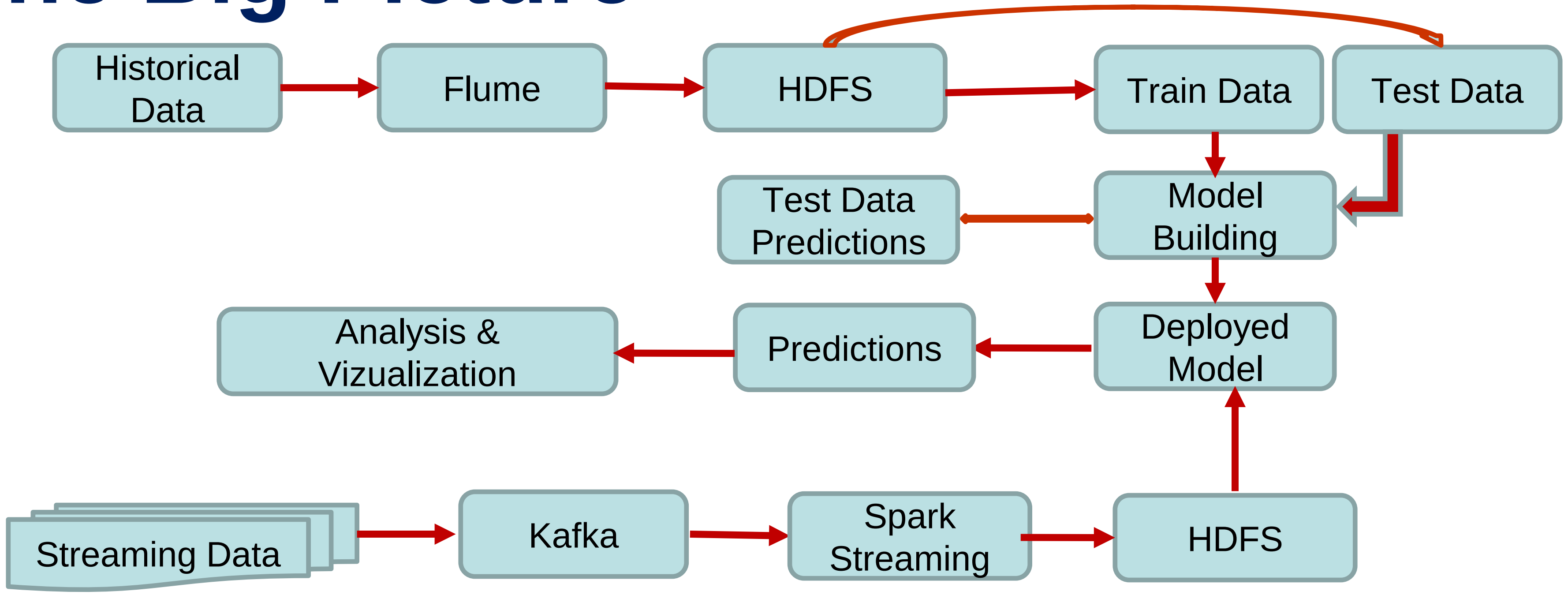
Inspire…Educate…Transform.

Kafka Activity & Spark Streaming

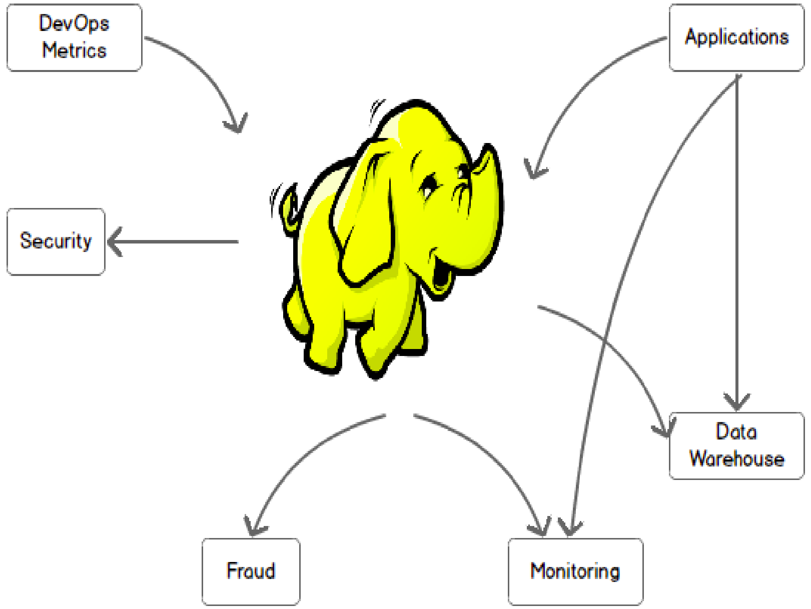
Ref: https://kafka.apache.org/intro

The Big Picture



Why Kafka

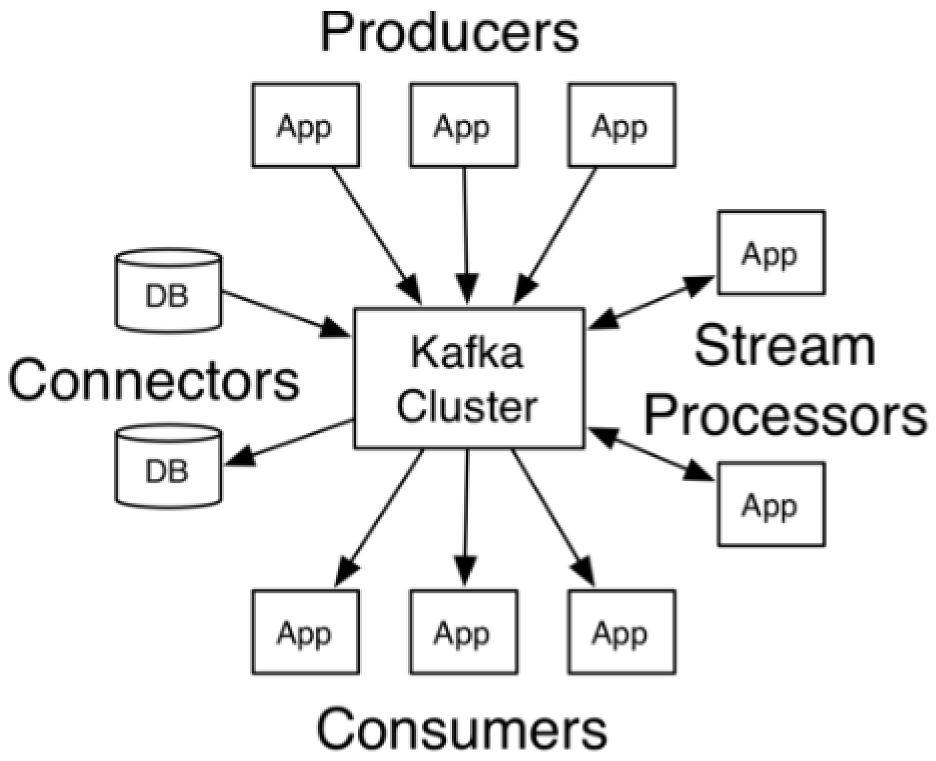
- In the real world data exists on many systems in parallel, all of which need to interact with Hadoop and with each other.
- The situation quickly becomes more complex, ending with a system where multiple data systems are talking to one another over many channels.



Contd..

- Each of these channels requires separate way and communication methods and moving data between these systems becomes a full-time job for a team of developers.
- Solution for this is a single component to act like a broker which will serves the purpose of taking the data from different sources and sending them to the different sources.

Think of it


Producers
App
App
App
DB
Connectors
DB
Kafka Cluster
App
Stream Processors
App
App
App
App
Consumers

Kafka

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Contd..

- It 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

Contd..

- Kafka is built on top of the ZooKeeper synchronization service.
- It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Applications

- Kafka is generally used for two broad classes of applications:
  - Building real-time streaming data pipelines that reliably get data between systems or applications
  - Building real-time streaming applications that transform or react to the streams of data

Contd..

- Kafka is run as a cluster on one or more servers that can span multiple datacenters.
- The Kafka cluster stores streams of *records* in categories called *topics*.
- Each record consists of a key, a value, and a timestamp.

Terminologies in Kafka

Topics

- 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 minimum of one partition.
- Each such partition contains messages in an immutable ordered sequence that is continuously append to .

Contd..

Partition

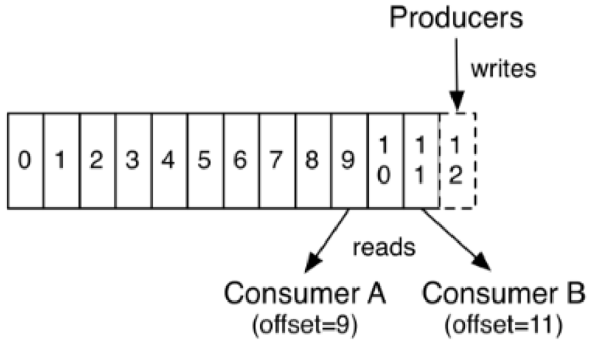
- Topics may have many partitions, so it can handle an arbitrary amount of data.
- Partition is like a Single log.
- Messages are written to it in an append-only fashion, and are read in order from beginning to end.

Partition offset

- The records in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each record within the partition.

Contd..

- Metadata retained on a per-consumer basis is the offset or position of that consumer in the log.
- This offset is controlled by the consumer
- Normally a consumer will advance its offset linearly as it reads records, but, in fact,
- Since the position is controlled by the consumer it can consume records in any order it likes.



For example: a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".

Contd..

Replicas of partition

- Each partition is replicated across a configurable number of servers for fault tolerance.
- The Kafka cluster durably persists all published records—whether or not they have been consumed—using a configurable retention period.

Leader

- Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". The leader handles all read and write requests for the partition while the followers passively replicate the leader.If the leader fails one of the partition will be selected as a leader by zookeeper.

Producers

- Producers publish data to the topics of their choice.
- The producer is responsible for choosing which record to assign to which partition within the topic.
- This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function
- 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 messages to a partition of their choice.

Contd..

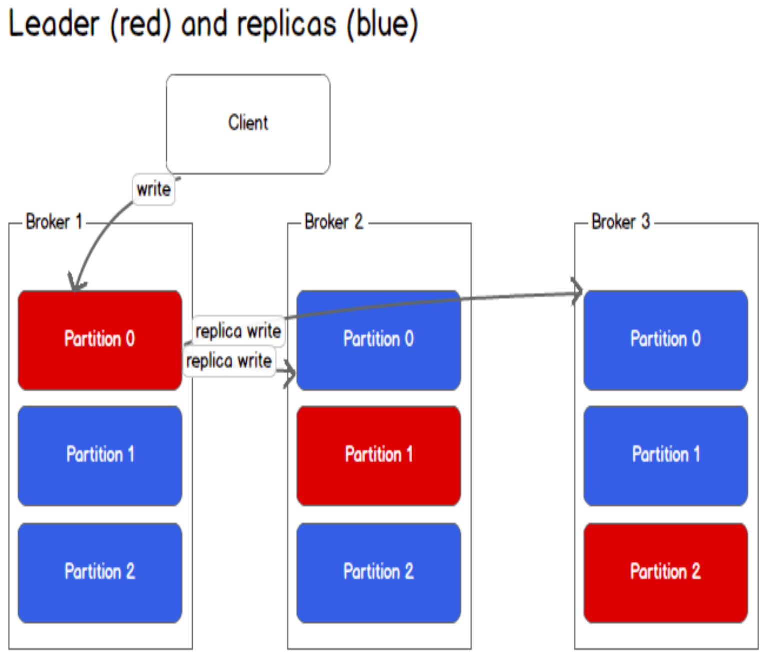
Brokers

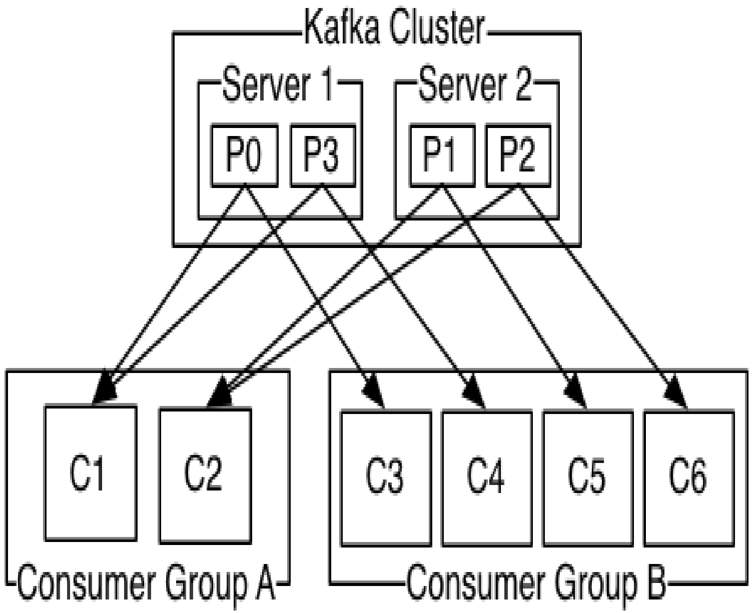
- Brokers are simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic.

Kafka Cluster

- Kafka’s having more than one broker are called as Kafka cluster. These clusters are used to manage the persistence and replication of message data.

Read and Write





Zookeeper

- For the purpose of managing and coordinating, Kafka broker uses **ZooKeeper**.
- Also, uses it 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 soon as Zookeeper send the notification regarding presence or failure of the broker then producer and consumer, take the decision and starts coordinating their task with some other broker

Zookeeper and Broker nodes

Zookeeper Nodes:

a.insofe.edu.in:2181

b.insofe.edu.in:2181

c.insofe.edu.in:2181

e.insofe.edu.in:2181

Broker ID:

c.insofe.edu.in:9092

Kafka Topic Creation

```
export PATH=$PATH:/usr/hdp/current/kafka-broker/bin
## list the topics on the zookeper
kafka-topics.sh --list --zookeeper <zookeeper-ip>
## Create a topic using the below command
## Topic Name is insofe_<batch >_<topic name>
kafka-topics.sh --create --zookeeper <zookeper-ip> --
replication-factor 1 --partitions 1 --topic <topic_name>
```

Producer

```
kafka-console-producer.sh --broker-list <broker-id> --topic <topic_name>
```

Consumer

```
kafka-console-consumer.sh --zookeeper <zookeeper-ip> --topic <topic_name> --from-beginning
```

Spark Streaming

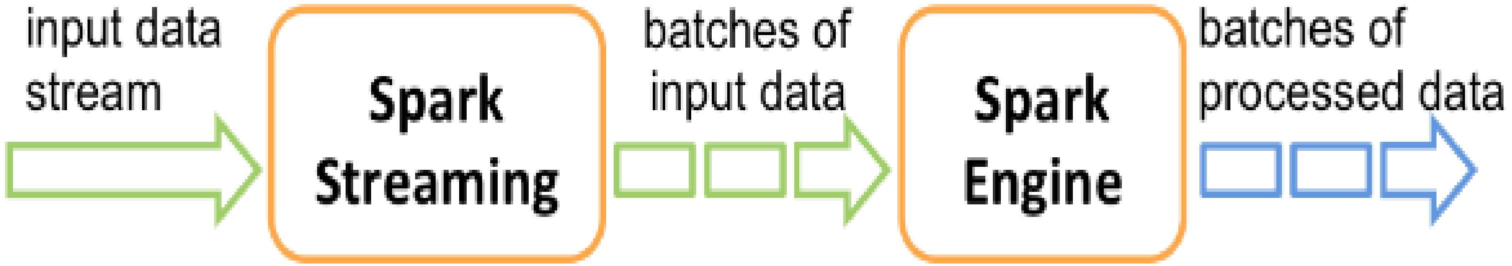
- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map,reduce, join and window.

Contd..

- Finally, processed data can be pushed out to file systems, databases, and live dashboards. In fact, you can apply Spark’s machine learning and graph processing algorithms on data streams.

Contd..

- Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



Contd..

- Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data.
- DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Initialize Streaming Context

- A **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

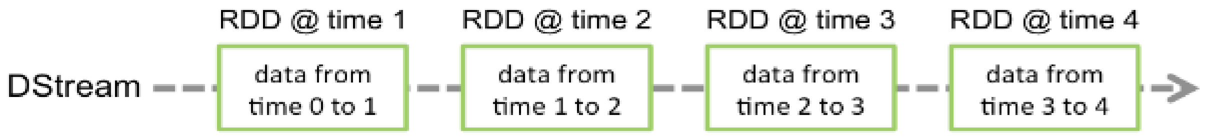
$\xrightarrow{}$ Batch Interval

After defining Streaming Context

- Define the input sources by creating input DStreams.
- Define the streaming computations by applying transformation and output operations to DStreams.
- Start receiving data and processing it using streamingContext.start().
- Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
- The processing can be manually stopped using streamingContext.stop().

DStream

- DStream is represented by a continuous series of RDDs, which is Spark’s abstraction of an immutable, distributed database.



- Any operation applied on a DStream translates to operations on the underlying RDDs.It applies on every RDD that is splitted by time.

Input DStreams and Receivers

- Every input DStream is associated with a **Receiver** (Scala doc, Java doc) object which receives the data from a source and stores it in Spark's memory for processing.
- Spark Streaming provides two categories of built-in streaming sources.
  - *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
  - *Advanced sources*: Sources like Kafka, Flume, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

Transformations

- Return a new DStream after the transformation.

Types:

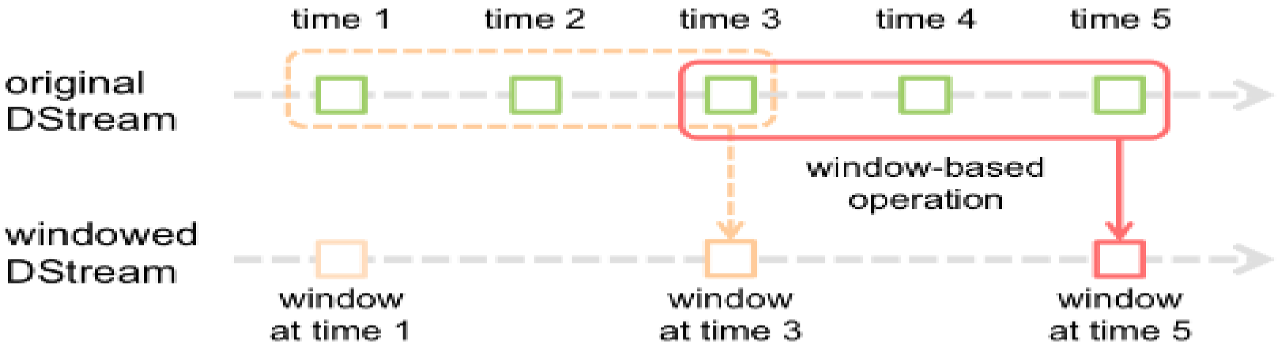
Stateless Transformation:

- Do not consider the previous state of the batch
- Applied on to each RDD that Dstreams is composed of.
- Ex: map,flatmap,reduceByKey,

Stateful Transformation:

- Results of the previous batch are used to produce current batch.
- Ex : upDateStateByKey

Window Operations



The source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

This shows that any window operation needs to specify two parameters.

- *window length* - The duration of the window
- *sliding interval* - The interval at which the window operation is performed

Output Operations

- Operations that help in pushing the data on to a external System.
- This will trigger the actual execution of the RDD

Ex: pprint,saveAsTextFile etc

Cache & Persist

- DStream will automatically persist every RDD of that DStream in memory.
- This is useful if the data in the DStream will be computed multiple times (e.g., multiple operations on the same data).
- For window-based operations like `reduceByWindow` and `reduceByKeyAndWindow` and state-based operations like `updateStateByKey`, this is implicitly true. Hence, DStreams generated by window-based operations are automatically persisted in memory, without the developer calling `persist()`.

Check Pointing

- Streaming application must operate 24/7 and hence must be resilient to failures.
  - *Metadata checkpointing*
  - *Data checkpointing*