BIG DATA ANALYTICS

UNIT-6

Spark Streaming: Stream Processing Fundamentals, Structured Streaming basics- Core Concepts, Structured Streaming in Action, Transformations on Streams, Input and Output(Kafka)

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, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to file systems, databases, and live dashboards.
- In fact, we can apply Spark's machine learning and graph processing algorithms on data streams.

- Internally, it works as follows.
- 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.

- 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, and Kinesis, or by applying high-level operations on other DStreams.
- Internally, a DStream is represented as a sequence of RDDs.

First, we import the names of the Spark Streaming classes and some implicit conversions from StreamingContext into our environment in order to add useful methods to other classes we need (like DStream). StreamingContext is the main entry point for all streaming functionality

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
```

```
// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent a starvation scenario.

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

- Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

- This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space characters into words.

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

- flatMap is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream.

- In this case, each line will be split into multiple words and the stream of words is represented as the words DStream

Stream Processing Fundamentals:

- Stream processing is a data management technique that involves ingesting a continuous data stream to quickly analyze, filter, transform or enhance the data in real time.

- Once processed, the data is passed off to an application, data store or another stream processing engine.

- Stream processing services and architectures are growing in popularity because they allow enterprises to combine data feed from various sources.

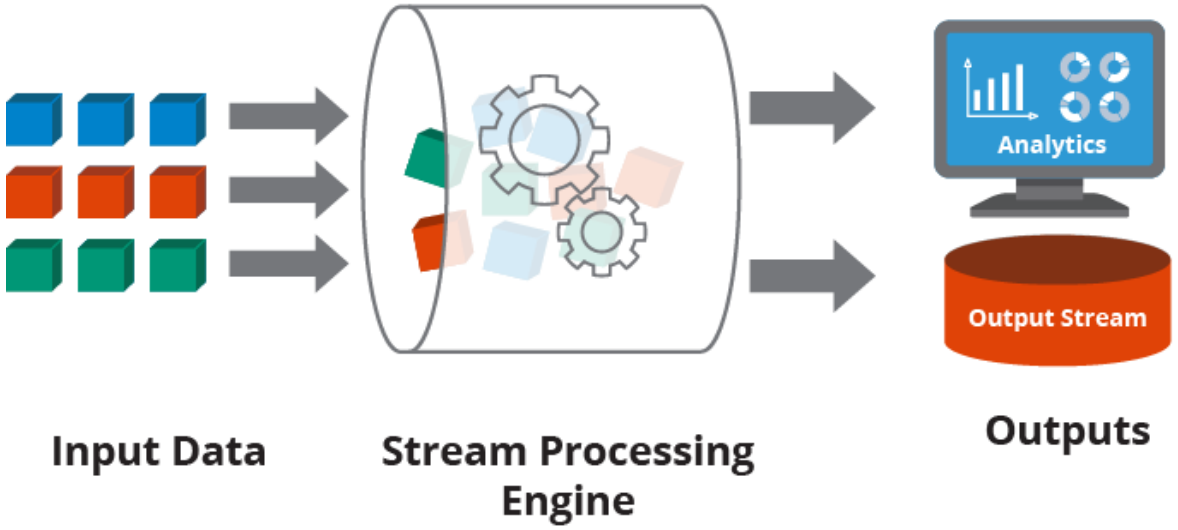
- Sources can include transactions, stock feeds, website analytics, connected devices, operational databases, weather reports and other commercial services.

- The core ideas behind stream processing have been around for decades but are getting easier to implement with various open source tools and cloud services.

How does stream processing work:

- Stream processing architectures help simplify the data management tasks required to consume, process and publish the data securely and reliably.
- Stream processing starts by ingesting data from a publish-subscribe service, performs an action on it and then publishes the results back to the publish-subscribe service or another data store.
- These actions can include processes such as analyzing, filtering, transforming, combining or cleaning data.

- Stream processing commonly connotes the notion of *real-time analytics*, which is a relative term.
- The stream processing engine organizes data events arriving in short batches and presents them to other applications as a continuous feed.
- This simplifies the logic for application developers combining and recombining data from various sources and from different time scales.



Need of stream processing :

Stream processing is needed to:

- Develop adaptive and responsive applications
- Help enterprises improve real-time business analytics
- Facilitate faster decisions
- Accelerate decision-making
- Improve decision-making with increased context
- Improve the user experience

- Create new applications that use a wider variety of data sources

Common stream processing use cases include:

- Fraud detection
- Detecting anomalous events
- Tuning business application features
- Managing location data
- Personalizing customer experience
- Stock market trading
- Analyzing and responding to IT infrastructure events
- Predictive analytics

Stream processing frameworks:

- Spark, Flink and Kafka Streams are the most common open source stream processing frameworks.

- In addition, all the primary cloud services also have native services that simplify stream processing development on their respective platforms, such as Amazon Kinesis, Azure Stream Analytics and Google Cloud Dataflow.

- These often go hand in hand with other publish-subscribe frameworks used for connecting applications and data stores.

For example,

- **Apache Kafka** is a popular open source publish-subscribe framework that simplifies integrating data across multiple applications.

- Apache Kafka Streams is a stream processing library for creating applications that ingest data from Kafka, process it and then publish the results back to Kafka as a new data source for other applications to consume.

Stream Processing Use Cases:

Notifications and alerting:

- Probably the most obvious streaming use case involves notifications and alerting. Given some series of events, a notification or alert should be triggered if some sort of event or series of events occurs.
- This doesn't necessarily imply autonomous or preprogrammed decision making; alerting can also be used to notify a human counterpart of some action that needs to be taken.
- An example might be driving an alert to an employee at a fulfillment center that they need to get a certain item from a location in the warehouse and ship it to a customer. In either case, the notification needs to happen quickly.

Incremental ETL:

- One of the most common streaming applications is to reduce the latency companies must endure while retreiving information into a data warehouse—in short, "my batch job, but streaming."
- Spark batch jobs are often used for Extract, Transform, and Load (ETL) workloads that turn raw data into a structured format like Parquet to enable efficient queries.
- Using Structured Streaming, these jobs can incorporate new data within seconds, enabling users to query it faster downstream.

- In this use case, it is critical that data is processed exactly once and in a fault-tolerant manner: we don't want to *lose* any input data before it makes it to the warehouse, and we don't want to load the same data twice.
- Moreover, the streaming system needs to make updates to the data warehouse transactionally so as not to confuse the queries running on it with partially written data

Update data to serve in real time:

- Streaming systems are frequently used to compute data that gets served interactively by another application.
- For example, a web analytics product such as Google Analytics might continuously track the number of visits to each page, and use a streaming system to keep these counts up to date.
- When users interact with the product's UI, this web application queries the latest counts. Supporting this use case requires that the streaming system can perform incremental updates to a key–value store (or other serving system) as a sync, and often also that these updates are transactional, as in the ETL case, to avoid corrupting the data in the application.

Real-time decision making:

- Real-time decision making on a streaming system involves analyzing new inputs and responding to them automatically using business logic.
- An example use case would be a bank that wants to automatically verify whether a new transaction on a customer's credit card

represents fraud based on their recent history, and deny the transaction if the charge is determined fradulent.

- This decision needs to be made in real-time while processing each transaction, so developers could implement this business logic in a streaming system and run it against the stream of transactions.
- This type of application will likely need to maintain a significant amount of *state* about each user to track their current spending patterns, and automatically compare this state against each new transaction.

Online machine learning:

- A close derivative of the real-time decision-making use case is online machine learning. In this scenario, you might want to train a model on a combination of streaming and historical data from multiple users.
- An example might be more sophisticated than the aforementioned credit card transaction use case: rather than reacting with hardcoded rules based on *one* customer's behavior, the company may want to continuously update a model from *all* customers' behavior and test each transaction against it.
- This is the most challenging use case of the bunch for stream processing systems because it requires aggregation across multiple customers, joins against static datasets, integration with machine learning libraries, and low-latency response times.

Structured Streaming Basics:

- Structured Streaming, is a stream processing framework built on the Spark SQL engine.

- Rather than introducing a separate API, Structured Streaming uses the existing structured APIs in Spark (DataFrames, Datasets, and SQL), meaning that all the operations you are familiar with there are supported.
- Structured Streaming engine will take care of running your query incrementally and continuously as new data arrives into the system.
- These logical instructions for the computation are then executed using the same Catalyst engine including query optimization, code generation, etc.
- Structured Streaming includes a number of features specifically for streaming. For instance, Structured Streaming ensures end-to-end, exactly-once processing as well as fault-tolerance through checkpointing and write-ahead logs.
- The main idea behind Structured Streaming is to treat a *stream* of data as a *table* to which data is continuously appended.
- The job then periodically checks for new input data, process it, updates some internal state located in a state store if needed, and updates its result.
- Internally, Structured Streaming will automatically figure out how to "incrementalize" your query, i.e., update its result efficiently whenever new data arrives, and will run it in a fault tolerant fashion.

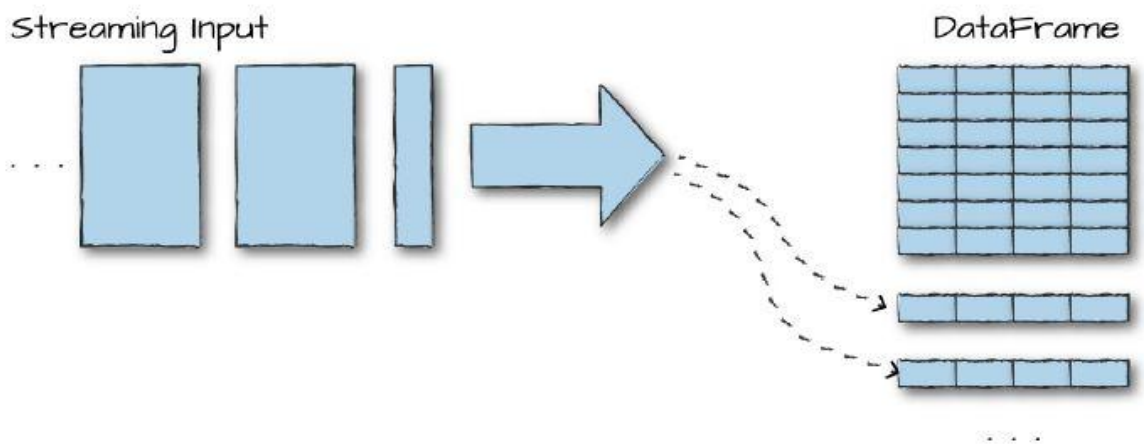


Figure 21-1. Structured streaming input

Core Concepts:

Transformations and Actions

- Structured Streaming maintains the same concept of transformations and actions that we have seen .
- The transformations available in Structured Streaming are, with a few restrictions, the exact same transformations .
- The restrictions usually involve some types of queries that the engine cannot incrementalize yet, although some of the limitations are being lifted in new versions of Spark.
- There is generally only one action available in Structured Streaming: that of starting a stream, which will then run continuously and output
  results.

Input Sources:

- Structured Streaming supports several **input sources** for reading in a streaming fashion. As of Spark 2.2, the supported input sources are as follows:
- Apache Kafka 0.10

- Files on a distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)

Sinks:

- Just as sources allow you to get data into Structured Streaming, *sinks* specify the destination for the result set of that stream. Sinks and the execution engine are also responsible for reliably tracking the exact progress of data processing.
- Here are the supported output sinks as of Spark 2.2:
- Apache Kafka 0.10
- Almost any file format
- A foreach sink for running arbitary computation on the output records
- A console sink for testing
- A memory sink for debugging

Output Modes:

- We define an *output mode*, similar to how we define output modes in the static Structured APIs.
- The supported output modes are as follows:
- Append (only add new records to the output sink)
- Update (update changed records in place)
- Complete (rewrite the full output)

Triggers:

- Triggers define *when* data is output—that is, when Structured Streaming should check for new input data and update its result.

- By default,Structured Streaming will look for new input records as soon as it has finished processing the last group of input data, giving the lowest latency possible for new results. However, this behavior can lead to writing many small output files when the sink is a set of files.
- Thus, Spark also supports triggers based on processing time (only look for new data at a fixed interval).

Event-Time Processing:

- Structured Streaming also has support for event-time processing (i.e., processing data based on timestamps included in the record that may arrive out of order).

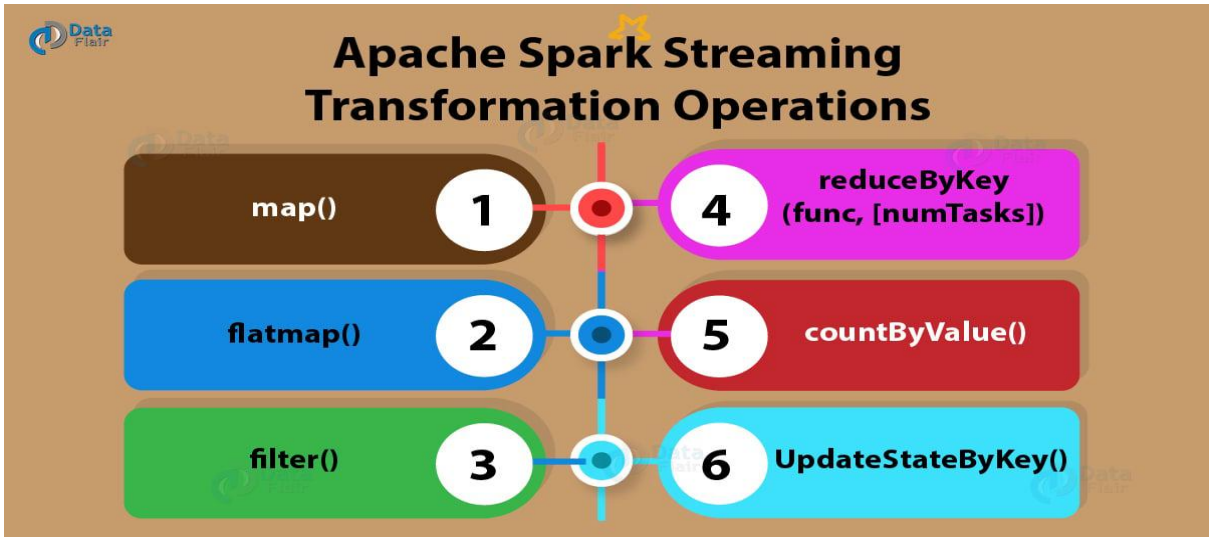
Event-time data:

- *Event-time* means time fields that are embedded in your data.
- This means that rather than processing data according to the time it reaches your system, you process it according to the time that it was generated, even if records arrive out of order at the streaming application due to slow uploads or network delays.
- Expressing event-time processing is simple in Structured Streaming.

Transformations on Streams:

- Through this Apache Spark Transformation Operations we will learn about various Apache Spark streaming transformation operations with example being used.
- Streaming operations like Spark Map operation, flatmap operation, Spark filter operation, count operation, Spark ReduceByKey operation, Spark CountByValue operation

with example and Spark UpdateStateByKey operation with example that will help in the Spark jobs.



a. map()

- Map function in Spark passes each element of the source DStream through a function and returns a new DStream.

- **Spark map() example**:
  [php]val conf = new SparkConf().setMaster("local[2]")
  .setAppName("MapOpTest")
  val ssc = new StreamingContext(conf , Seconds(1))val words = ssc.socketTextStream("localhost", 9999)
  val ans = words.map { word => ("hello" ,word) } // map hello with each line
  ans.print()
  ssc.start() // Start the computation
  ssc.awaitTermination() // Wait for termination
  }[/php]

b. flatMap()

- FlatMap function in Spark is similar to Spark map function, but in flatmap, input item can be mapped to 0 or more output items.
- This creates difference between map and flatmap operations in spark.

- **Spark FlatMap Example**
  [php]val lines = ssc.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_.split(" ")) // for each line it split the words by space
  val pairs = words.map(word => (word, 1))
  val wordCounts = pairs.reduceByKey(_ + _)
  wordCounts.print()[/php]

c. filter():

- Filter function in Apache Spark returns selects only those records of the source DStream on which func returns true and returns a new DStream of those records.

- **Spark Filter function example**
  [php]val lines = ssc.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_.split(" "))
  val output = words.filter { word => word.startsWith("s") } // filter the words starts with letter"s"
  output.print()[/php]

d. reduceByKey(func, [numTasks])

- When called on a DStream of (K, V) pairs, ReduceByKey function in Spark returns a new DStream of (K, V) pairs where the values

for each key are aggregated using the given reduce function.

- **Spark reduceByKey example**
  [php]val lines = ssc.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_.split(" "))
  val pairs = words.map(word => (word, 1))
  val wordCounts = pairs.reduceByKey(_ + _)
  wordCounts.print()[/php]

e. countByValue()

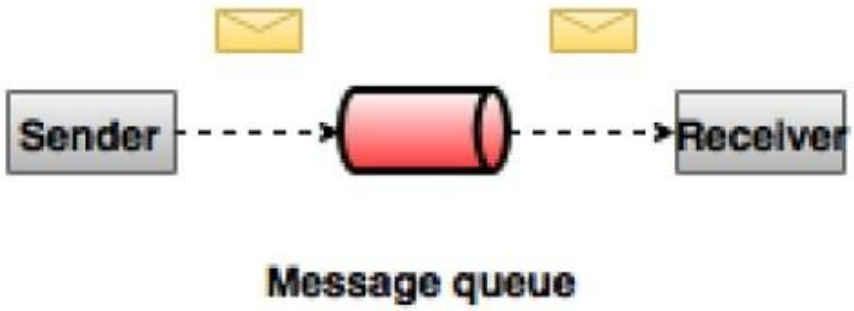
- CountByValue function in Spark is called on a DStream of elements of type K and it returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each Spark RDD of the source DStream.

- **Spark CountByValue function example**
  [php]val line = ssc.socketTextStream("localhost", 9999)
  val words = line.flatMap(_.split(" "))
  words.countByValue().print()[/php]

Kafka:

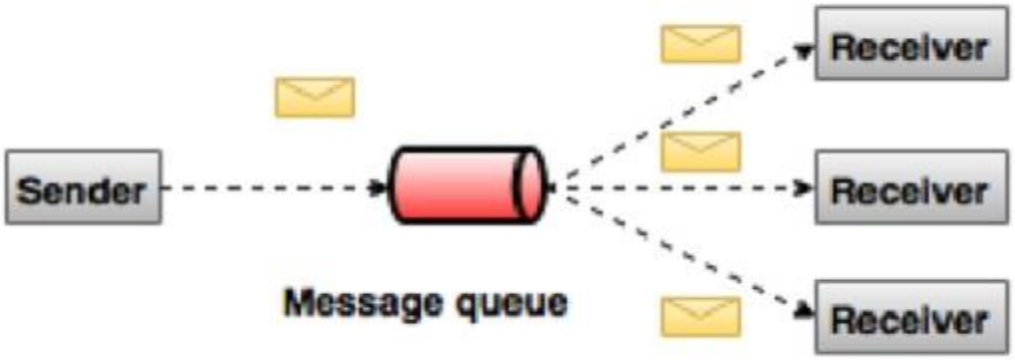
- Apache Kafka is a publish-subscribe messaging system.
- A messaging system let you send messages between processes, applications, and servers.
- Broadly Speaking, Apache Kafka is a software where topics can be defined and further processed.
- Applications may connect to this system and transfer a message onto the topic.

- A message can include any kind of information ,from any event on your Personal blog or can be a very simple text message that would trigger any other event.
- 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 with Apache Storm and Spark for real-time streaming data analysis

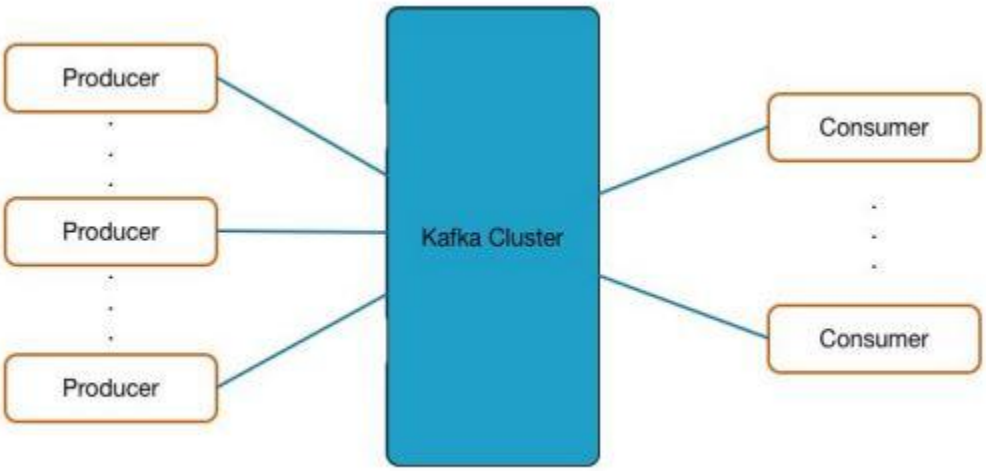


Publish-Subscribe Messaging System

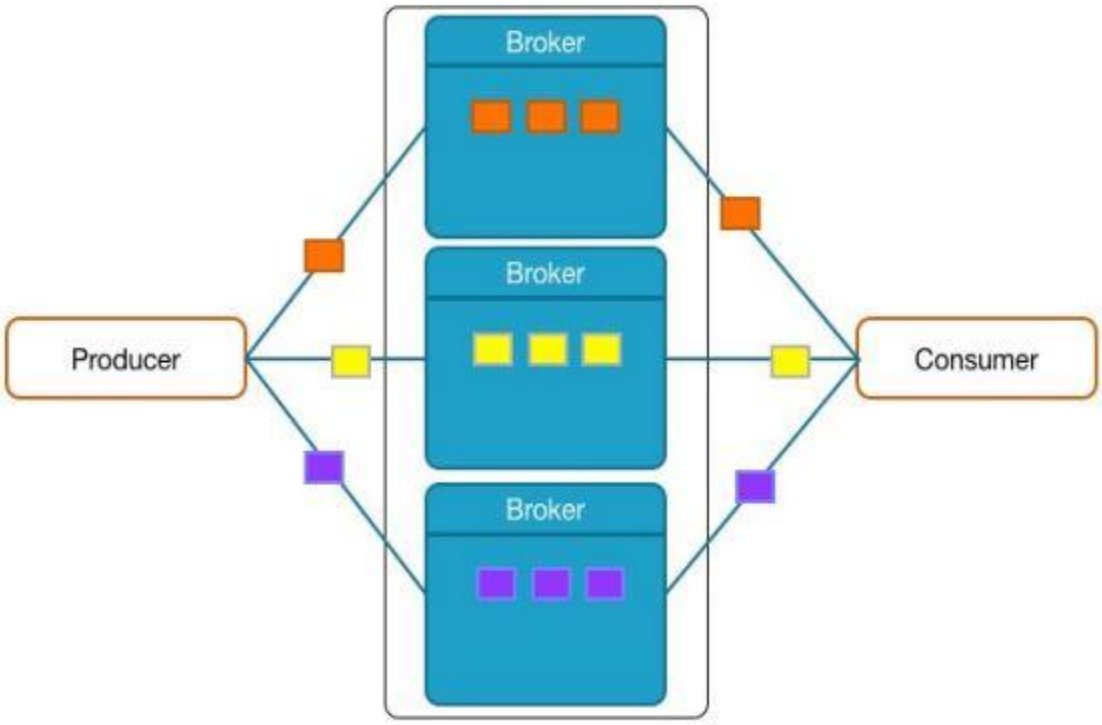
- In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers.
- A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



- A Kafka cluster usually consists of one or more servers (called as kafka brokers), which are running Kafka over them. **Producers** are processes that publish data (push messages over trigger) into Kafka topics within the specified broker.

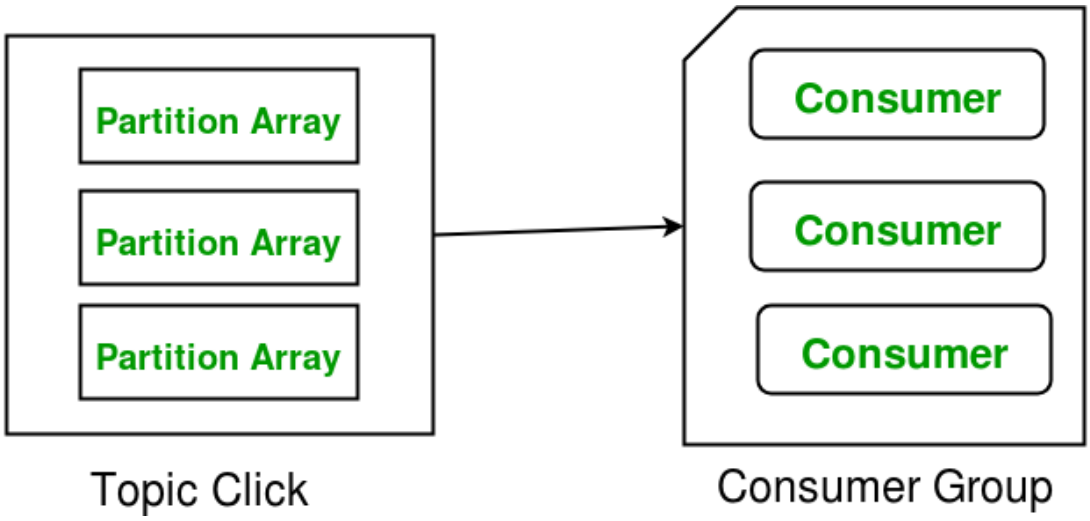


- The basic unit of data in **Kafka is a message**
- Message is sometimes used interchangeably with record
- Producers write messages to Brokers
- Consumers read messages from Brokers



Consumers and consumer groups : Consumers can always read messages starting from a specific offset and are allowed to read from any offset point they choose in between. This allows consumers to join the cluster at any point in time.This makes functioning and working really smooth.

Partitions allow you to parallelise a topic by splitting the data in a particular topic across multiple brokers.



Benefits:

Following are a few benefits of Kafka −

- **Reliability** − Kafka is distributed, partitioned, replicated and fault tolerance.
- **Scalability** − Kafka messaging system scales 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.

Real time Applications:

- **Twitter**: Registered users can read and post tweets, but unregistered users can only read tweets. Twitter uses Storm-Kafka as a part of their stream processing infrastructure.

- **LinkedIn**: Apache Kafka is used at LinkedIn for activity stream data and operational metrics. Kafka messaging system helps LinkedIn with various products like LinkedIn Newsfeed, LinkedIn Today for online message consumption and in addition to offline analytics systems like Hadoop.
- **Netflix**: Netflix is an American multinational provider of on-demand Internet streaming media. Netflix uses Kafka for real-time monitoring and event processing.
- **Box**: At Box, Kafka is used for the production analytics pipeline & real time monitoring infrastructure.

For your Reference:

https://spark.apache.org/docs/latest/streaming-programming-guide.html