefits that make Kafka such an attractive option for these use cases include the following.

- Reliability:  Kafka is distributed, partitioned, replicated and fault tolerant
- Scalability:  Kafka messaging system scaled easily without down time.

- Durability: Kafka uses “Distributed Commit log” which means messages persists on disk as fast as possible, hence it is durable.
- Performance: Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

## 2. How Kafka Works?

Kafka’s system design can be thought of as that of a distributed commit log, where incoming data is written sequentially to disk. There are four main Components involved in moving data in and out of Kafka.

- Topic
- Producers
- Consumers
- Brokers

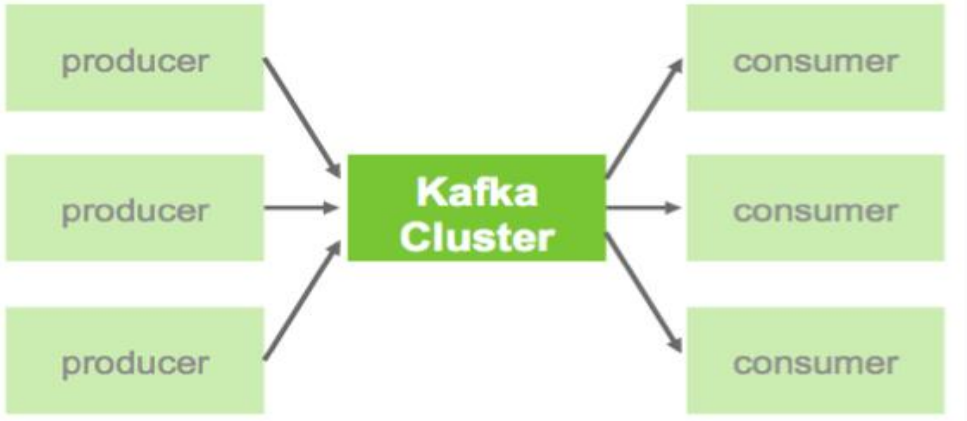
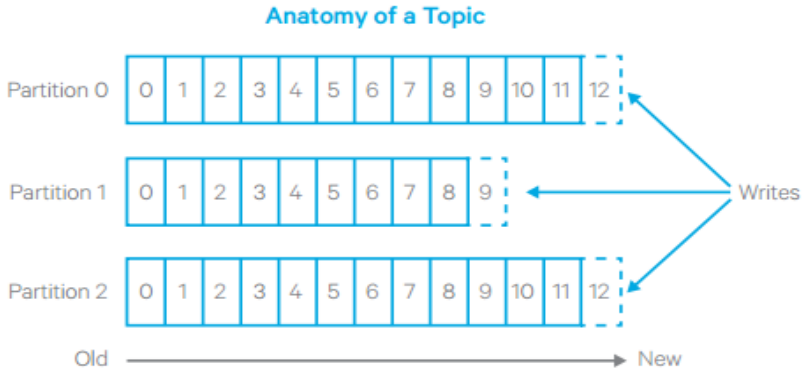



Figure: Kafka Components & anatomy of a Topic



## Topic

A stream of messages belonging to a particular category is called a topic. Data is stored in topics. Topics are split into partitions. For each topic, Kafka keeps a mini-mum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.

## Producers

Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer publishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producer can also send message to a partition of their choice.

## Consumers

Consumers read data from brokers. Consumers subscribe to one or more topics and consume published messages by pulling data from the brokers.

## Brokers

Brokers are simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic. Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition.

If there are N partitions in a topics and more than N broker (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic.

If there are N partitions in a topic and less than N brokers (n-m), each broker will have one or more partition sharing among them. This scenario is not recommended due to unequal load distribution among the broker.

# 3. Topology Fundamentals

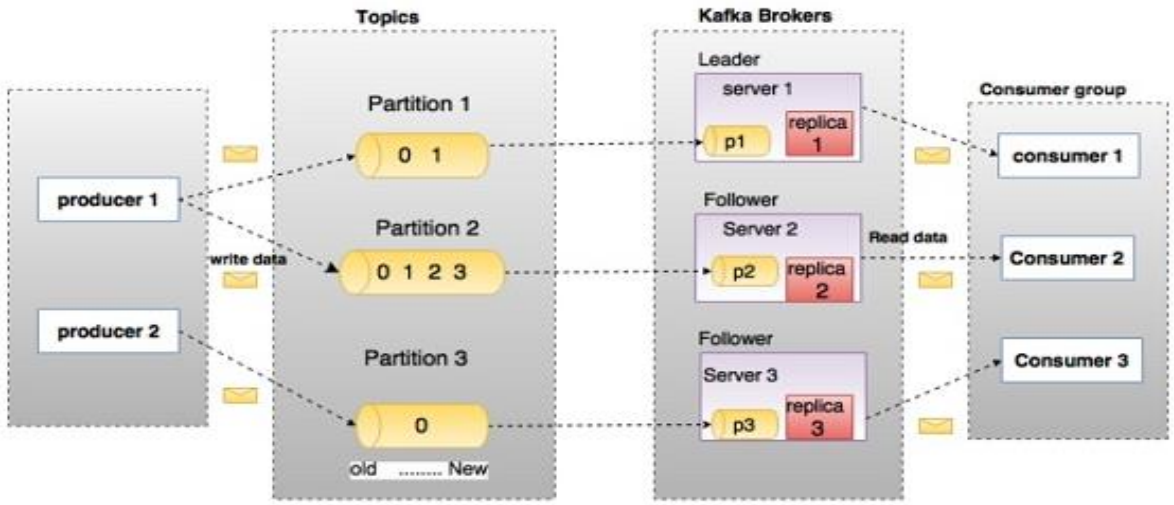



Figure: Typical Kafka Cluster Topology diagram

Before we go ahead with anything else, it is required to have some fundamental knowledge about its topology.

In the above diagram, a topic is configured into three partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0. The id of the replica is same as the id of the server that hosts it.

Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

Now let's understand some unknown terminology from the diagram:

## Kafka Cluster

Kafka's having more than one broker are called as Kafka Cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.

## Leader

"Leader" is the node responsible for all reads and writes for the given partition. Every partition has one server as a leasder.

### Follower

Node which follows leader instructions are called as follower. If the leader fails, one of the follower will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store.

## 4. Cluster Architecture

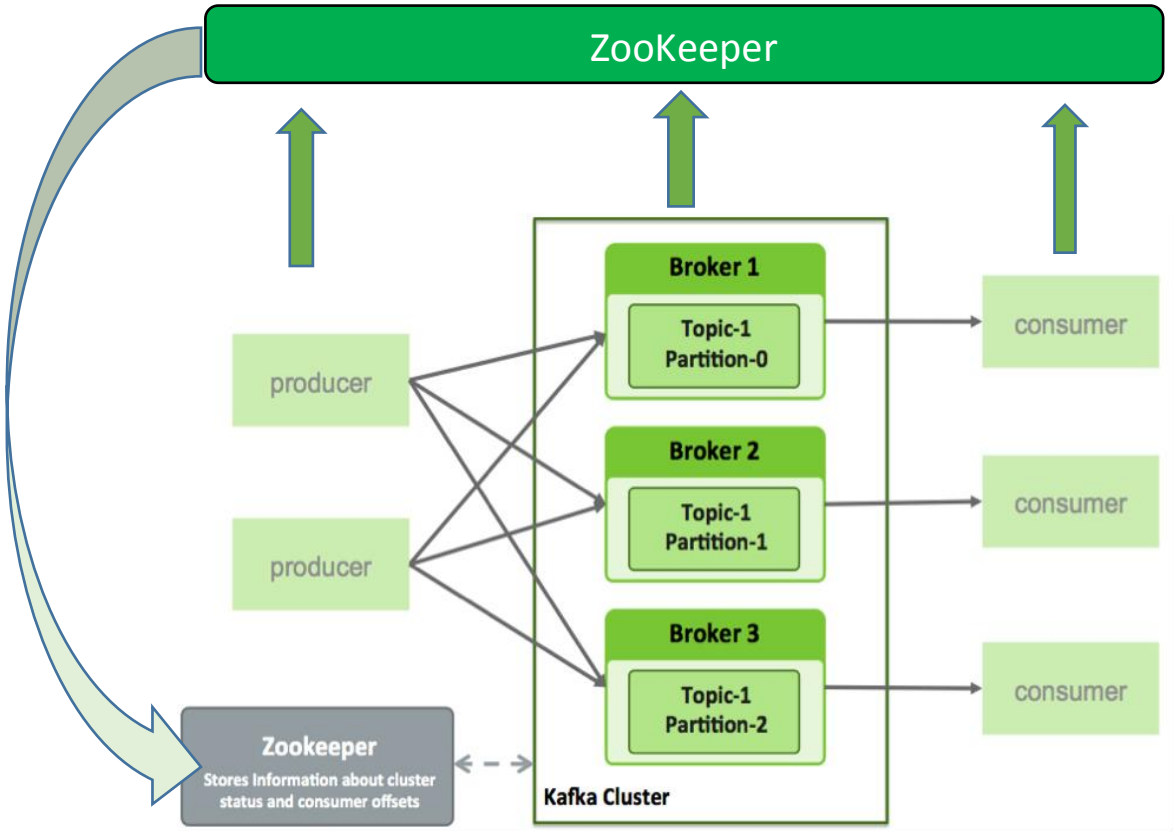



Figure: Kafka Cluster Diagram

One of the keys to Kafka's high performance is the simplicity of the brokers' responsibilities. In Kafka, topics consist of one or more Partitions that are ordered, immutable sequences of messages. Since writes to a partition are sequential, this design greatly reduces the number of hard disk seeks (with their resulting latency).

Another factor contributing to Kafka's performance and scalability is the fact that Kafka brokers are not responsible for keeping track of what messages have been consumed – that responsibility falls on the consumer. In traditional messaging systems such as JMS, the broker bore this responsibility, severely limiting the system's ability to scale as the number of consumers increased.

For Kafka consumers, keeping track of which messages have been consumed (processed) is simply a matter of keeping track of an Offset, which is a sequential id number that uniquely identifies a message within a partition. Because Kafka retains all messages on disk (for a configurable amount of time), consumers can rewind or skip to any point in a partition simply by supplying an offset value. Finally, this design eliminates the potential for back-pressure when consumers process messages at different rates.

**ZooKeeper**

ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.

**Kafka Directory**

The files in Kafka Directory look like this:

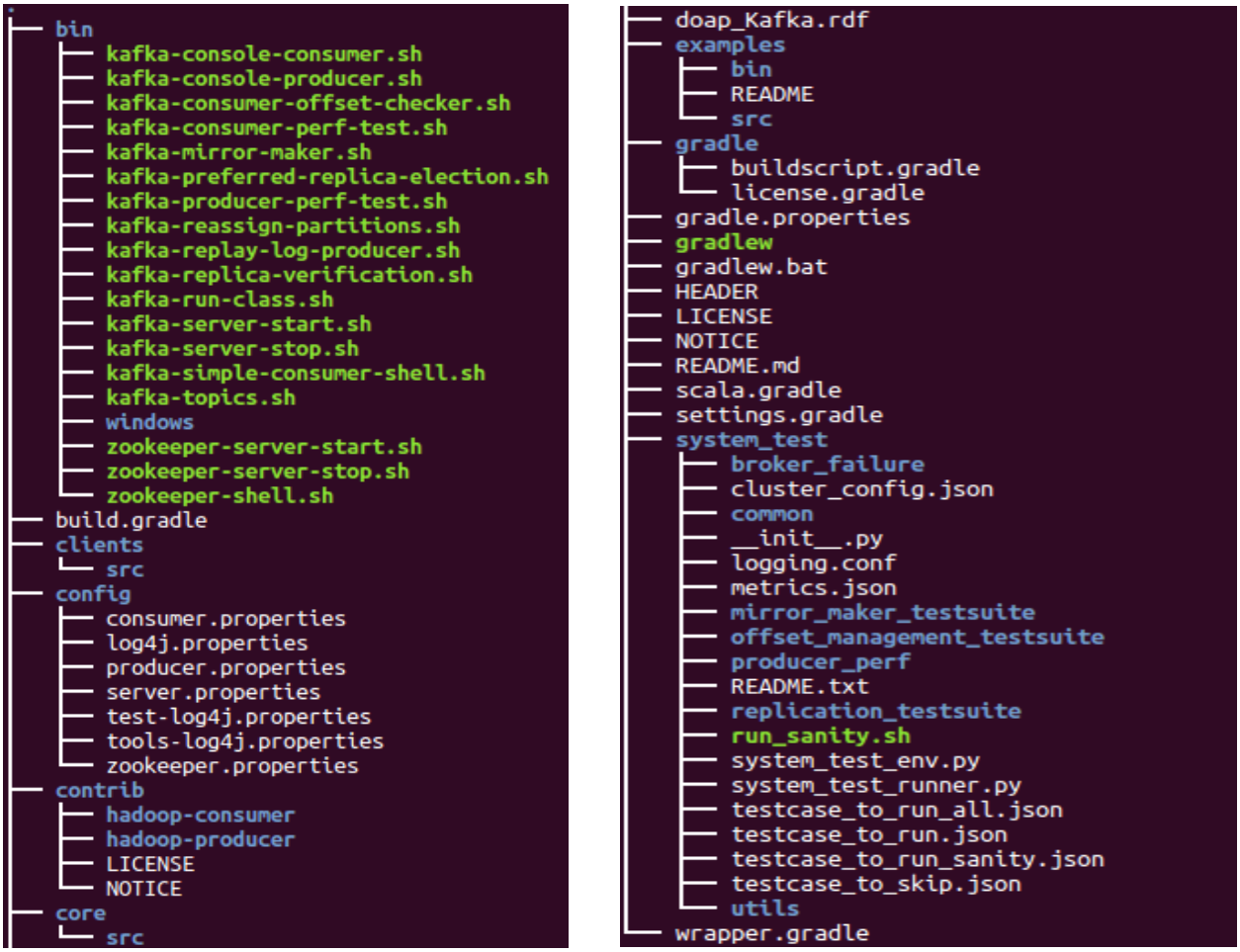



Figure: Kafka Directory tree structure

# 5. Integration with Spark

### About Spark

Spark Streaming API enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, etc., and can be processed using complex algorithms such as high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dash-boards. Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

### Integration with Spark

Kafka is a potential messaging and integration platform for Spark streaming. Kafka act as the central hub for real-time streams of data and are processed using complex algorithms in Spark Streaming. Once the data is processed, Spark Streaming could be publishing results into yet another Kafka topic or store in HDFS, databases or dashboards. The following diagram depicts the conceptual flow.

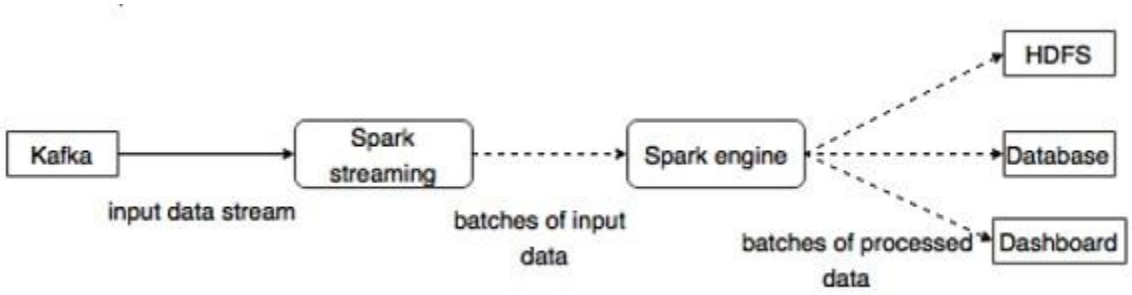



Figure: Conceptual Flow- Kafka integration with Spark

# 6. Integration with Storm

### About Storm

Storm was originally created by Nathan Marz and team at BackType. In a short time, Apache Storm became a standard for distributed real-time processing system that allows you to process a huge volume of data. Storm is very fast and a benchmark clocked it at over a million tuples processed per second per node. Apache Storm runs continuously, consuming data from the configured sources (Spouts) and passes the data down the processing pipeline (Bolts). Combined, Spouts and Bolts make a Topology.

### Integration with Storm

Kafka and Storm naturally complement each other, and their powerful cooperation enables real-time streaming analytics for fast-moving big data. Kafka and Storm integration is to make easier for developers to ingest and publish data streams from Storm topologies.

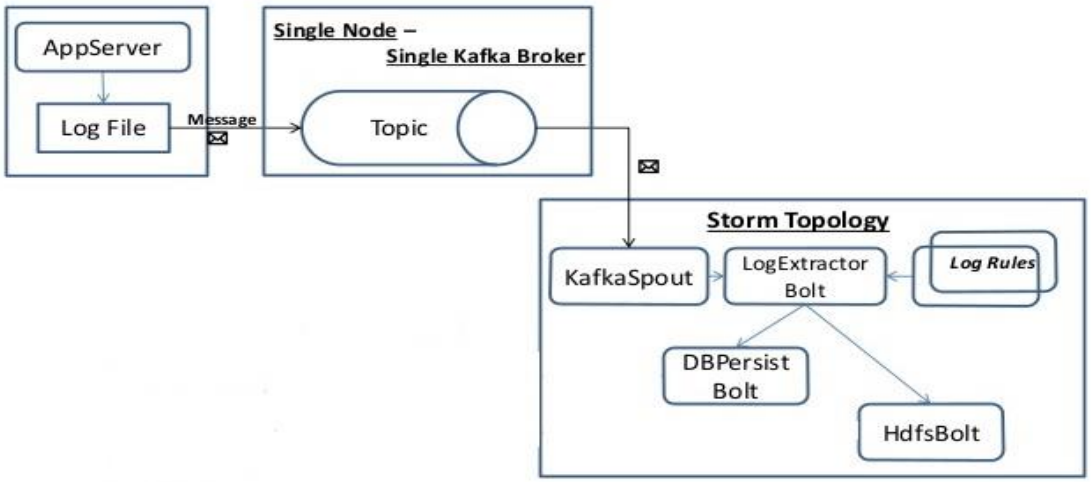



Figure: Conceptual flow – Kafka and Storm Integration

### Conceptual Flow

A spout is a source of streams. For example, a spout may read tuples off a Kafka Topic and emit them as a stream. A bolt consumes input streams, process and possibly emits new streams. Bolts can do anything from running functions, filtering tuples, do streaming aggregations, streaming joins, talk to databases, and more. Each node in a Storm topology executes in parallel. A topology runs indefinitely until you terminate it. Storm will automatically reassign any failed tasks. Additionally, Storm guarantees that there will be no data loss, even if the machines go down and messages are dropped.

# References


1. https://kafka.apache.org/
2. https://kafka.apache.org/documentation.html
3. https://www.tutorialspoint.com/apache_kafka/index.htm
4. http://hortonworks.com/apache/kafka/
5. http://www.bogotobogo.com/Hadoop/BigData_hadoop_Zookeeper_Kafka.php
6. http://www.slideshare.net/Sidhartha105/storm-kafka