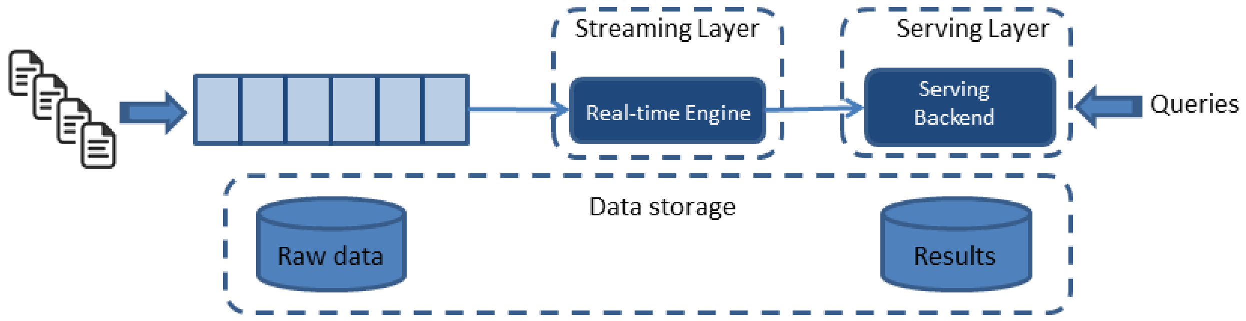
KAPPA ARCHITECURE

Submitted By: Wasama Jabeen

Contents

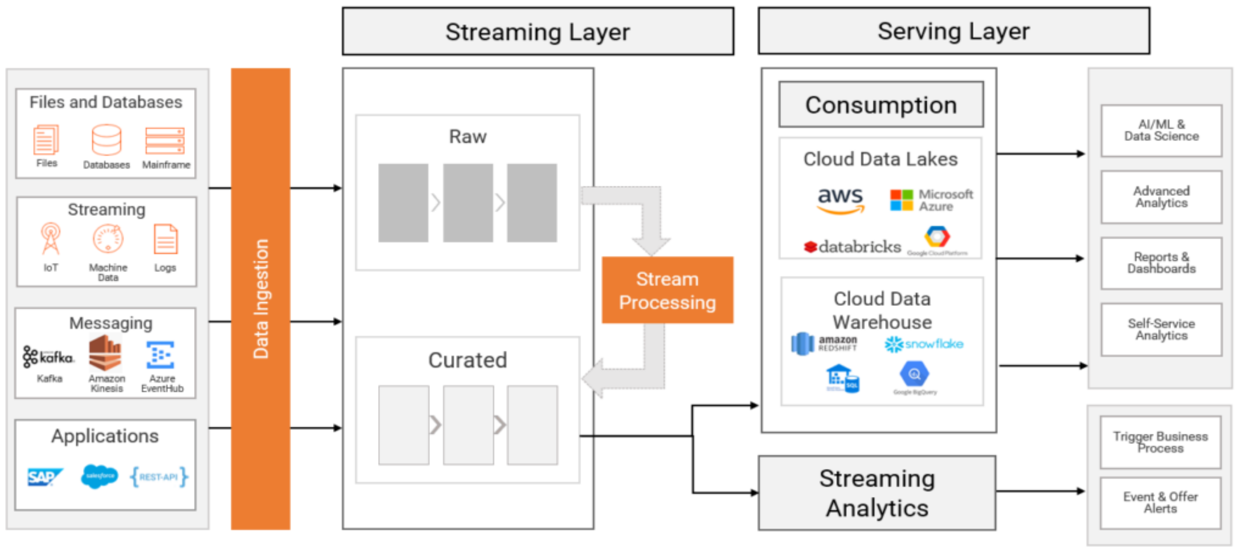
Kappa Architecture

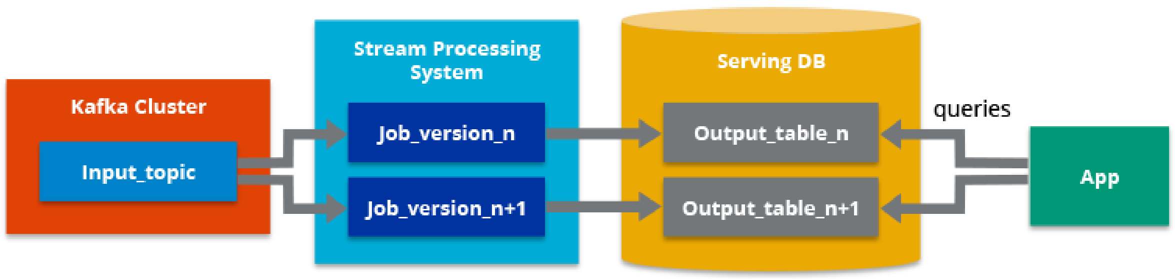
Kappa architecture is a streaming-first **architecture** deployment pattern – where data coming from streaming, IoT, batch or near-real time (such as change data capture), is ingested into a messaging system like Apache Kafka. A stream processing engine (like Apache Spark, Apache Flink, etc.)



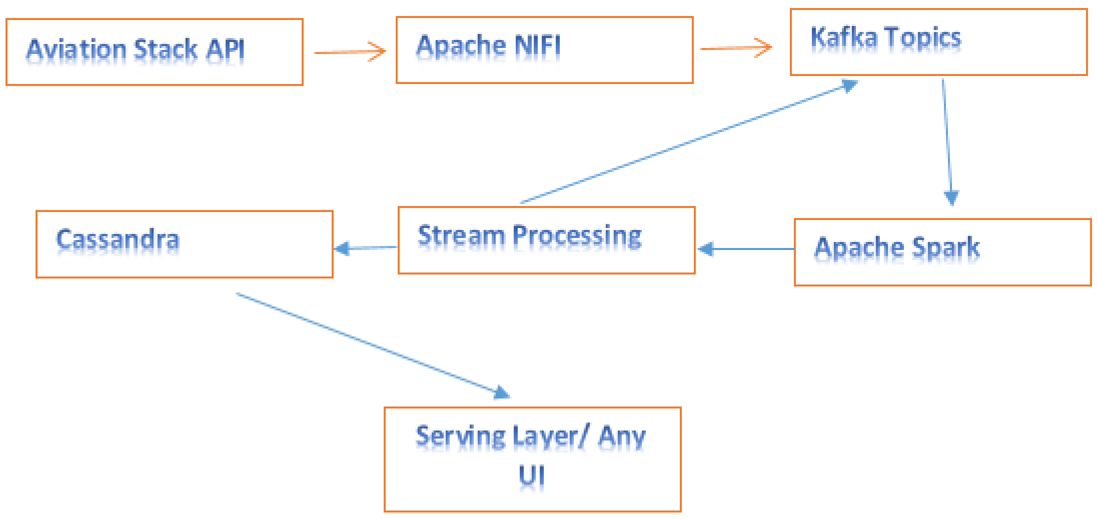
It can be used in architectures where the batch layer is not needed to fulfill the organization's quality of service needs, as well as in situations where specific transformations can be added to the streaming layer, like data quality techniques. As described above, in streaming-first implementation patterns, the Kappa architecture is used where data sources are both batch and real-time and where end-to-end latency specifications are rather tight.

You literally read the streaming data stored in parallel (assuming that the data in Kafka is correctly separated into different channels or 'partitions') and convert the data as if it were from a streaming source. With a sufficiently fast stream processing engine you may not need a separate technology that is optimized for batch processing. You simply read the stored streaming data in parallel (assuming the data in Kafka is appropriately split into separate channels, or “partitions”) and transform the data as if it were from a streaming source. Again, to allow low latency in the processing, this requires a high-speed stream processing engine.





My Project Use Case:



Event Streaming:

In the digital world, it plays the same role as our central nervous system does for our body, which involves regulating and messaging the whole body. It is the technological foundation for the 'always-on' future where software steadily detects and automates organizations, and where the user of software is more software.
Event Streaming is the implementation of real-time data processing from event sources such as databases, cloud servers, smart devices, cameras, and event streaming software applications. Then, after making needed adjustments and encoding, these stored event streams can be used in real time or for later use or can create any reaction to them. As per the criteria, these events can also be redirected to the appropriate locations. Thus, Event Streaming guarantees the transfer of information without delay with appropriately positioned information and in absolutely accurate pacing to the right location.

What can I use event streaming for?

Event streaming can be used to a large variation of use cases across a lot of industries and organizations. Few examples are below

- It can be used to process real-time financial transactions e.g., banks and stock exchange.
- To track the shipment in real time for logistic companies,
- Can be used to continuously gather and examine data from IOT devices in any business place or factory.
- To get the in-time customer interactions e.g., retail, travel, mobile applications and hotel.
- To track the in-time patient conditions' status for timely required treatment application specially in case of emergency,
- To keep all the divisions and stores up to date related to production in factories.
- To carry out the base for data platforms, micro services and the event-driven architectures

AviationStack API:

In order to provide a convenient way to access global aviation data for real-time and historical flights, the aviationstack API was created to enable passengers to access a detailed data collection of airline routes and other up-to-date aviation-related information. REST API requests are made using a basic HTTP GET URL structure, and answers are given in a lightweight JSON format.

I Created Free Account which allows 500 request with 100 records per request. My quota exceeded during implementation in which I utilized below to request.

Real-Time Flights

In real-time, the API is able to log flights and download flight status information. You may use the flight endpoint of the API along with optional parameters to filter the result collection to find real-time information about one or several flights.

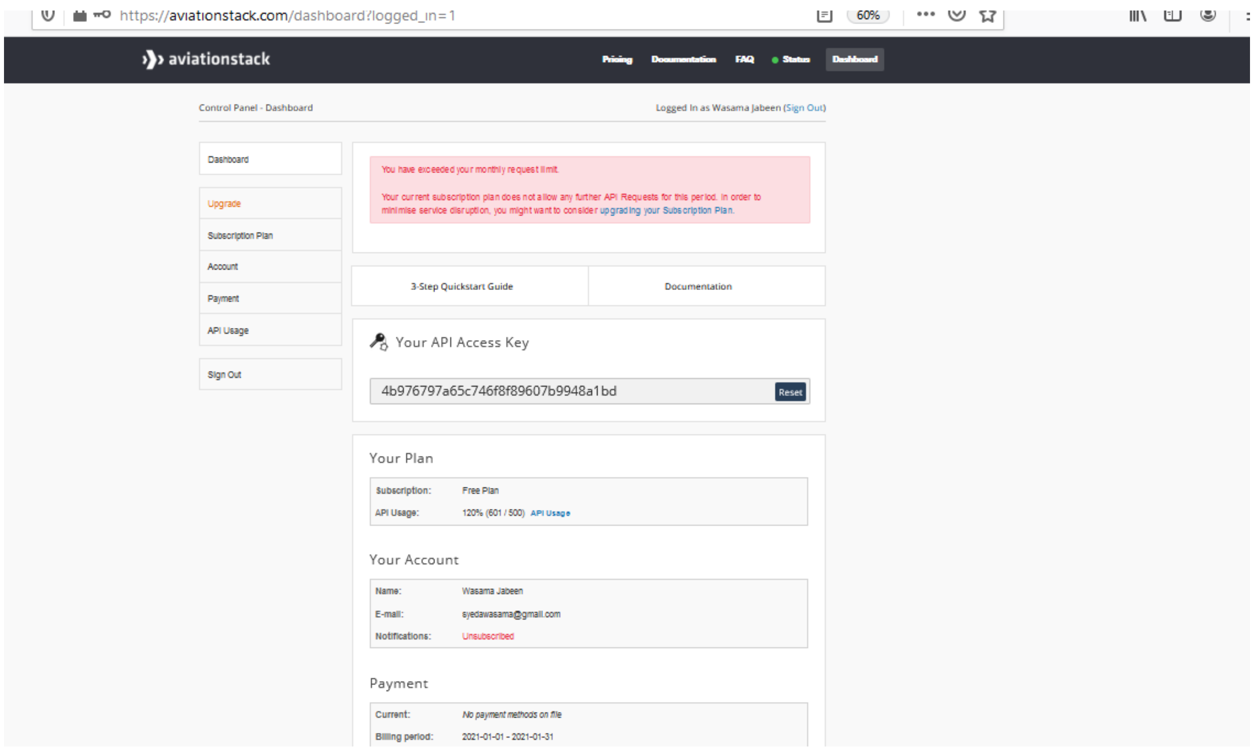
Example API Request:

https://api.aviationstack.com/v1/flights?access_key= YOUR_ACCESS_KEY

Example API Response:

```
{
   "pagination": {
      "limit": 100,
      "offset": 0,
      "count": 100,
      "total": 1669022
   },
   "data": [
      {
         "flight_date": "2019-12-12",
         "flight_status": "active",
         "departure": {
            "airport": "San Francisco International",
            "timezone": "America/Los_Angeles",
            "iata": "SFO",
            "icao": "KSFO",
            "terminal": "2",
            "gate": "D11",
            "delay": 13,
            "scheduled": "2019-12-12T04:20:00+00:00",
            "estimated": "2019-12-12T04:20:00+00:00",
            "actual": "2019-12-12T04:20:13+00:00",
            "estimated_runway": "2019-12-12T04:20:13+00:00",
            "actual_runway": "2019-12-12T04:20:13+00:00"
         },
         "arrival": {
            "airport": "Dallas/Fort Worth International",
            "timezone": "America/Chicago",
            "iata": "DFW",
            "icao": "KDFW",
            "terminal": "A",
            "gate": "A22",
            "baggage": "A17",
            "delay": 0,
            "scheduled": "2019-12-12T04:20:00+00:00",
            "estimated": "2019-12-12T04:20:00+00:00",
            "actual": null,
            "estimated_runway": null,
            "actual_runway": null
        },
        "airline": {
            "name": "American Airlines",
            "iata": "AA",
            "icao": "AAL"
        },
        "flight": {
            "number": "1004",
            "iata": "AA1004",
            "icao": "AAL1004",
            "codeshared": null
        },
        "aircraft": {
           "registration": "N160AN",
           "iata": "A321",
           "icao": "A321",
           "icao24": "A0F1BB"
        },
        "live": {
            "updated": "2019-12-12T10:00:00+00:00",
            "latitude": 36.28560000,
            "longitude": -106.80700000,
            "altitude": 8846.820,
            "direction": 114.340,
            "speed_horizontal": 894.348,
            "speed_vertical": 1.188,
            "is_ground": false
        }
    },
    [...]
  ]
}
```

Please note: The API response above has been shortened to show only one flight result for readability purposes.



Control Panel - Dashboard

Logged In as Wasama Jabeen (Sign Out)

You have exceeded your monthly request limit.

Your current subscription plan does not allow any further API Requests for this period. In order to minimise service disruption, you might want to consider upgrading your Subscription Plan.

Your API Access Key

4b976797a65c746f8f89607b9948a1bd

Your Plan

Subscription:	Free Plan
API Usage:	120% (601 / 500) API Usage

Your Account

Name:	Wasama Jabeen
E-mail:	syedawasama@gmail.com
Notifications:	Unsubscribed

Payment

Current:	*No payment methods on file*
Billing period:	2021-01-01 - 2021-01-31

Apache NIFI:

NiFi was built to automate the flow of data between systems. While the term 'dataflow' is used in a variety of contexts, we use it here to mean the automated and managed flow of information between systems. This problem space has been around ever since enterprises had more than one system, where some of the systems created data and some of the systems consumed data.

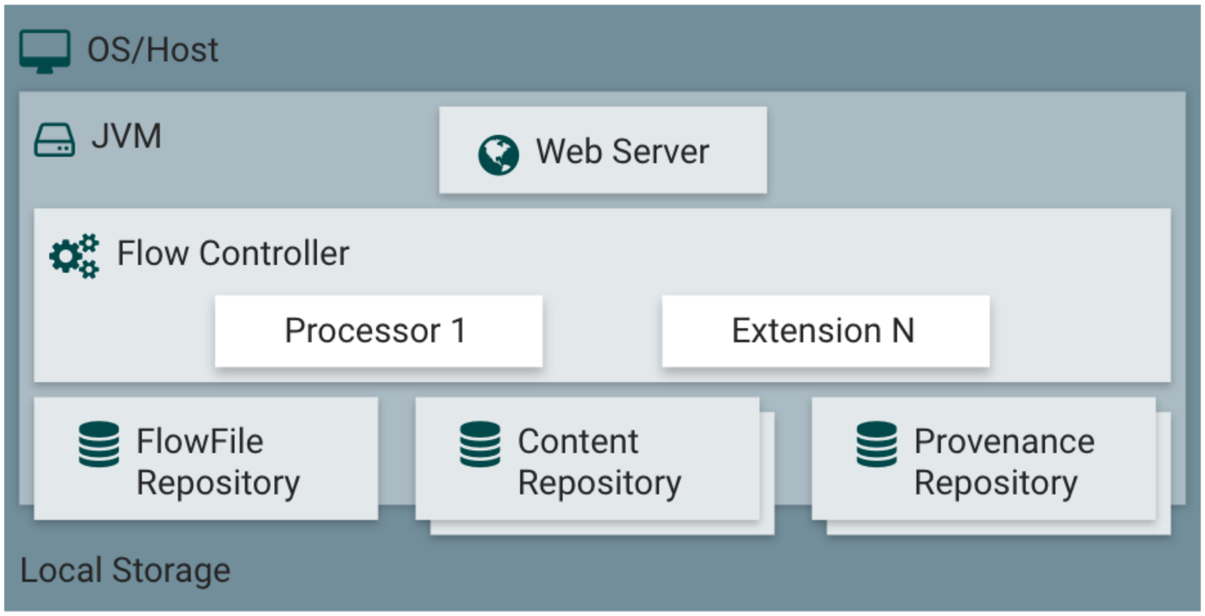
The core concepts of NiFi

NiFi’s fundamental design concepts closely relate to the main ideas of Flow Based Programming [fbp]. Here are some of the main NiFi concepts and how they map to FBP:

NiFi Term	FBP Term	Description
FlowFile	Information Packet	A FlowFile represents each object moving through the system and for each one, NiFi keeps track of a map of key/value pair attribute strings and its associated content of zero or more bytes.
FlowFile Processor	Black Box	Processors actually perform the work. In [eip] terms a processor is doing some combination of data routing, transformation, or mediation between systems. Processors have access to attributes of a given FlowFile and its content stream. Processors can operate on

NiFi Term	FBP Term	Description
		zero or more FlowFiles in a given unit of work and either commit that work or rollback.
Connection	Bounded Buffer	Connections provide the actual linkage between processors. These act as queues and allow various processes to interact at differing rates. These queues can be prioritized dynamically and can have upper bounds on load, which enable back pressure.
Flow Controller	Scheduler	The Flow Controller maintains the knowledge of how processes connect and manages the threads and allocations thereof which all processes use. The Flow Controller acts as the broker facilitating the exchange of FlowFiles between processors.
Process Group	subnet	A Process Group is a specific set of processes and their connections, which can receive data via input ports and send data out via output ports. In this manner, process groups allow creation of entirely new components simply by composition of other components.

NiFi Architecture:



NiFi executes within a JVM on a host operating system. The primary components of NiFi on the JVM are as follows:

Web Server

The purpose of the web server is to host NiFi’s HTTP-based command and control API.

Flow Controller

The flow controller is the brains of the operation. It provides threads for extensions to run on, and manages the schedule of when extensions receive resources to execute.

Extensions

There are various types of NiFi extensions which are described in other documents. The key point here is that extensions operate and execute within the JVM.

FlowFile Repository

The FlowFile Repository is where NiFi keeps track of the state of what it knows about a given FlowFile that is presently active in the flow. The implementation of the repository is pluggable. The default approach is a persistent Write-Ahead Log located on a specified disk partition.

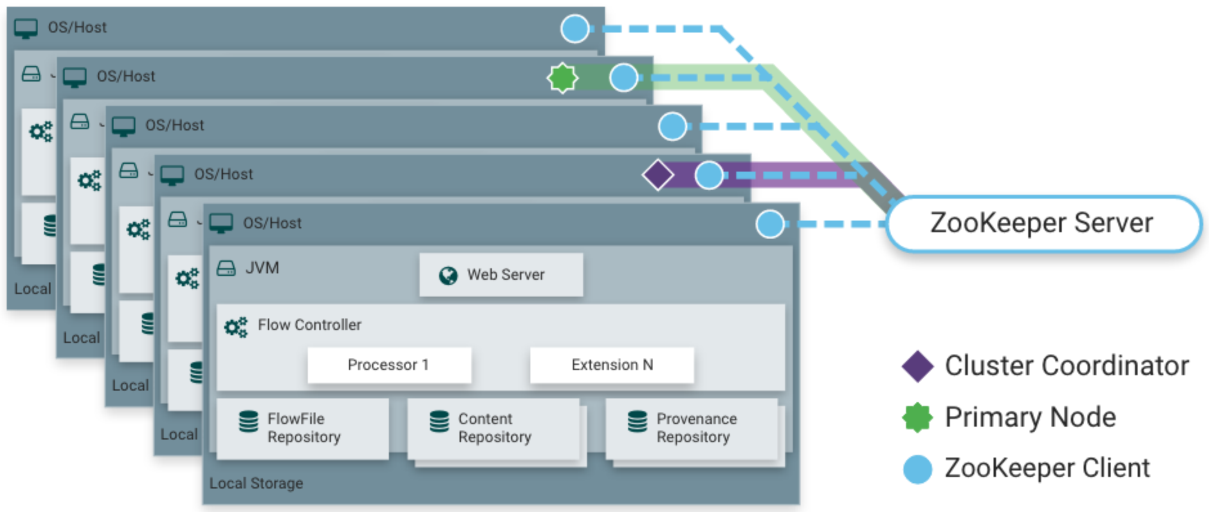
Content Repository

The Content Repository is where the actual content bytes of a given FlowFile live. The implementation of the repository is pluggable. The default approach is a fairly simple mechanism, which stores blocks of data in the file system. More than one file system storage location can be specified so as to get different physical partitions engaged to reduce contention on any single volume.

Provenance Repository

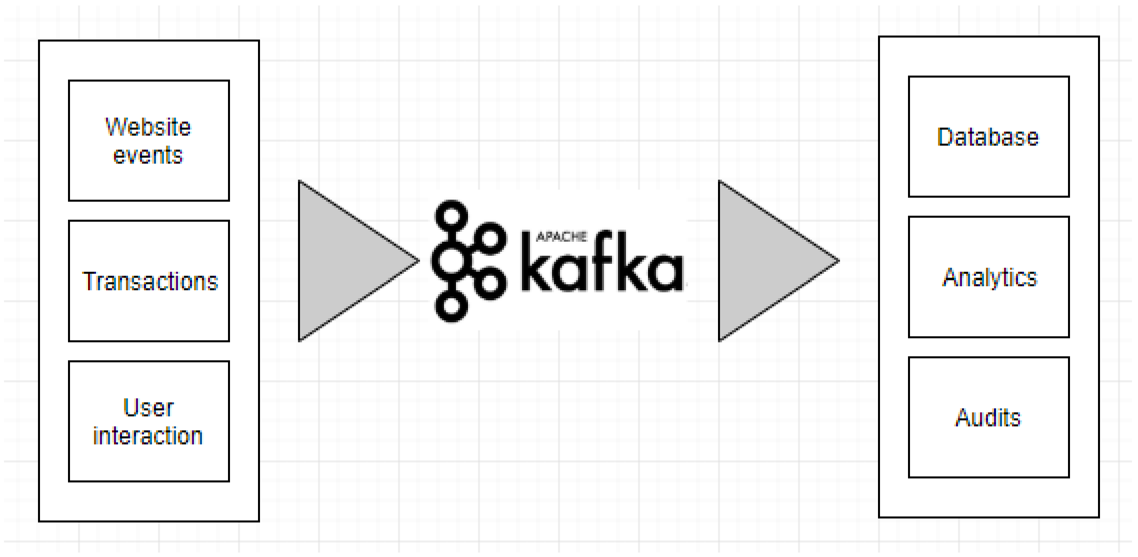
The Provenance Repository is where all provenance event data is stored. The repository construct is pluggable with the default implementation being to use one or more physical disk volumes. Within each location event data is indexed and searchable.

NiFi is also able to operate within a cluster.



Apache Kafka:

Apache Kafka allows you to decouple the data streams and system. It is a bridge between your source system and target system.



It is created by LinkedIn, Now Open source project maintained by confluent. It is powerful distributed data streaming platform for large scale enterprise application.

Characteristics of Kafka

Apache Kafka is popular because of the following major characteristics

- Distributed
- Resilient
- Fault tolerant
- Horizontal scalable
- High performance / Real time

Apache Kafka Use cases

Apache Kafka used for

- De-coupling of system dependencies
- Messaging / Activity Tracking
- Log processing
- Integration with Big Data technologies
- Stream processing

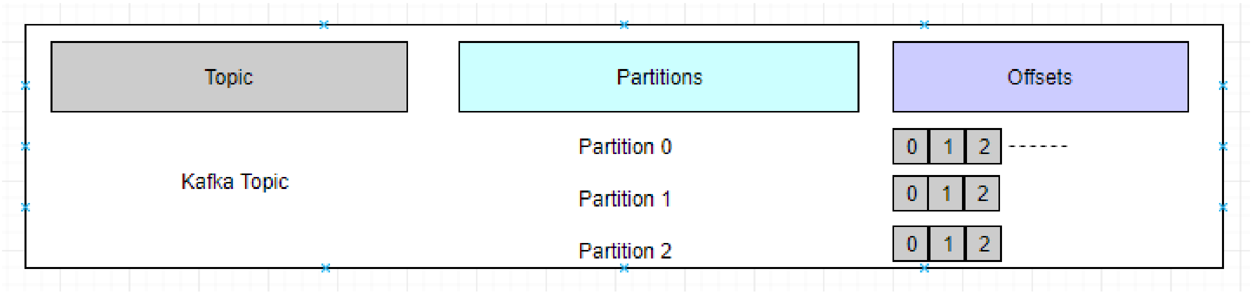
Understanding Apache Kafka Topics Partitions and Brokers:

Topics:

Topic in Kafka is heart of everything. It is stream of data / location of data in Kafka. We can create many topics in Apache Kafka, and it is identified by unique name.

Partitions:

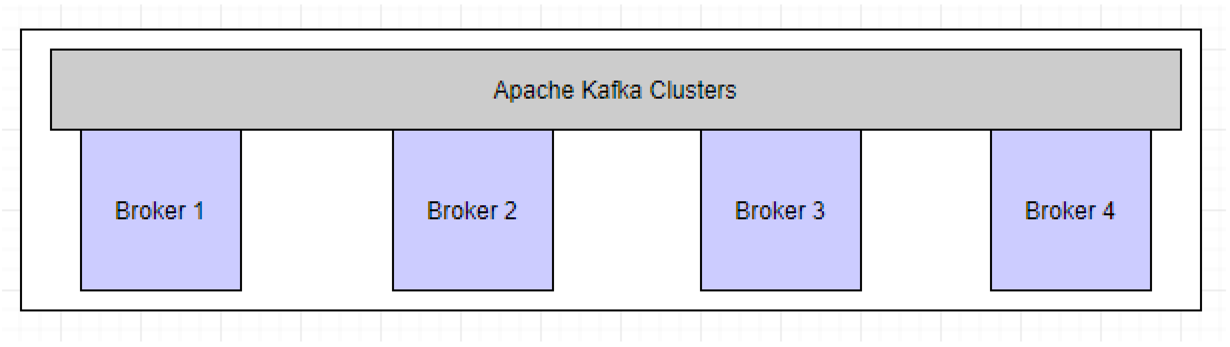
Topics are split into partitions, each partition is ordered and messages with in a partitions gets an id called Offset and it is incremental unique id.



Offset is specific to Partition, Offset 0 of Partition 0 is completely different from Offset 0 of Partition 1. No of partition is important when we create the topic and that need to be specified at the time of create topic CLI command. Data is stored in Partition for a limited time and it is immutable and it can’t be changed.

Broker:

Broker is a server / node in Apache Kafka. That means Apache Kafka cluster is composed of multiple brokers.



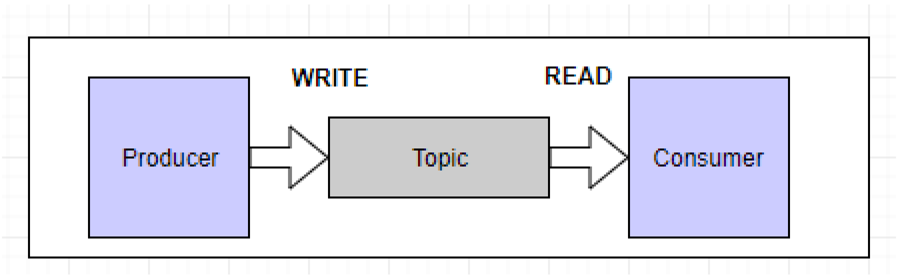
Each Broker in Cluster identified by unique ID (Integer). Each Broker contains certain partitions of a topic. When we specify number of partition at the time of Topic creation data is spread to Brokers available in the clusters. For example, Topic 1 with 2 partitions utilize Broker 1 and Broker 2 if cluster contains more than 2 brokers / node. If it is only one Broker, both partitions are stored in same Broker. That means broker count is less than the partition count, multiple partition of same topic is available in any one of the broker.

Topic Replication

Apache Kafka is distributed system. Replication (Copy) is important for Distributed system / Big data world. Topic replication factor indicates how many copy we need to maintain in Broker for topic messages / Partition. One of the required parameter we need to specify at the time of topic creation CLI command.

Producers & Consumers:

Producers write data to topics. And Consumers read the data from Topic.



Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments". Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

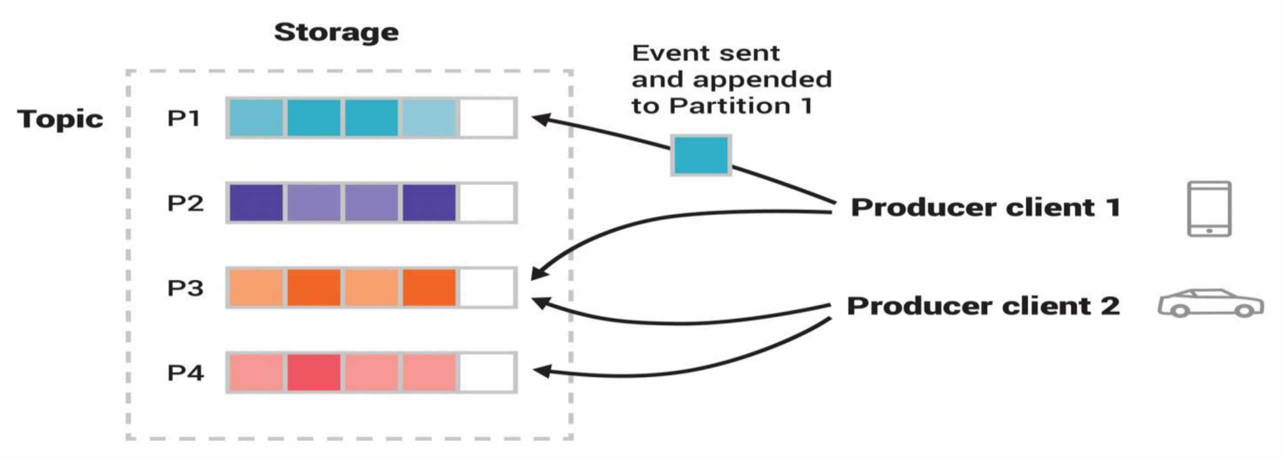


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

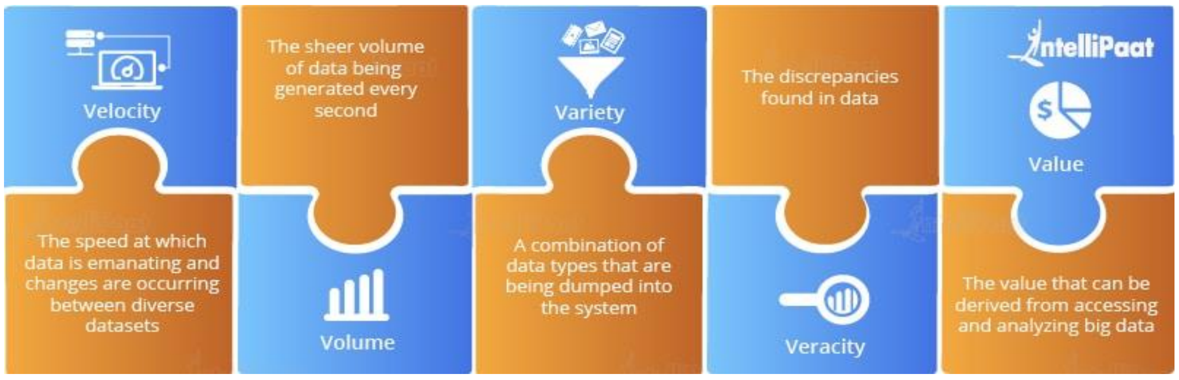
To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

Apache Spark:

Apache Spark is a lightning-fast cluster computing designed for fast computation. It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations which includes Interactive Queries and Stream Processing.

Five Vs of Big Data

Data can be categorized as big data based on various factors. The main concept common in all these factors is the amount of data. Let us understand the characteristics of big data which we have broken down into 5 Vs:


Velocity
The speed at which data is emanating and changes are occurring between diverse datasets
The sheer volume of data being generated every second
Volume
Variety
A combination of data types that are being dumped into the system
The discrepancies found in data
Veracity
IntelliPaat
Value
The value that can be derived from accessing and analyzing big data

1. Velocity

Velocity refers to the speed at which data arrives. Every day, huge amounts of data are generated, stored, and analyzed. This includes emails, images, financial reports, videos, etc. Data is being generated at lightning speed around the world. Big Data Analytics tools allow us to explore the data, at the very time it gets generated.

2. Volume

Volume refers to the huge amount of data, generated from credit cards, social media, IoT devices, smart home gadgets, videos, etc. Data is growing so large that traditional computing systems can no longer handle it the way we want.

3. Variety

Variety refers to the different types of data. Data is mainly categorized into structured and unstructured data. Structured data has a schema and well-defined tables to store information. Data without a schema and a pre-defined data model is called the unstructured data. In fact, more than 75 percent of the world's data exists in the unstructured form. The unstructured data includes images, videos, social media-generated data, etc.

4. Veracity

Veracity refers to the quality of the data. Let's suppose that we are storing some data using high computational power. If this data is of no use in the future, then we are wasting our resources on it. Thus, we have to check the trustworthiness of the data before storing it. It depends on the reliability and accuracy of the content. We should not store loads of data if the content is not reliable or accurate.

5. Value

Value is the most important part of big data. Organizations use big data to find hidden values from it. This data analysis can help increase financial benefits. Having a vast amount of data is useless until we extract something meaningful from it.

Although Hadoop made a grasp on the market, there were some limitations. Hadoop is used to process data in various batches, therefore real-time data streaming is not possible with Hadoop.

Why choose Apache Spark over Hadoop?

Both Hadoop and Spark are open-source projects from Apache Software Foundation, and they are the flagship products used for Big Data Analytics. The key difference between MapReduce and Spark is their approach toward data processing. Spark can perform in-memory processing, while Hadoop MapReduce has to read from/write to a disk. Let us understand some major differences between Apache Spark and Hadoop in the next section of this Apache Spark tutorial.

Differences Between Hadoop and Spark

1. Speed

Spark is a cluster computing platform for general use. It runs programs in memory up to 100 times faster and on disk 10 times faster than Hadoop. This is feasible for Spark, since it reduces the number of disk read/write cycles and preserves data in memory.

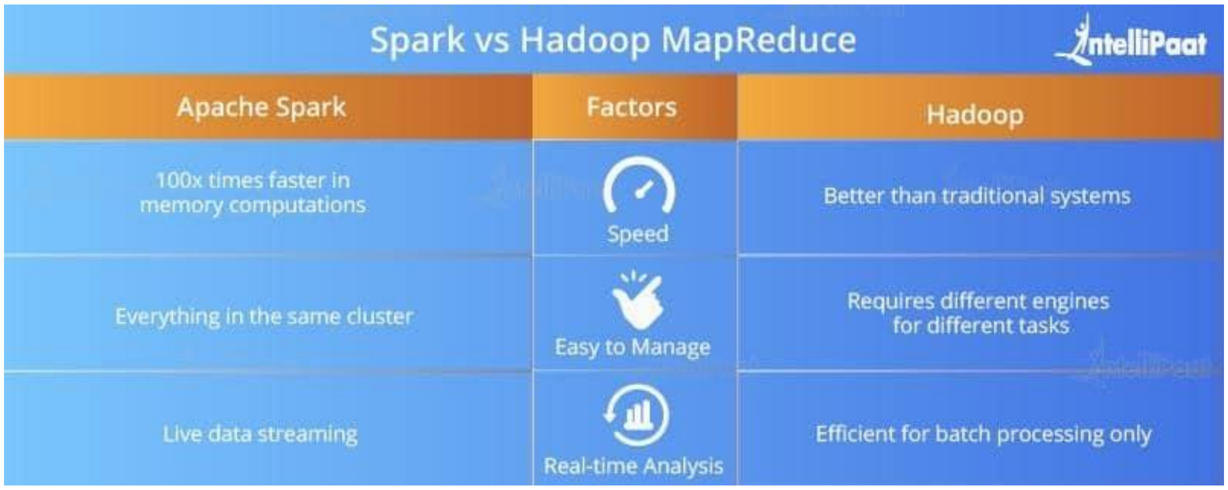
2. Easy to Manage

All in the same cluster, Spark will handle batch processing, immersive data analytics, deep learning, and streaming. This feature makes Apache Spark a full engine of Data Analytics. With Spark, for each mission, there is no need to handle different Spark components.

Just the batch-processing engine offers Hadoop MapReduce. So, in Hadoop, for each mission, we need a different engine. Many components are very difficult to treat..

3. Real-time Analysis

Spark can handle real-time data quickly, i.e. real-time event streaming at a rate of millions of events per second, e.g. Spark processes data streaming live from Twitter, Facebook, Instagram, etc. effectively. Here, since it does not accommodate real-time data analysis, MapReduce fails. It is intended to do only batch processing on immense data volumes.



Spark vs Hadoop MapReduce

IntelliPaat

Apache Spark	Factors	Hadoop
100x times faster in memory computations	Speed	Better than traditional systems
Everything in the same cluster	Easy to Manage	Requires different engines for different tasks
Live data streaming	Real-time Analysis	Efficient for batch processing only

The main variations between Apache Spark and Hadoop are these. But, what if, with Hadoop, we use Apache Spark? It provides more efficient cluster computing with batch processing and real-time processing as we use all technologies together.

Two Main Abstractions of Apache Spark

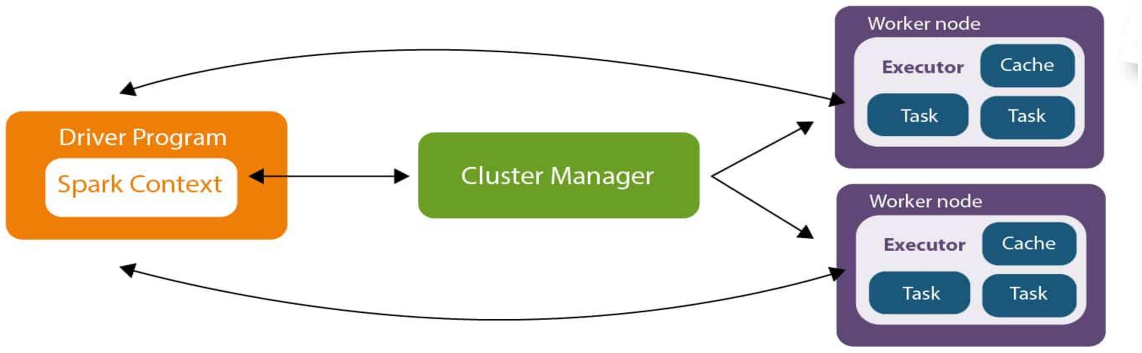
Apache Spark has a well-defined layer architecture which is designed on two main abstractions:

- Resilient Distributed Dataset (RDD): RDD is an immutable (read-only) simple set of elements or goods that can be operated at the same time on several computers (parallel processing). It is possible to split each dataset in an RDD into logical parts, which are then executed on separate cluster nodes.

- Directed Acyclic Graph (DAG): DAG is the Apache Spark architecture's scheduling layer that implements stage-oriented scheduling. Apache Spark will generate DAGs that include several phases, relative to MapReduce, which generates a graph in two phases, Map and Reduce.

Working of the Apache Spark Architecture:

The basic Apache Spark architecture is shown in the figure below:



In the Apache Spark architecture, the driver program calls an application's main program and emits SparkContext. A SparkContext consists of all the fundamental functions. Spark Driver comprises numerous other items, such as DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager, which are responsible for converting user-written code into cluster jobs. Spark Driver and SparkContext collectively watch over the job execution within the cluster. Spark Driver works with the Cluster Manager to manage various other jobs. Cluster Manager does the resource allocating work. And then, the job is split into multiple smaller tasks which are further distributed to worker nodes.

It can be spread over several worker nodes whenever an RDD is created in the SparkContext and can also be cached there. Worker nodes perform and return the tasks delegated by the Cluster Manager to the Spark Context.

For the execution of these functions, an executor is responsible. The executors' lifetime is the same as that of the Spark Program. If we wish to maximize the system's efficiency, we should increase the number of employees so that it is possible to split the positions into more rational parts.