Unit 2

Hadoop: History of Hadoop, Apache Hadoop, the Hadoop Distributed File System, components of Hadoop, data format, analyzing data with Hadoop, scaling out, Hadoop streaming, Hadoop pipes, Hadoop Echo System. Map-Reduce: Map-Reduce framework and basics, how Map Reduce works, developing a Map Reduce application, unit tests with MR unit, test data and local tests, anatomy of a Map Reduce job run, failures, job scheduling, shuffle and sort, task execution, Map Reduce types, input formats, output formats, Map Reduce features, Real-world Map Reduce

History of Hadoop

Hadoop is an open source framework overseen by Apache Software Foundation which is written in Java for storing and processing of huge datasets with the cluster of commodity hardware. There are mainly two problems with the big data. First one is to store such a huge amount of data and the second one is to process that stored data. The traditional approach like RDBMS is not sufficient due to the heterogeneity of the data. So Hadoop comes as the solution to the problem of big data i.e. storing and processing the big data with some extra capabilities. There are mainly two components of Hadoop which are Hadoop Distributed File System (HDFS) **and** Yet Another Resource Negotiator(YARN)**.**

Hadoop History

Hadoop was started with Doug Cutting and Mike Cafarella in the year 2002 when they both started to work on Apache Nutch project. Apache Nutch project was the process of building a search engine system that can index 1 billion pages. After a lot of research on Nutch, they concluded that such a system will cost around half a million dollars in hardware, and along with a monthly running cost of $30, 000 approximately, which is very expensive. So, they realized that their project architecture will not be capable enough to the workaround with billions of pages on the web. So they were looking for a feasible solution which can reduce the implementation cost as well as the problem of storing and processing of large datasets.

In 2003, they came across a paper that described the architecture of Google's distributed file system, called GFS (Google File System) which was published by Google, for storing the large data sets. Now they realize that this paper can solve their problem of storing very large files which were being generated because of web crawling and indexing processes. But this paper was just the half solution to their problem.

In 2004, Google published one more paper on the technique MapReduce, which was the solution of processing those large datasets. Now this paper was another half solution for Doug Cutting and Mike Cafarella for their Nutch project. These both techniques (GFS & MapReduce) were just on white paper at Google. Google didn't implement these two techniques. Doug Cutting knew from his work on Apache Lucene (It is a free and open-source information retrieval software library, originally written in Java by Doug Cutting in 1999) that open-source is a great way to spread the technology to more people. So, together with Mike Cafarella, he started implementing Google's techniques (GFS & MapReduce) as open-source in the Apache Nutch project.

In 2005, Cutting found that Nutch is limited to only 20-to-40 node clusters. He soon realized two problems:

(a) Nutch wouldn't achieve its potential until it ran reliably on the larger clusters
(b) And that was looking impossible with just two people (Doug Cutting & Mike Cafarella). The engineering task in Nutch project was much bigger than he realized. So he started to find a job with a company who is interested in investing in their efforts. And he found Yahoo!.Yahoo had a large team of engineers that was eager to work on this there project.

So in 2006, Doug Cutting joined Yahoo along with Nutch project. He wanted to provide the world with an open-source, reliable, scalable computing framework, with the help of Yahoo. So at Yahoo first, he separates the distributed computing parts from Nutch and formed a new project Hadoop (He gave name Hadoop it was the name of a yellow toy elephant which was owned by the Doug Cutting's son. and it was easy to pronounce and was the unique word.) Now he wanted to make Hadoop in such a way that it can work well on thousands of nodes. So with GFS and MapReduce, he started to work on Hadoop.

In 2007, Yahoo successfully tested Hadoop on a 1000 node cluster and start using it.

In January of 2008, Yahoo released Hadoop as an open source project to ASF(Apache Software Foundation). And in July of 2008, Apache Software Foundation successfully tested a 4000 node cluster with Hadoop.

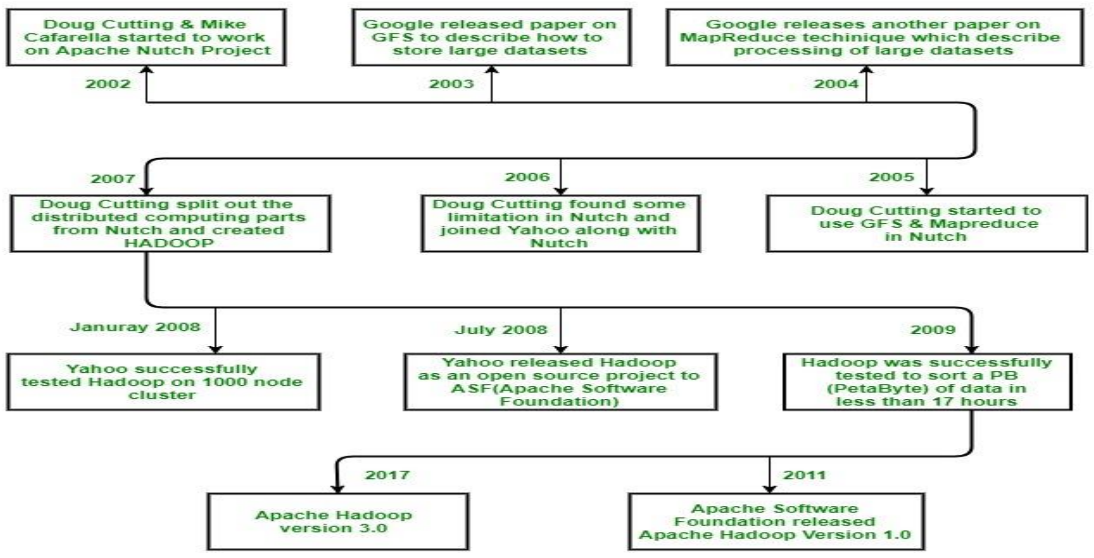
In 2009, Hadoop was successfully tested to sort a PB (PetaByte) of data in less than 17 hours for handling billions of searches and indexing millions of web pages. And Doug Cutting left the Yahoo and joined Cloudera to fulfill the challenge of spreading Hadoop to other industries.

In December of 2011, Apache Software Foundation released Apache Hadoop version 1.0.

And later in Aug 2013, Version 2.0.6 was available.

And currently, we have Apache Hadoop version 3.0 which released in December 2017.

Let's Summarize above History :


Doug Cutting & Mike Cafarella started to work on Apache Nutch Project
Google released paper on GFS to describe how to store large datasets
Google releases another paper on MapReduce techinique which describe processing of large datasets
2002
2003
2004
2007
2006
2005
Doug Cutting split out the distributed computing parts from Nutch and created HADOOP
Doug Cutting found some limitation in Nutch and joined Yahoo along with Nutch
Doug Cutting started to use GFS & Mapreduce in Nutch
Januray 2008
July 2008
2009
Yahoo successfully tested Hadoop on 1000 node cluster
Yahoo released Hadoop as an open source project to ASF(Apache Software Foundation)
Hadoop was successfully tested to sort a PB (PetaByte) of data in less than 17 hours
2017
2011
Apache Hadoop version 3.0
Apache Software Foundation released Apache Hadoop Version 1.0

Apache Hadoop

Apache Hadoop is a collection of open-source software utilities that facilitates using a network of many computers to solve problems involving massive amounts of data and computation. It provides a software framework for distributed storage and processing of big data using the MapReduce programming model. Hadoop was originally designed for computer clusters built from commodity hardware, which is still the common use. It has since also found use on clusters of higher-end hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel. This approach takes advantage of data locality, where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities needed by other Hadoop modules;
- Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – (introduced in 2012) a platform responsible for managing computing resources in clusters and using them for scheduling users' applications;[10][11]
- Hadoop MapReduce – an implementation of the MapReduce programming model for large-scale data processing.
- Hadoop Ozone – (introduced in 2020) An object store for Hadoop

The term Hadoop is often used for both base modules and sub-modules and also the ecosystem, or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, and Apache Storm.[13]

Apache Hadoop's MapReduce and HDFS components were inspired by Google papers on MapReduce and Google File System.

The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell scripts. Though MapReduce Java code is common, any programming language can be used with Hadoop Streaming to implement the map and reduce parts of the user's program.[15] Other projects in the Hadoop ecosystem expose richer user interfaces.

The Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is the primary data storage system used by name node and data node by Hadoop applications. HDFS employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters.

Hadoop itself is an open source distributed processing framework that manages data processing and storage for big data applications. HDFS is a key part of the many Hadoop ecosystem technologies. It provides a reliable means for managing pools of big data and supporting related big data analytics applications.

How does HDFS work?

HDFS enables the rapid transfer of data between compute nodes. At its outset, it was closely coupled with MapReduce, a framework for data processing that filters and divides up work among the nodes in a cluster, and it organizes and condenses the results into a cohesive answer to a query. Similarly, when HDFS takes in data, it breaks the information down into separate blocks and distributes them to different nodes in a cluster.

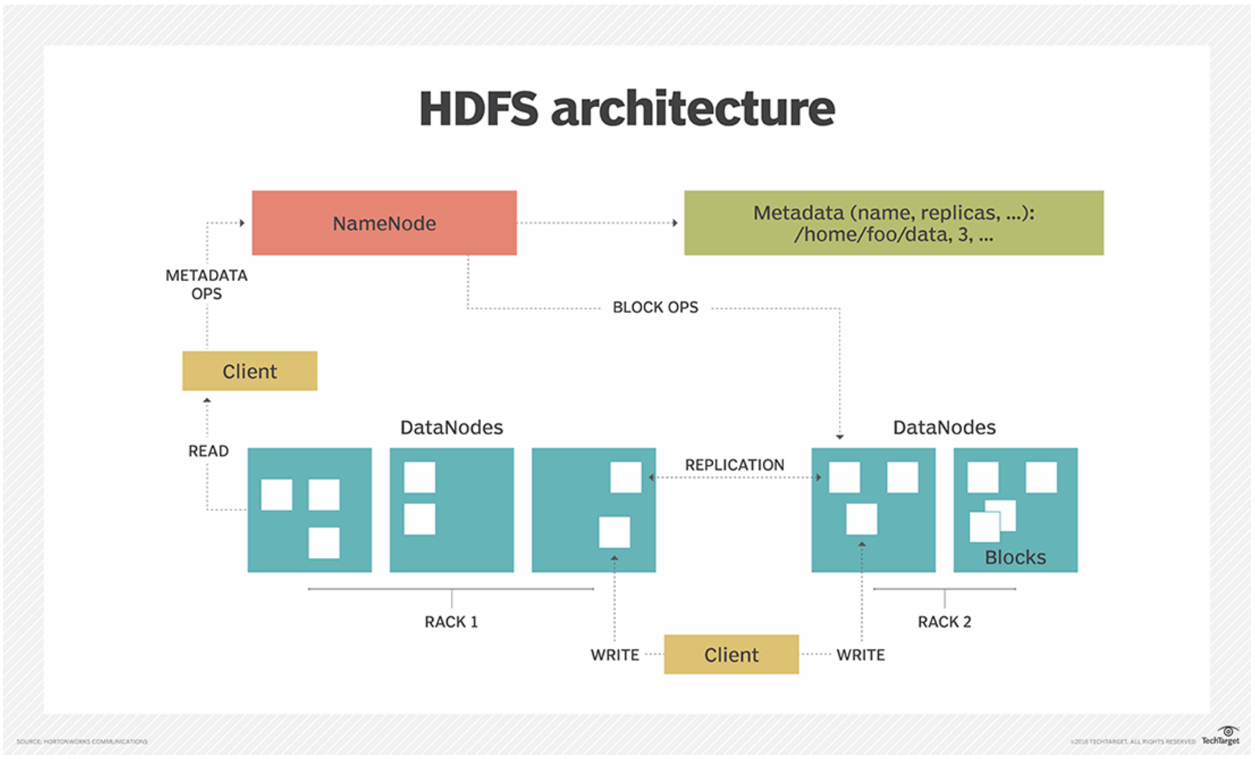
With HDFS, data is written on the server once, and read and reused numerous times after that. HDFS has a primary NameNode, which keeps track of where file data is kept in the cluster.

HDFS also has multiple DataNodes on a commodity hardware cluster -- typically one per node in a cluster. The DataNodes are generally organized within the same rack in the data center. Data is broken down into separate blocks and distributed among the various DataNodes for storage. Blocks are also replicated across nodes, enabling highly efficient parallel processing.

The NameNode knows which DataNode contains which blocks and where the DataNodes reside within the machine cluster. The NameNode also manages access to the files, including reads, writes, creates, deletes and the data block replication across the DataNodes.

The NameNode operates in conjunction with the DataNodes. As a result, the cluster can dynamically adapt to server capacity demand in real time by adding or subtracting nodes as necessary.

The DataNodes are in constant communication with the NameNode to determine if the DataNodes need to complete specific tasks. Consequently, the NameNode is always aware of the status of each DataNode. If the NameNode realizes that one DataNode isn't working properly, it can immediately reassign that DataNode's task to a different node containing the same data block. DataNodes also communicate with each other, which enables them to cooperate during normal file operations.
Moreover, the HDFS is designed to be highly fault-tolerant. The file system replicates -- or copies -- each piece of data multiple times and distributes the copies to individual nodes, placing at least one copy on a different server rack than the other copies.



HDFS architecture, NameNodes and DataNodes
HDFS uses a primary/secondary architecture. The HDFS cluster's NameNode is the primary server that manages the file system namespace and controls client access to files. As the central component of the Hadoop Distributed File System, the NameNode maintains and manages the file system namespace and provides clients with the right access permissions. The system's DataNodes manage the storage that's attached to the nodes they run on.
HDFS exposes a file system namespace and enables user data to be stored in files. A file is split into one or more of the blocks that are stored in a set of DataNodes. The NameNode performs file system namespace operations, including opening, closing and renaming files and directories. The NameNode also governs the mapping of blocks to the DataNodes. The DataNodes serve read and write requests from the clients of the file system. In addition, they perform block creation, deletion and replication when the NameNode instructs them to do so.
HDFS supports a traditional hierarchical file organization. An application or user can create directories and then store files inside these directories. The file system namespace hierarchy is like most other file systems -- a user can create, remove, rename or move files from one directory to another.
The NameNode records any change to the file system namespace or its properties. An application can stipulate the number of replicas of a file that the HDFS should maintain. The NameNode stores the number of copies of a file, called the replication factor of that file.

Components of Hadoop

There are four major elements or component of Hadoop i.e. HDFS, MapReduce, YARN, and Hadoop Common. Most of the tools or solutions are used to supplement or support these major elements. All these tools work collectively to provide services such as absorption, analysis, storage and maintenance of data etc.

Following are the components that collectively form a Hadoop ecosystem:

- HDFS: Hadoop Distributed File System
- YARN: Yet Another Resource Negotiator
- MapReduce: Programming based Data Processing
- Spark: In-Memory data processing
- PIG, HIVE: Query based processing of data services
- HBase: NoSQL Database
- Mahout, Spark MLLib: Machine Learning algorithm libraries
- Solar, Lucene: Searching and Indexing
- Zookeeper: Managing cluster
- Oozie: Job Scheduling

Data Format in Hadoop

The file format in Hadoop roughly divided into two categories: **row-oriented and column-oriented:**

Row-oriented:

The same row of data stored together that is continuous storage: SequenceFile, MapFile, Avro Datafile. In this way, if only a small amount of data of the row needs to be accessed, the entire row needs to be read into the memory. Delaying the serialization can lighten the problem to a certain amount, but the overhead of reading the whole row of data from the disk cannot be withdrawn. Row-oriented storage is suitable for situations where the entire row of data needs to be processed simultaneously.

Column-oriented:

The entire file cut into several columns of data, and each column of data stored together: Parquet, RCFile, ORCFile. The column-oriented format makes it possible to skip unneeded columns when reading data, suitable for situations where only a small portion of the rows are in the field. But this format of reading and write requires more memory space because the cache line needs to be in memory (to get a column in multiple rows). At the same time, it is not suitable for streaming to write, because once the write fails, the current file cannot be recovered, and the line-oriented data can be resynchronized to the last synchronization point when the write fails, so Flume uses the line-oriented storage format.

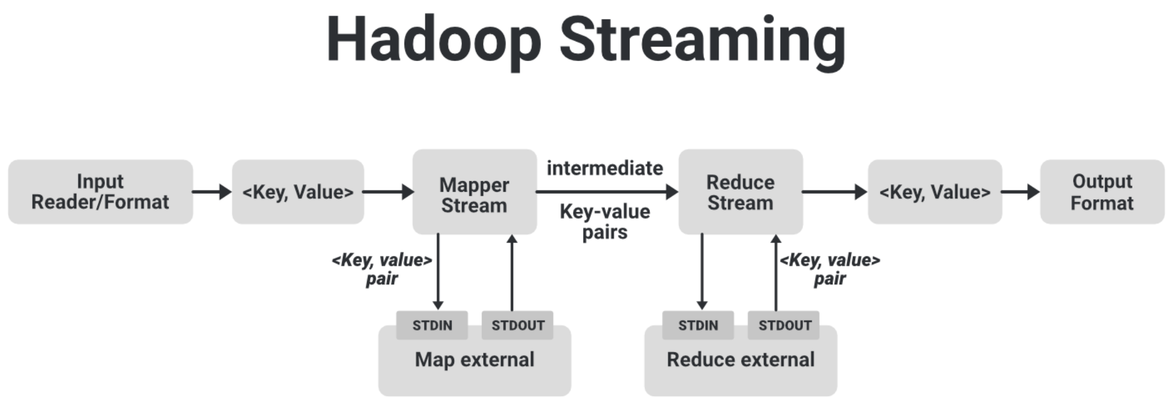
Analyzing data with Hadoop

While the MapReduce programming model is at the heart of Hadoop, it is low-level and as such becomes a unproductive way for developers to write complex analysis jobs. To increase developer productivity, several higher-level languages and APIs have been created that abstract away the low-level details of the MapReduce programming model. There are several choices available for writing data analysis jobs. The Hive and Pig projects are popular choices that provide SQL-like and procedural data flow-like languages, respectively. HBase is also a popular way to store and analyze data in HDFS. It is a column-oriented database, and unlike MapReduce, provides random read and write access to data with low latency. MapReduce jobs can read and write data in HBase's table format, but data processing is often done via HBase's own client API. In this chapter, we will show how to use Spring for Apache Hadoop to write Java applications that use these Hadoop technologies.

Scaling out

Scaling are of two types
Vertical scaling : When we add more resources to a single machine when the load increases. For example you need 20gb of ram but currently your server has 10 GB of ram so you add extra ram to the same server to meet the needs.
Horizontal scaling of scaling out : when you add more machines to match the resources need it's called horizontal scaling. So if I have a machine of already 10 GB I'll add an extra machine with 10 GB ram.

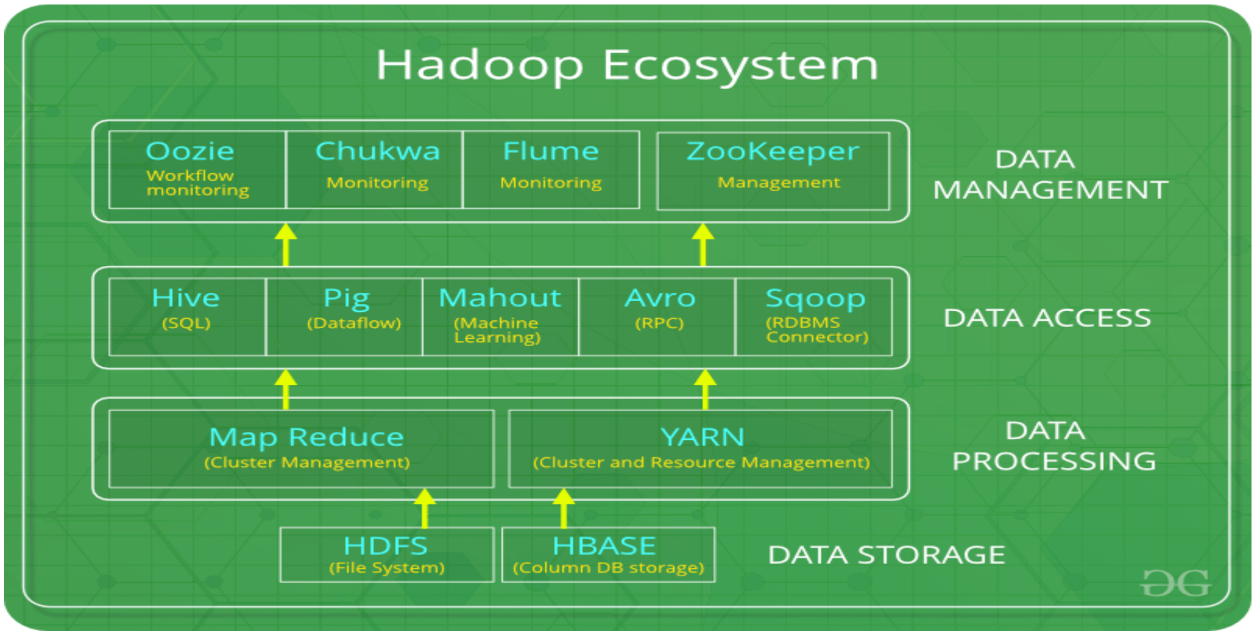
Hadoop Streaming
It is a utility or feature that comes with a Hadoop distribution that allows developers or programmers to write the Map-Reduce program using different programming languages like Ruby, Perl, Python, C++, etc. We can use any language that can read from the standard input(STDIN) like keyboard input and all and write using standard output(STDOUT). We all know the Hadoop Framework is completely written in java but programs for Hadoop are not necessarily need to code in Java programming language. feature of Hadoop Streaming is available since Hadoop version 0.14.1.



Hadoop Pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Unlike Streaming, which uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the tasktracker communicates with the process running the C++ map or reduce function. JNI is not used.

Hadoop Ecosystem



MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a map procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is a specialization of the split-apply-combine strategy for data analysis. It is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms. The key contributions of the MapReduce framework are not the actual map and reduce functions (which, for example, resemble the 1995 Message Passing Interface standard's reduce and scatter operations), but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine. As such, a single-threaded implementation of MapReduce is usually not faster than a traditional (non-MapReduce) implementation; any gains are usually only seen with multi-threaded implementations on multi-processor hardware. The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play. Optimizing the communication cost is essential to a good MapReduce algorithm.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been genericized. By 2014, Google was no longer using MapReduce as their primary big data processing model, and development on Apache Mahout had moved on to more capable and less disk-oriented mechanisms that incorporated full map and reduce capabilities.

MapReduce Framework

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware). Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

A MapReduce framework (or system) is usually composed of three operations (or steps):

1. **Map:** each worker node applies the `map` function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of the redundant input data is processed.
2. **Shuffle:** worker nodes redistribute data based on the output keys (produced by the `map` function), such that all data belonging to one key is located on the same worker node.
3. **Reduce:** worker nodes now process each group of output data, per key, in parallel.

MapReduce allows for the distributed processing of the map and reduction operations. Maps can be performed in parallel, provided that each mapping operation is independent of the others; in practice, this is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time, or that the reduction function is associative. While this process often appears inefficient compared to algorithms that are more sequential (because multiple instances of the reduction process must be run), MapReduce can be applied to significantly larger datasets than a single "commodity" server can handle – a large server farm can use MapReduce to sort a petabyte of data in only a few hours. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data are still available.

Another way to look at MapReduce is as a 5-step parallel and distributed computation:

1. **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key K1 that each processor would work on, and provides that processor with all the input data associated with that key.
2. **Run the user-provided Map() code** – Map() is run exactly once for each K1 key, generating output organized by key K2.
3. **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the K2 key each processor should work on, and provides that processor with all the Map-generated data associated with that key.
4. **Run the user-provided Reduce() code** – Reduce() is run exactly once for each K2 key produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by K2 to produce the final outcome.

These five steps can be logically thought of as running in sequence – each step starts only after the previous step is completed – although in practice they can be interleaved as long as the final result is not affected.

In many situations, the input data might have already been distributed ("sharded") among many different servers, in which case step 1 could sometimes be greatly simplified by assigning Map servers that would process the locally present input data. Similarly, step 3 could sometimes be sped up by assigning Reduce processors that are as close as possible to the Map-generated data they need to process.

How Hadoop MapReduce works?

The whole process goes through various MapReduce phases of execution, namely, splitting, mapping, sorting and shuffling, and reducing. Let us explore each phase in detail.

1. InputFiles
The data that is to be processed by the MapReduce task is stored in input files. These input files are stored in the Hadoop Distributed File System. The file format is arbitrary, while the line-based log files and the binary format can also be used.
2. InputFormat
It specifies the input-specification for the job. InputFormat validates the MapReduce job input-specification and splits-up the input files into logical InputSplit instances. Each InputSplit is then assigned to the individual Mapper. TextInputFormat is the default InputFormat.
3. InputSplit
It represents the data for processing by the individual Mapper. InputSplit typically presents the byte-oriented view of the input. It is the RecordReader responsibility to process and present the record-oriented view. The default InputSplit is the FileSplit.
4. RecordReader
RecordReader reads the <key, value> pairs from the InputSplit. It converts a byte-oriented view of the input and presents a record-oriented view to the Mapper implementations for processing.
It is responsible for processing record boundaries and presenting the Map tasks with keys and values. The record reader breaks the data into the <key, value> pairs for input to the Mapper.
5. Mapper
Mapper maps the input <key, value> pairs to a set of intermediate <key, value> pairs. It processes the input records from the RecordReader and generates the new <key, value> pairs. The <key, value> pairs generated by Mapper are different from the input <key, value> pairs.
The generated <key, value> pairs is the output of Mapper known as intermediate output. These intermediate outputs of the Mappers are written to the local disk.
The Mappers output is not stored on the Hadoop Distributed File System because this is the temporary data, and writing this data on HDFS will create unnecessary copies. The output of the Mappers is then passed to the Combiner for further processing.
6. Combiner
It is also known as the ‘Mini-reducer’. Combiner performs local aggregation on the output of the Mappers. This helps in minimizing data transfer between the Mapper and the Reducer.
After the execution of the Combiner function, the output is passed to the Partitioner for further processing.
7. Partitioner
When we are working on the MapReduce program with more than one Reducer then only the Partitioner comes into the picture. For only one reducer, we do not use Partitioner.
It partitions the keyspace. It controls the partitioning of keys of the Mapper intermediate outputs.
Partitioner takes the output from the Combiner and performs partitioning. Key is for deriving the partition typically through the hash function. The number of partitions is similar to the number of reduce tasks. HashPartitioner is the default Partitioner.
8. Shuffling and Sorting
The input to the Reducer is always the sorted intermediate output of the mappers. After combining and partitioning, the framework via HTTP fetches all the relevant partitions of the output of all the mappers.
Once the output of all the mappers is shuffled, the framework groups the Reducer inputs on the basis of the keys. This is then provided as an input to the Reducer.
9. Reducer
Reducer then reduces the set of intermediate values who shares a key to the smaller set of values. The output of reducer is the final output. This output is stored in the Hadoop Distributed File System.
10. RecordWriter
RecordWriter writes the output (key, value pairs) of Reducer to an output file. It writes the MapReduce job outputs to the FileSystem.
11. OutputFormat

The OutputFormat specifies the way in which these output key-value pairs are written to the output files. It validates the output specification for a MapReduce job.
OutputFormat basically provides the RecordWriter implementation used for writing the output files of the MapReduce job. The output files are stored in a FileSystem.
Hence, in this manner, MapReduce works over the Hadoop cluster in different phases.

Developing MapReduce Application

1. Configuration API. ...
2. Configuring the Development Environment. ...
3. GenericOptionsParser, Tool and ToolRunner. ...
4. Writing Unit Tests. ...
5. Running locally and in a cluster on Test Data. ...
6. The MapReduce Web UI. ...
7. Hadoop Logs. ...
8. Tuning a Job to improve performance.

Unit Test with MRUnit

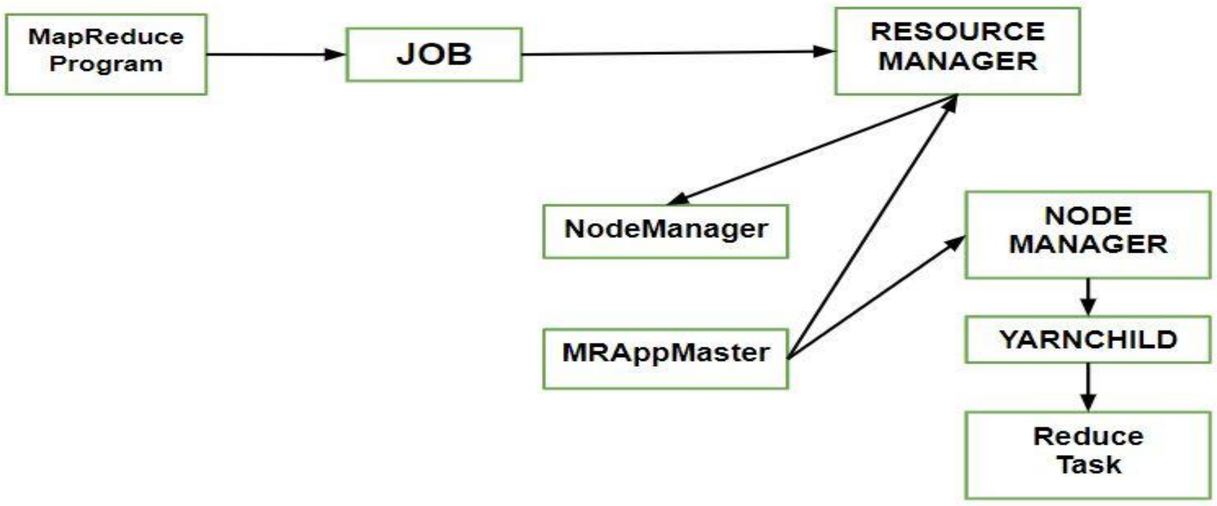
Hadoop MapReduce jobs have a unique code architecture that follows a specific template with specific constructs. This architecture raises interesting issues when doing test-driven development (TDD) and writing unit tests. This is a real-world example using MRUnit, Mockito, and PowerMock. I will touch upon 1) using MRUnit to write JUnit tests for hadoop MR applications, 2) using PowerMock & Mockito to mock static methods, 3) mocking-out business-logic contained in another class, 4) verifying that mocked-out business logic was called (or not) 5) testing counters, 6) testing statements in a log4j conditional block, and 7) handling exceptions in tests. I'm assuming the reader is already familiar with JUnit 4.

With MRUnit, you can craft test input, push it through your mapper and/or reducer, and verify it's output all in a JUnit test. As do other JUnit tests, this allows you to debug your code using the JUnit test as a driver. A map/reduce pair can be tested using MRUnit's MapReduceDriver. A combiner can be tested using MapReduceDriver as well. A PipelineMapReduceDriver allows you to test a workflow of map/reduce jobs. Currently, partitioners do not have a test driver under MRUnit. MRUnit allows you to do TDD and write light-weight unit tests which accommodate Hadoop's specific architecture and constructs.

Anatomy of MapReduce Job

MapReduce can be used to work with a solitary method call: **submit()** on a Job object (you can likewise call **waitForCompletion()**, which presents the activity on the off chance that it hasn't been submitted effectively, at that point sits tight for it to finish).
Let's understand the components –

1. **Client :** Submitting the MapReduce job.
2. **Yarn node manager :** In a cluster , it monitors and launches the compute containers on machines.
3. **Yarn resource manager :** Handles the allocation of compute resources coordination on the cluster.
4. **MapReduce application master :** Facilitates the tasks running the MapReduce work.
5. **Distributed Filesystem :** Shares job files with other entities.



How to submit Job?
To create an internal JobSubmitter instance, use the **submit()** which further calls **submitJobInternal()** on it. Having submitted the job, **waitForCompletion()** polls the job's progress after submitting the job once per second. If the reports have changed since the last report, it further reports the progress to the console. The job counters are displayed when the job completes successfully. Else the error (that caused the job to fail) is logged to the console.
Processes implemented by **JobSubmitter** for submitting the Job :

- The resource manager askes for a new application ID that is used for MapReduce Job ID.
- Output specification of the job is checked. For e.g. an error is thrown to the MapReduce program or the job is not submitted or the output directory already exists or it has not been specified.
- If the splits cannot be computed, it computes the input splits for the job. This can be due to the job is not submitted and an error is thrown to the MapReduce program.
- Resources needed to run the job is copied – it includes the job JAR file, the computed input splits, to the shared filesystem in a directory named after the job ID and the configuration file.
- It copies job **JAR** with a high replication factor, which is controlled by **mapreduce.client.submit.file.replication property**. AS there are the number of copies across the cluster for the node managers to access.
- By calling **submitApplication()**, submits the job on the resource manager.

Shuffling in MapReduce

The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate **key-value pairs** in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

Note that shuffling and sorting in Hadoop MapReduce is not performed at all if you specify zero reducers (setNumReduceTasks(0)). Then, the MapReduce job stops at the map phase, and the map phase does not include any kind of sorting (so even the map phase is faster).

MapReduce Types and Formats

Mapping is the core technique of processing a list of data elements that come in pairs of keys and values. The map function applies to individual elements defined as key-value pairs of a list and produces a new list. The general idea of map and reduce function of Hadoop can be illustrated as follows:

```
map: (K1, V1) -> list (K2, V2)


reduce: (K2, list(V2)) -> list (K3, V3)
```

The input parameters of the key and value pair, represented by K1 and V1 respectively, are different from the output pair type: K2 and V2. The reduce function accepts the same format output by the map, but the type of output again of the reduce operation is different: K3 and V3.

The *OutputCollector* is the generalized interface of the Map-Reduce framework to facilitate collection of data output either by the *Mapper* or the *Reducer*. These outputs are nothing but intermediate output of the job. Therefore, they must be parameterized with their types. The *Reporter* facilitates the Map-Reduce application to report progress and update counters and status information. If, however, the combine function is used, it has the same form as the reduce function and the output is fed to the reduce function.

The partition function operates on the intermediate key-value types. It controls the partitioning of the keys of the intermediate map outputs. The key derives the partition using a typical hash function. The total number of partitions is the same as the number of reduce tasks for the job. The partition is determined only by the key ignoring the value.

Input Formats

Hadoop has to accept and process a variety of formats, from text files to databases. A chunk of input, called *input split*, is processed by a single map. Each split is further divided into logical records given to the map to process in key-value pair. In the context of database, the split means reading a range of tuples from an SQL table, as done by the *DBInputFormat* and producing *LongWritables* containing record numbers as keys and *DBWritables* as values.

Output Formats

The output format classes are similar to their corresponding input format classes and work in the reverse direction.

For example, the *TextOutputFormat* is the default output format that writes records as plain text files, whereas key-values any be of any types, and transforms them into a string by invoking the *toString()* method. The key-value character is separated by the tab character, although this can be customized by manipulating the separator property of the text output format.

For binary output, there is *SequenceFileOutputFormat* to write a sequence of binary output to a file. Binary outputs are particularly useful if the output becomes input to a further MapReduce job.

The output formats for relational databases and to HBase are handled by *DBOutputFormat*. It sends the reduced output to a SQL table. For example, the HBase's *TableOutputFormat* enables the MapReduce program to work on the data stored in the HBase table and uses it for writing outputs to the HBase table.

MapReduce Features

- Scalability. Apache Hadoop is a highly scalable framework. ...
- Flexibility. MapReduce programming enables companies to access new sources of data. ...
- Security and Authentication. ...
- Cost-effective solution. ...
- Fast. ...
- Simple model of programming. ...
- Parallel Programming. ...
- Availability and resilient nature.

Search MapReduce real world example on e-commerce transactions data on Internet