ADVANCE BIG DATA

UNIT 1 UNDERSTANDING SPARK

INTRODUCTION

- Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective.
- Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.
- Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.
- As against a common belief, Spark is not a modified version of Hadoop and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.
- Spark uses Hadoop in two ways – one is storage and second is processing. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

APACHE SPARK

- Apache Spark is a lightning-fast cluster computing technology, designed for fast computation.
- It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing.
- The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.
- Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.
- Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

EVOLUTION OF APACHE SPARK

- Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia.
- It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

FEATURES OF APACHE SPARK

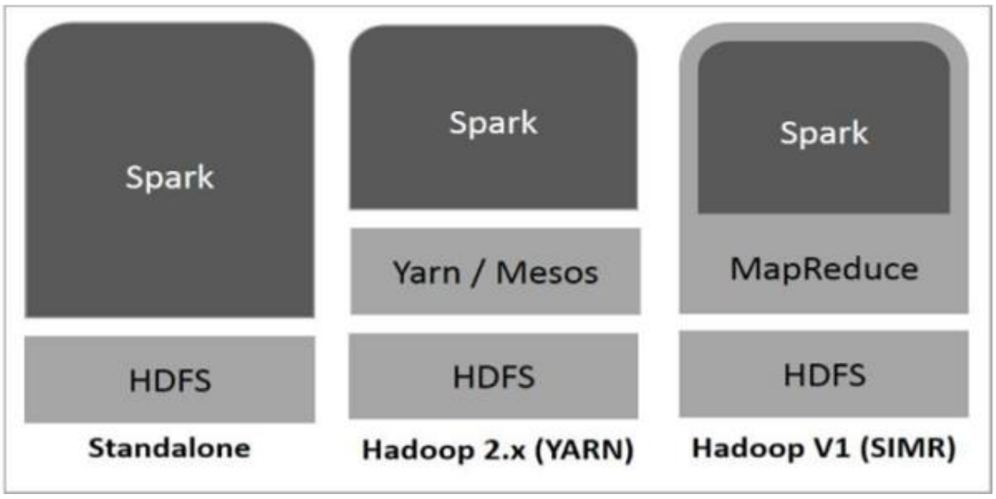
Apache Spark has following features.

1. **Speed** − Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.
2. **Supports multiple languages** − Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.

3. **Advanced Analytics** − Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

SPARK BUILT ON HADOOP

The following diagram shows three ways of how Spark can be built with Hadoop components.

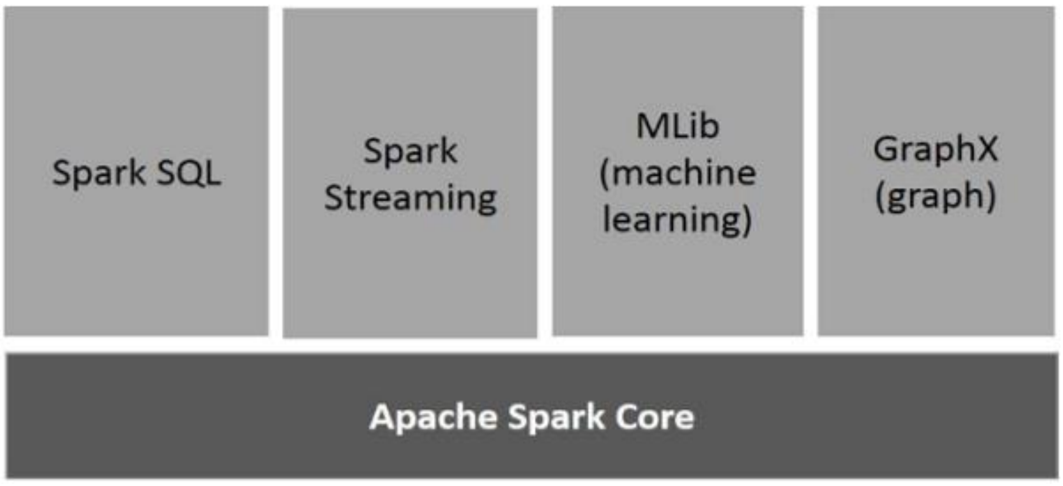


There are three ways of Spark deployment as explained below.

1. **Standalone** − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
2. **Hadoop Yarn** − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
3. **Spark in MapReduce (SIMR)** − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

COMPONENTS OF SPARK

The following illustration depicts the different components of Spark.



1. **Apache Spark Core -** Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.
2. **Spark SQL -** Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.
3. **Spark Streaming -** Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.
4. **MLlib (Machine Learning Library) -** MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of Apache Mahout (before Mahout gained a Spark interface).
5. **GraphX -** GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

RESILIENT DISTRIBUTED DATASETS (RDD)

- 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. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Formally, an RDD is a read-only, partitioned collection of records.
- RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- There are two ways to create RDDs − **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.
- Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

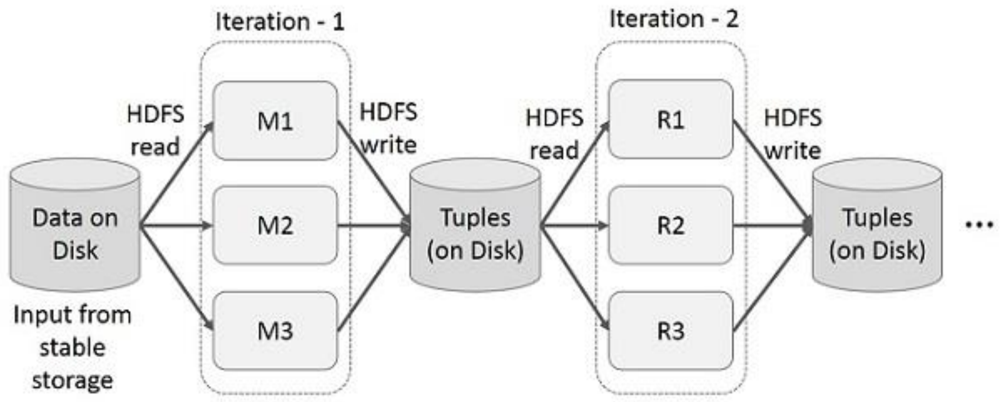
DATA SHARING IS SLOW IN MAPREDUCE

- MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster.
- It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.
- Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system (Ex − HDFS).

- Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.
- Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**.
- Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

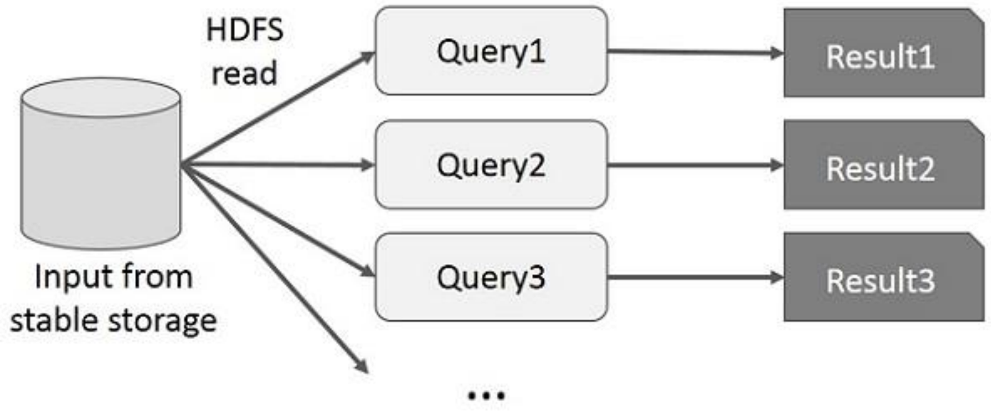
ITERATIVE OPERATIONS ON MAPREDUCE

- Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce.
- This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



INTERACTIVE OPERATIONS ON MAPREDUCE

- User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.
- The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



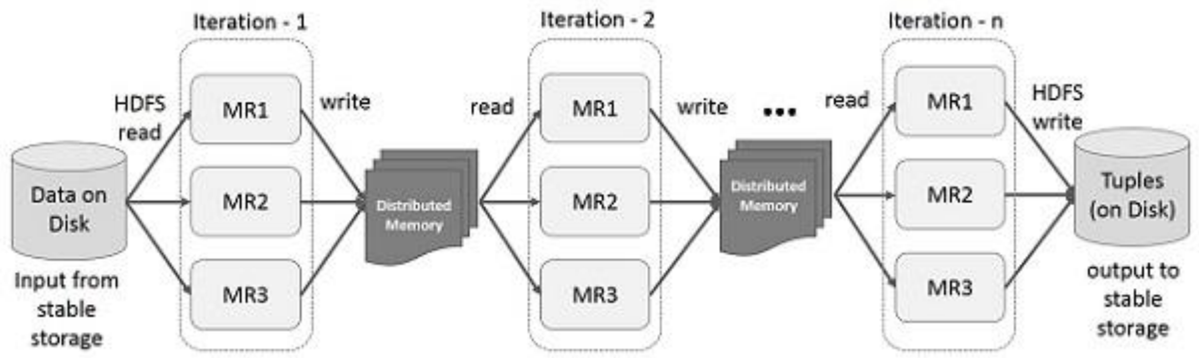
DATA SHARING USING SPARK RDD

- Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.
- Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.
- Let us now try to find out how iterative and interactive operations take place in Spark RDD.

1. Iterative Operations on Spark RDD

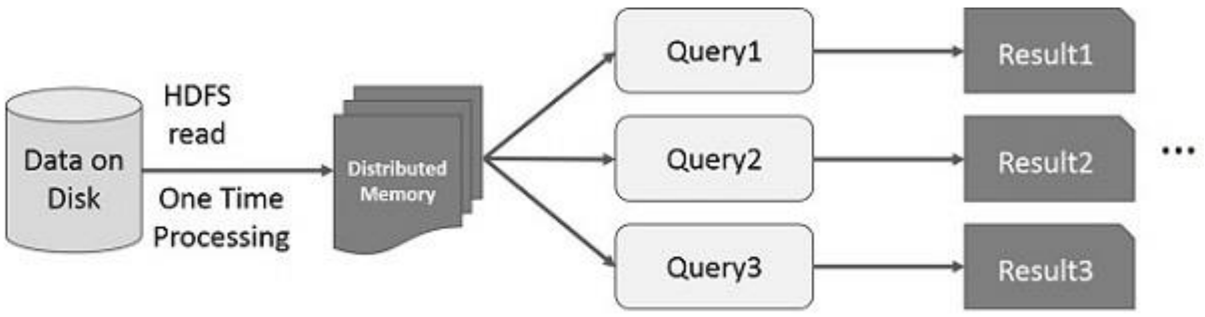
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



2. Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

SPARK PROGRAMMING

- Spark Core is the base of the whole project. It provides distributed task dispatching, scheduling, and basic I/O functionalities.
- Spark uses a specialized fundamental data structure known as RDD (Resilient Distributed Datasets) that is a logical collection of data partitioned across machines.
- RDDs can be created in two ways; one is by referencing datasets in external storage systems and second is by applying transformations (e.g. map, filter, reducer, join) on existing RDDs.
- The RDD abstraction is exposed through a language-integrated API. This simplifies programming complexity because the way applications manipulate RDDs is similar to manipulating local collections of data.

SPARK SHELL

- Spark provides an interactive shell − a powerful tool to analyze data interactively. It is available in either Scala or Python language.
- Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD).
- RDDs can be created from Hadoop Input Formats (such as HDFS files) or by transforming other RDDs.

Open Spark Shell

The following command is used to open Spark shell.

```
$ spark-shell
```

Create simple RDD

Let us create a simple RDD from the text file. Use the following command to create a simple RDD.

```
scala> val inputfile = sc.textFile(“input.txt”)
```

The output for the above command is

```
inputfile: org.apache.spark.rdd.RDD[String] = input.txt MappedRDD[1] at textFile at <console>:12
```

The Spark RDD API introduces few **Transformations** and few **Actions** to manipulate RDD.

RDD TRANSFORMATIONS

- RDD transformations returns pointer to new RDD and allows you to create dependencies between RDDs. Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD.
- Spark is lazy, so nothing will be executed unless you call some transformation or action that will trigger job creation and execution.
- Therefore, RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

Given below is a list of RDD transformations.

S.No	Transformations & Meaning
1	**map(func)** Returns a new distributed dataset, formed by passing each element of the source through a function **func**.
2	**filter(func)** Returns a new dataset formed by selecting those elements of the source on which **func** returns true.
3	**flatMap(func)** Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
4	**mapPartitions(func)** Similar to map, but runs separately on each partition (block) of the RDD, so **func** must be of type Iterator<T> ⇒ Iterator<U> when running on an RDD of type T.
5	**mapPartitionsWithIndex(func)** Similar to map Partitions, but also provides **func** with an integer value representing the index of the partition, so **func** must be of type (Int, Iterator<T>) ⇒ Iterator<U> when running on an RDD of type T.
6	**sample(withReplacement, fraction, seed)** Sample a **fraction** of the data, with or without replacement, using a given random number generator seed.
7	**union(otherDataset)** Returns a new dataset that contains the union of the elements in the source dataset and the argument.
8	**intersection(otherDataset)** Returns a new RDD that contains the intersection of elements in the source dataset and the argument.

9	**distinct([numTasks])** Returns a new dataset that contains the distinct elements of the source dataset.
10	**groupByKey([numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note** − If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
11	**reduceByKey(func, [numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V, V) ⇒ V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
12	**aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different from the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
13	**sortByKey([ascending], [numTasks])** When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
14	**join(otherDataset, [numTasks])** When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
15	**cogroup(otherDataset, [numTasks])** When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called group With.
16	**cartesian(otherDataset)** When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

17	**pipe(command, [envVars])** Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
18	**coalesce(numPartitions)** Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
19	**repartition(numPartitions)** Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
20	**repartitionAndSortWithinPartitions(partitioner)** Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

The following table gives a list of Actions, which return values.

S.No	Action & Meaning
1	**reduce(func)** Aggregate the elements of the dataset using a function **func** (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
2	**collect()** Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
3	**count()** Returns the number of elements in the dataset.
4	**first()** Returns the first element of the dataset (similar to take (1)).

5	**take(n)** Returns an array with the first **n** elements of the dataset.
6	**takeSample (withReplacement,num, [seed])** Returns an array with a random sample of **num** elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
7	**takeOrdered(n, [ordering])** Returns the first **n** elements of the RDD using either their natural order or a custom comparator.
8	**saveAsTextFile(path)** Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.
9	**saveAsSequenceFile(path) (Java and Scala)** Writes the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
10	**saveAsObjectFile(path) (Java and Scala)** Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
11	**countByKey()** Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
12	**foreach(func)** Runs a function **func** on each element of the dataset. This is usually, done for side effects such as updating an Accumulator or interacting with external storage systems. **Note** – modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

BROADCAST VARIABLE AND ACCUMULATORS

Spark contains two different types of shared variables − one is **broadcast variables** and second is **accumulators**.

- **Broadcast variables** − used to efficiently, distribute large values.
- **Accumulators** − used to aggregate the information of particular collection.

BROADCAST VARIABLES

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- They can be used, for example, to give every node, a copy of a large input dataset, in an efficient manner.
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.
- Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage.
- The data broadcasted this way is cached in serialized form and is deserialized before running each task.
- This means that explicitly creating broadcast variables, is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. The broadcast variable is a wrapper around **v**, and its value can be accessed by calling the **value** method. The code given below shows this −

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

Output −

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

- After the broadcast variable is created, it should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once.
- In addition, the object **v** should not be modified after its broadcast, in order to ensure that all nodes get the same value of the broadcast variable.

ACCUMULATORS

- Accumulators are variables that are only "added" to through an associative operation and can therefore, be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types. If accumulators

are created with a name, they will be displayed in **Spark's UI**. This can be useful for understanding the progress of running stages (NOTE − this is not yet supported in Python).

- An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**. Tasks running on the cluster can then add to it using the **add** method or the += operator (in Scala and Python).
- However, they cannot read its value. Only the driver program can read the accumulator's value, using its **value** method.

The code given below shows an accumulator being used to add up the elements of an array −

```
scala> val accum = sc.accumulator(0)


scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

If you want to see the output of above code then use the following command −

```
scala> accum.value
```

Output

```
res2: Int = 10
```

Numeric RDD Operations

- Spark allows you to do different operations on numeric data, using one of the predefined API methods.
- Spark's numeric operations are implemented with a streaming algorithm that allows building the model, one element at a time.
- These operations are computed and returned as a **StatusCounter** object by calling **status()** method.

The following is a list of numeric methods available in **StatusCounter**.

S.No	Methods & Meaning
1	**count()** Number of elements in the RDD.
2	**Mean()** Average of the elements in the RDD.
3	**Sum()** Total value of the elements in the RDD.

4	**Max()** Maximum value among all elements in the RDD.
5	**Min()** Minimum value among all elements in the RDD.
6	**Variance()** Variance of the elements.
7	**Stdev()** Standard deviation.

If you want to use only one of these methods, you can call the corresponding method directly on RDD.

APACHE STORM

What is Apache Storm?

- Apache Storm is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method.
- It is a streaming data framework that has the capability of highest ingestion rates. Though Storm is stateless, it manages distributed environment and cluster state via Apache ZooKeeper.
- It is simple and you can execute all kinds of manipulations on real-time data in parallel.
- Apache Storm is continuing to be a leader in real-time data analytics. Storm is easy to setup, operate and it guarantees that every message will be processed through the topology at least once.

Apache Storm vs Hadoop

Basically Hadoop and Storm frameworks are used for analyzing big data. Both of them complement each other and differ in some aspects. Apache Storm does all the operations except persistency, while Hadoop is good at everything but lags in real-time computation. The following table compares the attributes of Storm and Hadoop.

Storm	Hadoop
Real-time stream processing	Batch processing
Stateless	Stateful
Master/Slave architecture with ZooKeeper based coordination. The master node is called as nimbus and slaves are supervisors.	Master-slave architecture with/without ZooKeeper based coordination. Master node is job tracker and slave node is task tracker.
A Storm streaming process can access tens of thousands messages per second on cluster.	Hadoop Distributed File System (HDFS) uses MapReduce framework to process vast amount of data that takes minutes or hours.
Storm topology runs until shutdown by the user or an unexpected unrecoverable failure.	MapReduce jobs are executed in a sequential order and completed eventually.
Both are distributed and fault-tolerant	
If nimbus / supervisor dies, restarting makes it continue from where it stopped, hence nothing gets affected.	If the JobTracker dies, all the running jobs are lost.

Use-Cases of Apache Storm

Apache Storm is very famous for real-time big data stream processing. For this reason, most of the companies are using Storm as an integral part of their system. Some notable examples are as follows −

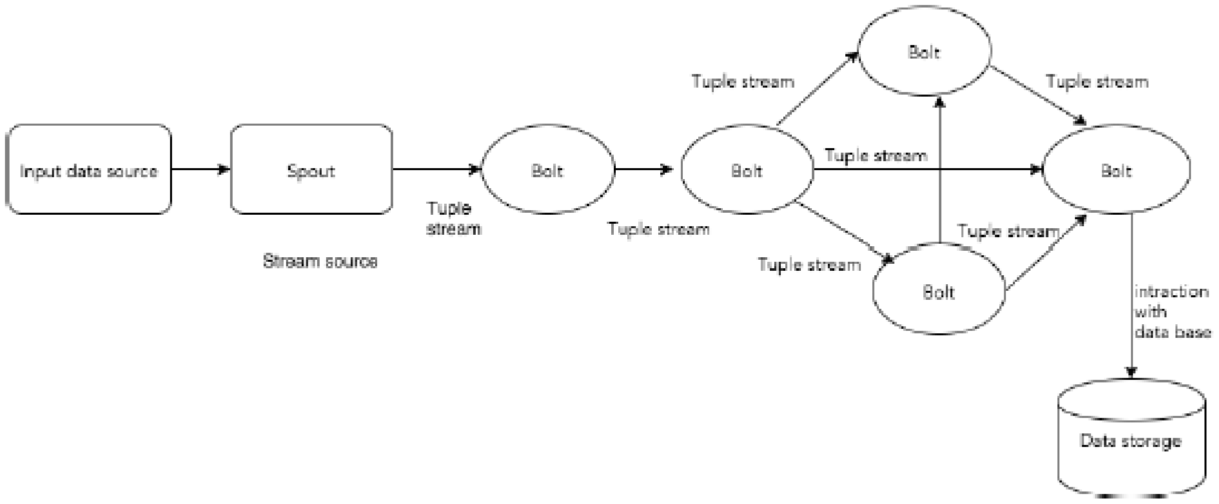
- **Twitter** − Twitter is using Apache Storm for its range of “Publisher Analytics products”. “Publisher Analytics Products” process each and every tweets and clicks in the Twitter Platform. Apache Storm is deeply integrated with Twitter infrastructure.
- **NaviSite** − NaviSite is using Storm for Event log monitoring/auditing system. Every logs generated in the system will go through the Storm. Storm will check the message against the configured set of regular expression and if there is a match, then that particular message will be saved to the database.
- **Wego** − Wego is a travel metasearch engine located in Singapore. Travel related data comes from many sources all over the world with different timing. Storm helps Wego to search real-time data, resolves concurrency issues and find the best match for the end-user.

Apache Storm Benefits

Here is a list of the benefits that Apache Storm offers −

- Storm is open source, robust, and user friendly. It could be utilized in small companies as well as large corporations.
- Storm is fault tolerant, flexible, reliable, and supports any programming language.
- Allows real-time stream processing.
- Storm is unbelievably fast because it has enormous power of processing the data.
- Storm can keep up the performance even under increasing load by adding resources linearly. It is highly scalable.
- Storm performs data refresh and end-to-end delivery response in seconds or minutes depends upon the problem. It has very low latency.
- Storm has operational intelligence.
- Storm provides guaranteed data processing even if any of the connected nodes in the cluster die or messages are lost.
- Apache Storm reads raw stream of real-time data from one end and passes it through a sequence of small processing units and output the processed / useful information at the other end.

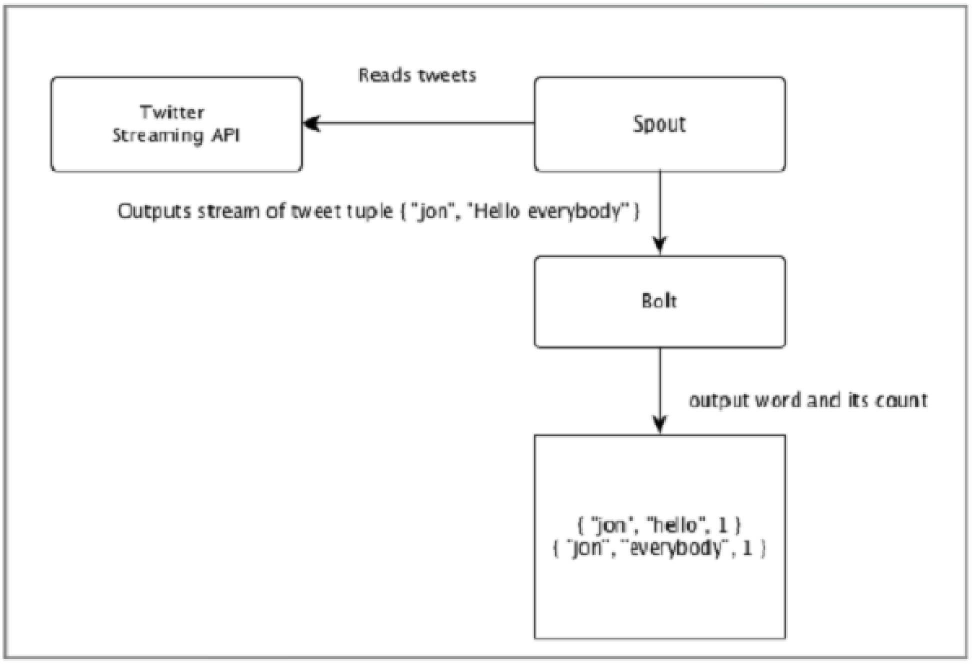
The following diagram depicts the core concept of Apache Storm.



Let us now have a closer look at the components of Apache Storm −

Components	Description
Tuple	Tuple is the main data structure in Storm. It is a list of ordered elements. By default, a Tuple supports all data types. Generally, it is modelled as a set of comma separated values and passed to a Storm cluster.
Stream	Stream is an unordered sequence of tuples.
Spouts	Source of stream. Generally, Storm accepts input data from raw data sources like Twitter Streaming API, Apache Kafka queue, Kestrel queue, etc. Otherwise you can write spouts to read data from datasources. “ISpout" is the core interface for implementing spouts. Some of the specific interfaces are IRichSpout, BaseRichSpout, KafkaSpout, etc.
Bolts	Bolts are logical processing units. Spouts pass data to bolts and bolts process and produce a new output stream. Bolts can perform the operations of filtering, aggregation, joining, interacting with data sources and databases. Bolt receives data and emits to one or more bolts. “IBolt” is the core interface for implementing bolts. Some of the common interfaces are IRichBolt, IBasicBolt, etc.

Let’s take a real-time example of “Twitter Analysis” and see how it can be modelled in Apache Storm. The following diagram depicts the structure.


Reads tweets
Twitter Streaming API
Spout
Outputs stream of tweet tuple { "jon", "Hello everybody" }
Bolt
output word and its count
{ "jon", "hello", 1 }
{ "jon", "everybody", 1 }

- The input for the “Twitter Analysis” comes from Twitter Streaming API. Spout will read the tweets of the users using Twitter Streaming API and output as a stream of tuples.
- A single tuple from the spout will have a twitter username and a single tweet as comma separated values.
- Then, this steam of tuples will be forwarded to the Bolt and the Bolt will split the tweet into individual word, calculate the word count, and persist the information to a configured datasource. Now, we can easily get the result by querying the datasource.

Topology

- Spouts and bolts are connected together and they form a topology. Real-time application logic is specified inside Storm topology. In simple words, a topology is a directed graph where vertices are computation and edges are stream of data.
- A simple topology starts with spouts. Spout emits the data to one or more bolts. Bolt represents a node in the topology having the smallest processing logic and the output of a bolt can be emitted into another bolt as input.
- Storm keeps the topology always running, until you kill the topology. Apache Storm’s main job is to run the topology and will run any number of topology at a given time.

Tasks

- Now you have a basic idea on spouts and bolts. They are the smallest logical unit of the topology and a topology is built using a single spout and an array of bolts. They should be executed properly in a particular order for the topology to run successfully.
- The execution of each and every spout and bolt by Storm is called as “Tasks”. In simple words, a task is either the execution of a spout or a bolt. At a given time, each spout and bolt can have multiple instances running in multiple separate threads.

Workers

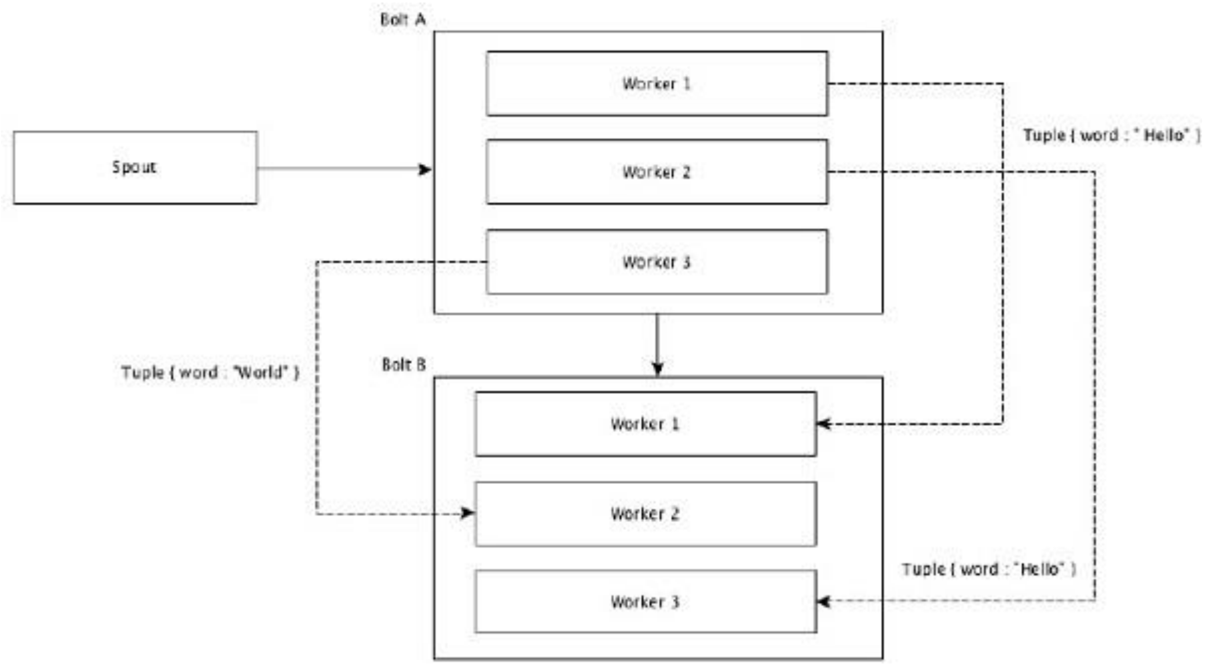
- A topology runs in a distributed manner, on multiple worker nodes. Storm spreads the tasks evenly on all the worker nodes. The worker node’s role is to listen for jobs and start or stop the processes whenever a new job arrives.

Stream Grouping

- Stream of data flows from spouts to bolts or from one bolt to another bolt. Stream grouping controls how the tuples are routed in the topology and helps us to understand the tuples flow in the topology. There are four in-built groupings as explained below.

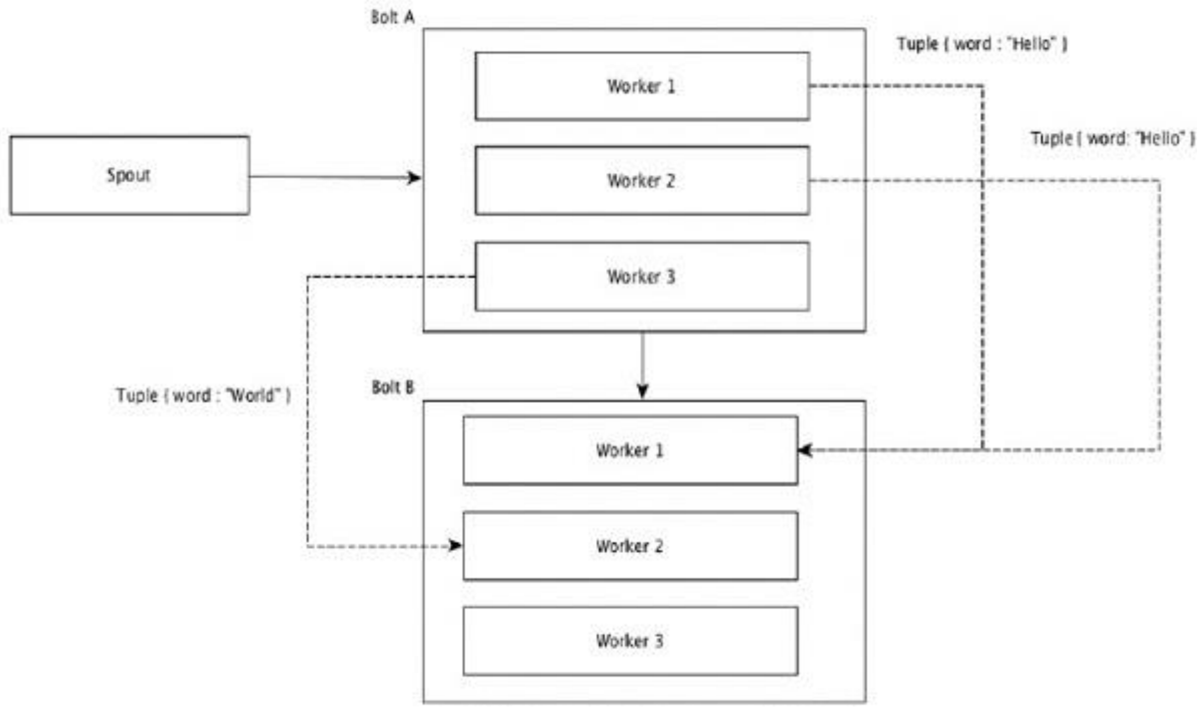
Shuffle Grouping

- In shuffle grouping, an equal number of tuples is distributed randomly across all of the workers executing the bolts. The following diagram depicts the structure.



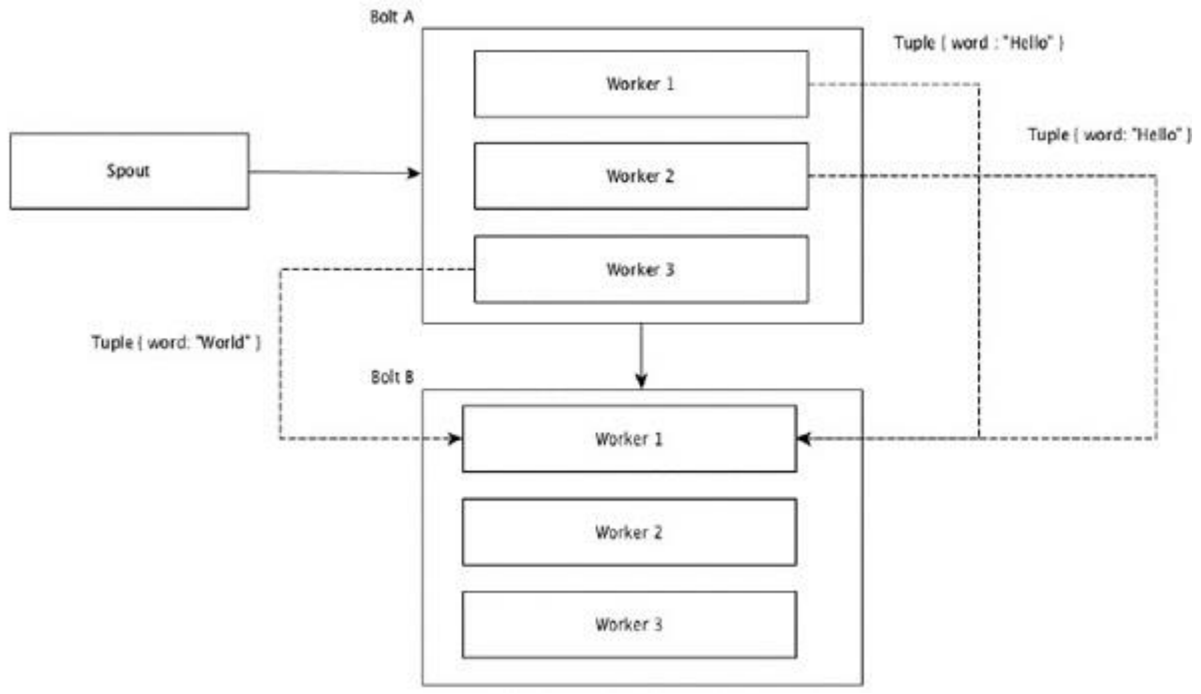
Field Grouping

- The fields with same values in tuples are grouped together and the remaining tuples kept outside. Then, the tuples with the same field values are sent forward to the same worker executing the bolts.
- For example, if the stream is grouped by the field "word", then the tuples with the same string, "Hello" will move to the same worker. The following diagram shows how Field Grouping works.



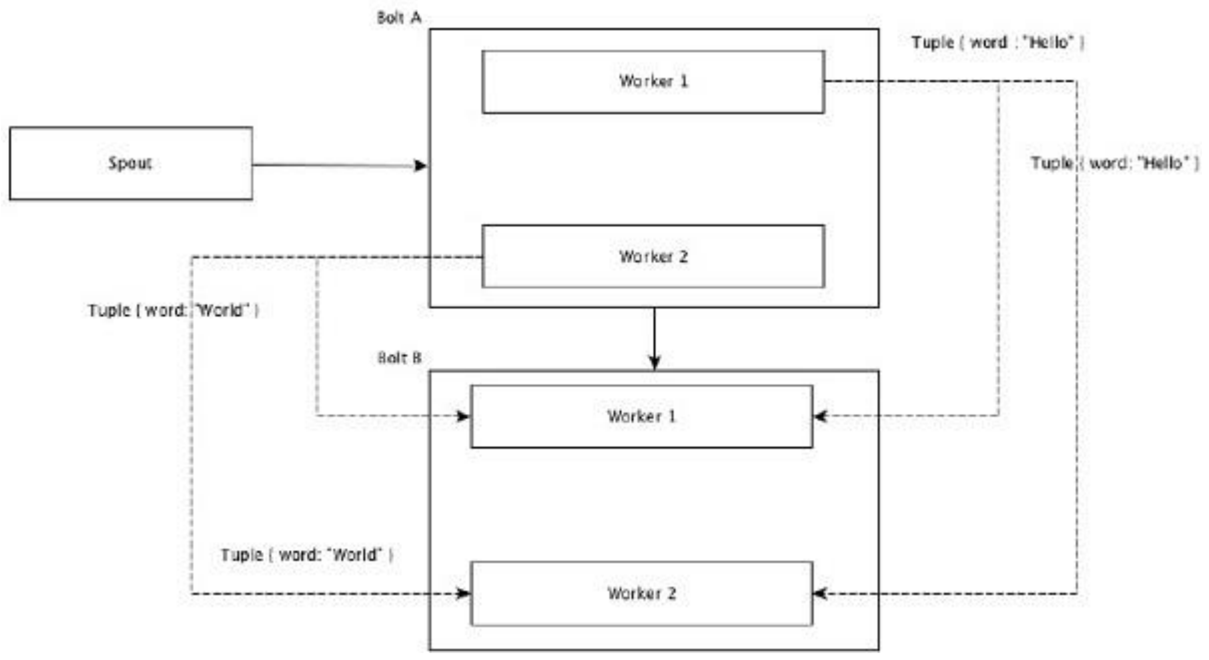
Global Grouping

- All the streams can be grouped and forward to one bolt. This grouping sends tuples generated by all instances of the source to a single target instance (specifically, pick the worker with lowest ID).



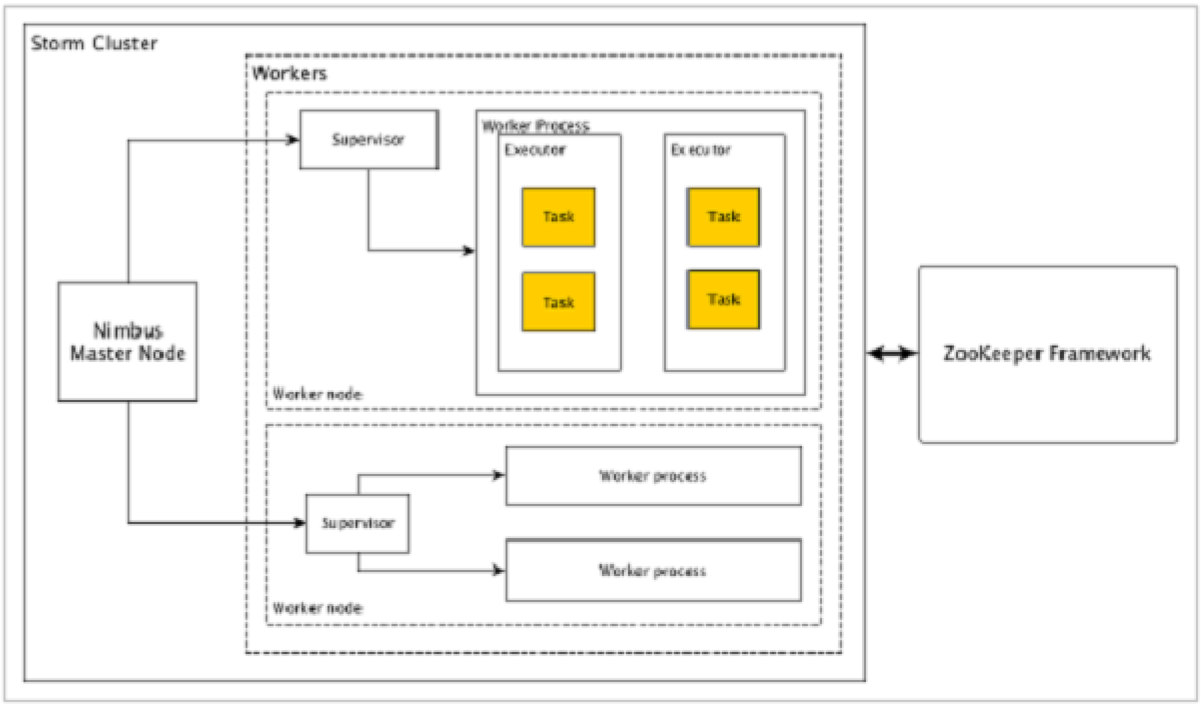
All Grouping

- All Grouping sends a single copy of each tuple to all instances of the receiving bolt. This kind of grouping is used to send signals to bolts. All grouping is useful for join operations.



- One of the main highlight of the Apache Storm is that it is a fault-tolerant, fast with no "Single Point of Failure" (SPOF) distributed application. We can install Apache Storm in as many systems as needed to increase the capacity of the application.

Let's have a look at how the Apache Storm cluster is designed and its internal architecture. The following diagram depicts the cluster design.



- Apache Storm has two type of nodes, Nimbus (master node) and Supervisor(worker node). Nimbus is the central component of Apache Storm.
- The main job of Nimbus is to run the Storm topology. Nimbus analyzes the topology and gathers the task to be executed. Then, it will distributes the task to an available supervisor.
- A supervisor will have one or more worker process. Supervisor will delegate the tasks to worker processes. Worker process will spawn as many executors as needed and run the task.
- Apache Storm uses an internal distributed messaging system for the communication between nimbus and supervisors.

Components	Description
Nimbus	Nimbus is a master node of Storm cluster. All other nodes in the cluster are called as worker nodes. Master node is responsible for distributing data among all the worker nodes, assign tasks to worker nodes and monitoring failures.
Supervisor	The nodes that follow instructions given by the nimbus are called as Supervisors. A supervisor has multiple worker processes and it governs worker processes to complete the tasks assigned by the nimbus.
Worker process	A worker process will execute tasks related to a specific topology. A worker process will not run a task by itself, instead it creates executors and asks them to perform a particular task. A worker process will have multiple executors.
Executor	An executor is nothing but a single thread spawn by a worker process. An executor runs one or more tasks but only for a specific spout or bolt.
Task	A task performs actual data processing. So, it is either a spout or a bolt.
ZooKeeper framework	Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintaining shared data with robust synchronization techniques. Nimbus is stateless, so it depends on ZooKeeper to monitor the working node status. ZooKeeper helps the supervisor to interact with the nimbus. It is responsible to maintain the state of nimbus and supervisor.

- Storm is stateless in nature. Even though stateless nature has its own disadvantages, it actually helps Storm to process real-time data in the best possible and quickest way.
- Storm is *not entirely* stateless though. It stores its state in Apache ZooKeeper. Since the state is available in Apache ZooKeeper, a failed nimbus can be restarted and made to work from where it left.
- Usually, service monitoring tools like monit will monitor Nimbus and restart it if there is any failure.

- Apache Storm also have an advanced topology called Trident Topology with state maintenance and it also provides a high-level API like Pig. We will discuss all these features in the coming chapters.
- A working Storm cluster should have one nimbus and one or more supervisors. Another important node is Apache ZooKeeper, which will be used for the coordination between the nimbus and the supervisors.

Let us now take a close look at the workflow of Apache Storm −

- Initially, the nimbus will wait for the “Storm Topology” to be submitted to it.
- Once a topology is submitted, it will process the topology and gather all the tasks that are to be carried out and the order in which the task is to be executed.
- Then, the nimbus will evenly distribute the tasks to all the available supervisors.
- At a particular time interval, all supervisors will send heartbeats to the nimbus to inform that they are still alive.
- When a supervisor dies and doesn’t send a heartbeat to the nimbus, then the nimbus assigns the tasks to another supervisor.
- When the nimbus itself dies, supervisors will work on the already assigned task without any issue.
- Once all the tasks are completed, the supervisor will wait for a new task to come in.
- In the meantime, the dead nimbus will be restarted automatically by service monitoring tools.
- The restarted nimbus will continue from where it stopped. Similarly, the dead supervisor can also be restarted automatically. Since both the nimbus and the supervisor can be restarted automatically and both will continue as before, Storm is guaranteed to process all the task at least once.
- Once all the topologies are processed, the nimbus waits for a new topology to arrive and similarly the supervisor waits for new tasks.

By default, there are two modes in a Storm cluster −

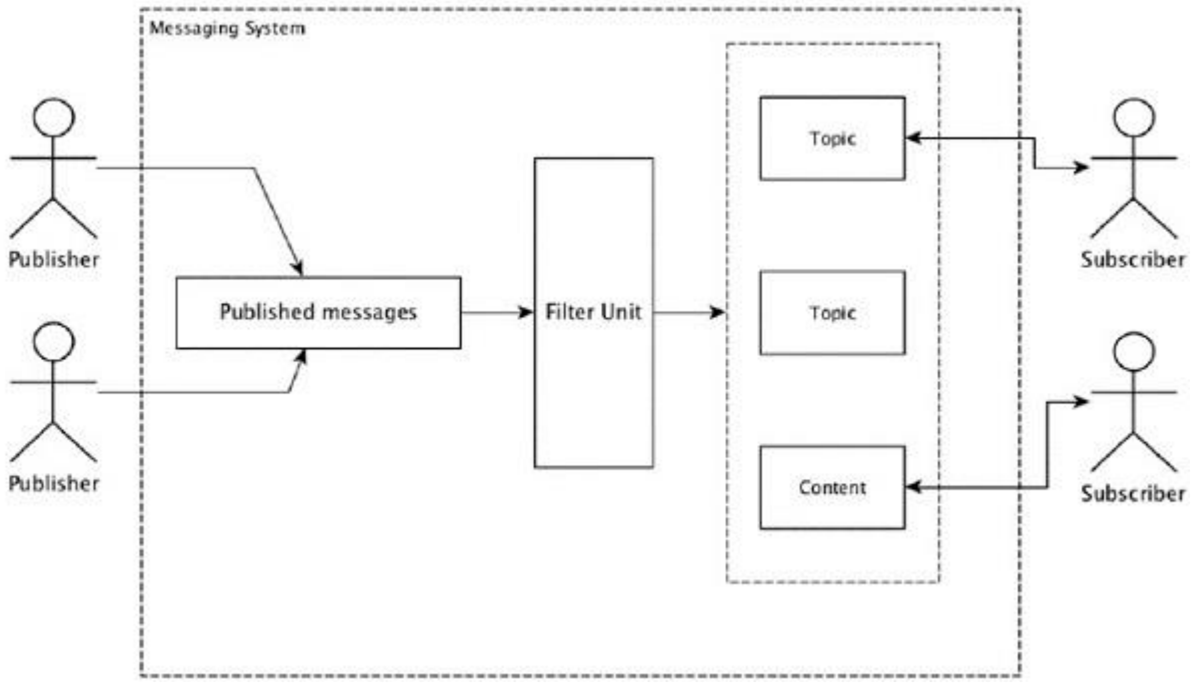
- **Local mode –**
    - This mode is used for development, testing, and debugging because it is the easiest way to see all the topology components working together.
    - In this mode, we can adjust parameters that enable us to see how our topology runs in different Storm configuration environments.
    - In Local mode, storm topologies run on the local machine in a single JVM.

- **Production mode –**
    - In this mode, we submit our topology to the working storm cluster, which is composed of many processes, usually running on different machines.
    - As discussed in the workflow of storm, a working cluster will run indefinitely until it is shut down.

- Apache Storm processes real-time data and the input normally comes from a message queuing system. An external distributed messaging system will provide the input necessary for the realtime computation.
- Spout will read the data from the messaging system and convert it into tuples and input into the Apache Storm. The interesting fact is that Apache Storm uses its own distributed messaging system internally for the communication between its nimbus and supervisor.

What is Distributed Messaging System?

- Distributed messaging is based on the concept of reliable message queuing.
- Messages are queued asynchronously between client applications and messaging systems.
- A distributed messaging system provides the benefits of reliability, scalability, and persistence.
- Most of the messaging patterns follow the publish-subscribe model (simply Pub-Sub) where the senders of the messages are called publishers and those who want to receive the messages are called subscribers.
- Once the message has been published by the sender, the subscribers can receive the selected message with the help of a filtering option.
- Usually we have two types of filtering, one is topic-based filtering and another one is content-based filtering.
- Note that the pub-sub model can communicate only via messages. It is a very loosely coupled architecture; even the senders don't know who their subscribers are. Many of the message patterns enable with message broker to exchange publish messages for timely access by many 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.



The following table describes some of the popular high throughput messaging systems −

Distributed messaging system	Description
Apache Kafka	Kafka was developed at LinkedIn corporation and later it became a sub-project of Apache. Apache Kafka is based on brokerenabled, persistent, distributed publish-subscribe model. Kafka is fast, scalable, and highly efficient.
RabbitMQ	RabbitMQ is an open source distributed robust messaging application. It is easy to use and runs on all platforms.

JMS(Java Message Service)	JMS is an open source API that supports creating, reading, and sending messages from one application to another. It provides guaranteed message delivery and follows publish-subscribe model.
ActiveMQ	ActiveMQ messaging system is an open source API of JMS.
ZeroMQ	ZeroMQ is broker-less peer-peer message processing. It provides push-pull, router-dealer message patterns.
Kestrel	Kestrel is a fast, reliable, and simple distributed message queue.

Thrift Protocol

- Thrift was built at Facebook for cross-language services development and remote procedure call (RPC). Later, it became an open source Apache project.
- Apache Thrift is an Interface Definition Language and allows to define new data types and services implementation on top of the defined data types in an easy manner.
- Apache Thrift is also a communication framework that supports embedded systems, mobile applications, web applications, and many other programming languages.
- Some of the key features associated with Apache Thrift are its modularity, flexibility, and high performance. In addition, it can perform streaming, messaging, and RPC in distributed applications.
- Storm extensively uses Thrift Protocol for its internal communication and data definition. Storm topology is simply Thrift Structs.
- Storm Nimbus that runs the topology in Apache Storm is a Thrift service.

APACHE KAFKA

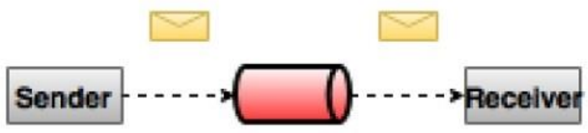
- In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges.The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data.
- To overcome those challenges, you must need a messaging system.
- Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker.
- In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

What is a Messaging System?

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing.
- Messages are queued asynchronously between client applications and messaging system. Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow pub-sub.

Point to Point Messaging System

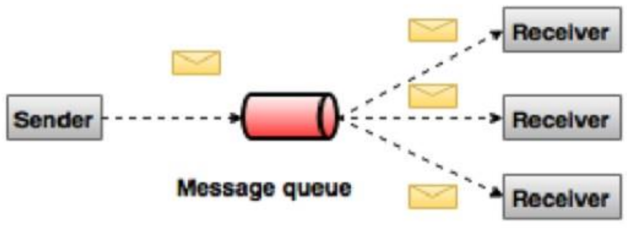
- In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only.
- Once a consumer reads a message in the queue, it disappears from that queue.
- The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.



Message queue

Publish-Subscribe Messaging System

- In the publish subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. 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.



What is Kafka?

- 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 with Apache Storm and Spark for real-time streaming data analysis.

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.
- Kafka is very fast and guarantees zero downtime and zero data loss.

Use Cases

Kafka can be used in many Use Cases. Some of them are listed below −

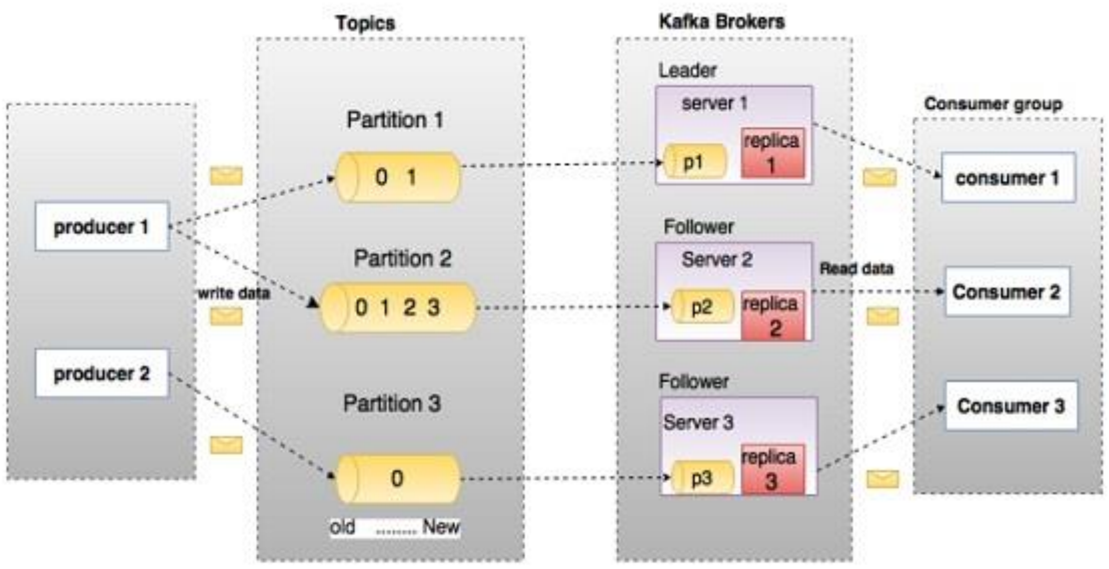
- **Metrics −** Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
- **Log Aggregation Solution −** Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple con-sumers.
- **Stream Processing −** Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.

Need for Kafka

- Kafka is a unified platform for handling all the real-time data feeds.
- Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the ability to handle a large number of diverse consumers.

- Kafka is very fast, performs 2 million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.
- Before moving deep into the Kafka, you must aware of the main terminologies such as topics, brokers, producers and consumers.

The following diagram illustrates the main terminologies and the table describes the diagram components in detail.

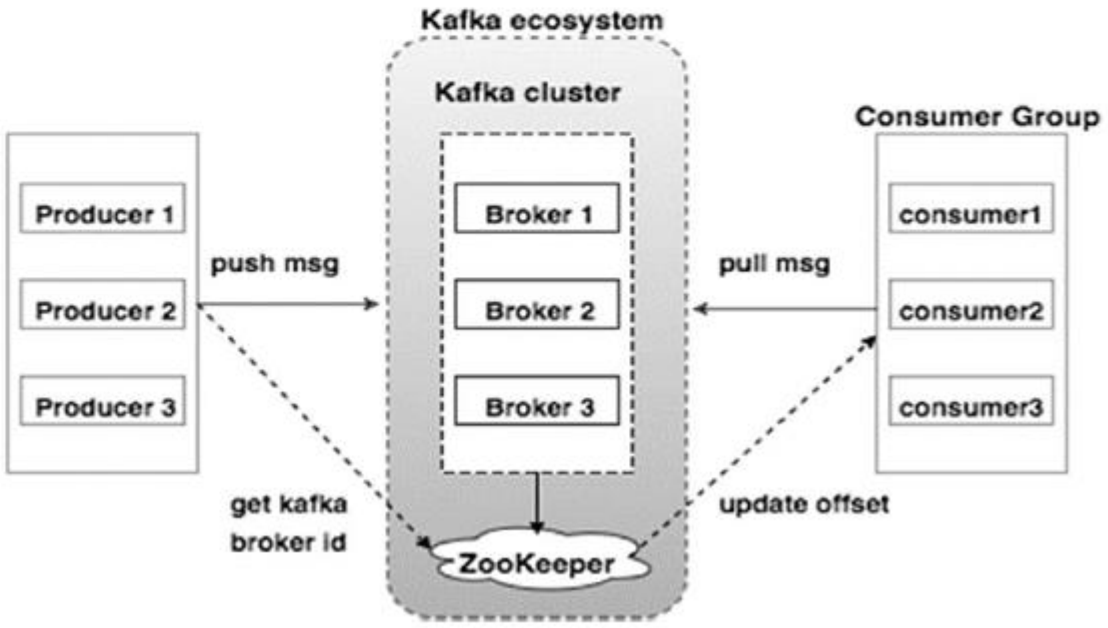


- In the above diagram, a topic is configured into three partitions.
- Partition 1 has two offset factors 0 and 1.
- Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0.
- The id of the replica is same as the id of the server that hosts it.
- Assume, if the replication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

S.No	Components and Description
1	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 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.
2	Partition Topics may have many partitions, so it can handle an arbitrary amount of data.
3	Partition offset

	Each partitioned message has a unique sequence id called as offset.
4	Replicas of partition Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.
5	Brokers Brokers are simple system responsible for maintaining the pub-lished data. 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. Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic. Assume 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 distri-bution among the broker.
6	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.
7	Producers Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer pub-lishes 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.
8	Consumers Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers.
9	Leader Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.
10	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.

Take a look at the following illustration. It shows the cluster diagram of Kafka.



The following table describes each of the components shown in the above diagram.

S.No	Components and Description
1	Broker Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.
2	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 pro-ducer and consumer takes decision and starts coordinating their task with some other broker.
3	Producers Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
4	Consumers Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer

	acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

- As of now, we discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.
- Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset).
- All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.
- Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner.
- In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need.

Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

Workflow of Pub-Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging −

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

Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.

2. **Consumer subscribes to a specific topic.**

Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.

3. **Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.**

Once Kafka receives the messages from producers, it forwards these messages to the consumers.

4. **Consumer will receive the message and process it.**

Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.

Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.

5. **This above flow will repeat until the consumer stops the request.**

Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of Queue Messaging / Consumer Group

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

- Producers send message to a topic in a regular interval.
- Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
- A single consumer subscribes to a specific topic, assume Topic-01with Group ID as Group-1.
- Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01with the same Group ID as Group-1.
- Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of consumers reach the number of partition configured for that particular topic.
- Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
- This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Role of ZooKeeper

- A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service.
- Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster.
- Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.
- Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts.
- This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.