Hadoop Enables Big Data Processing

Sheetal Gangakhedkar

Class: Big Data: Tools, Concepts and Deployment

Instructor: Alakh K. Verma

Feb 14, 2017

Introduction

Big Data produced by interactions and sensors has three key characteristics Volume, Variety and Velocity. Extracting, loading, and transforming this kind of data has outgrown the traditional single host storage and processing capabilities. To address these Big Data challenges, Hadoop was created based on Google's papers on Google distributed File System (GFS) and Map-Reduce computational framework for parallel processing. Hadoop, an Apache project, is a platform that provides both distributed storage and computational capabilities over a cluster of machines. (Alex, 2013)

Hadoop is mostly considered suited for handling large data processing batch jobs. Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch-processing methods from Hadoop and stream-processing methods from Apache Storm and Spark.

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Hadoop Architecture

Hadoop is a distributed master-slave architecture that consists of the Hadoop Distributed File System (HDFS) for storage and Map-Reduce for computational capabilities. Traits intrinsic to Hadoop are data partitioning and parallel computation of

large datasets.

The MapReduce master is responsible for organizing where computational work should be scheduled on the slave nodes. The HDFS master is responsible for partitioning the storage across the slave nodes and keeping track of where data is located. Scalability is provided by adding slave nodes for increased needs of storage and processing capabilities.

A Hadoop ecosystem is diverse and grows by the day, as mentioned before Hadoop has two core components, the HDFS and the MapReduce, but the rest of the ecosystem projects are like puzzle pieces that address the gaps or needs of the ecosystem. (Konstantin, Hairong, Sanjay, & Robert, 2010)

- HDFS - Distributed file system (Core)
- MapReduce - Distributed computation framework (Core)
- HBase - Column-oriented table service
- Pig - Dataflow language and parallel execution framework
- Hive - Data warehouse infrastructure
- ZooKeeper - Distributed coordination service
- Oozie - A workflow scheduler expressed as Directed Acyclic Graphs.
- YARN - Enabler for dynamic resource utilization
- Sqoop - Importing data from external sources into HDFS, HBase or Hive
- Flume - Gather and aggregate large amounts of data
- Mahout - Provides implementation of machine learning algorithms
- Apache Kafka - A distributed pub-sub message streaming platform

- Ambari - A Hadoop management web UI backed by its RESTful APIs

Hadoop Core Components

HDFS

The Hadoop Distributed File System (HDFS) is a distributed and highly fault-tolerant distributed file system to run on commodity hardware. HDFS is designed for applications to provide high-throughput access to large data sets. The primary objective of HDFS is to store large data sets reliably even in the presence of failures, by fragmenting data into data blocks and replicating these data blocks across multiple data nodes. HDFS enables streaming access to file system data, prioritizing high-throughput access for batch processing type of applications. HDFS applications need a write-once-read-many access model, this assumption simplifies data coherency issues. HDFS also makes an assumption that is it is much easier and causes less network congestion if the computation is moved closer to where the data is located, rather than migrating large datasets to where the application is running. (HDFS Architecture, 2017)

HDFS is designed with following goals and assumptions:

- Hardware failure
- Streaming data access
- Large data sets
- Simple coherency model
- Moving computation is cheaper than moving data
- Portability across heterogeneous hardware and software platforms

HDFS Architecture

HDFS distributes with replication of data blocks created by segmenting large data files into a master/slave NameNode and DataNodes configuration model. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. Internally, a file is split into one or more data blocks and these blocks are stored in a set of DataNodes.

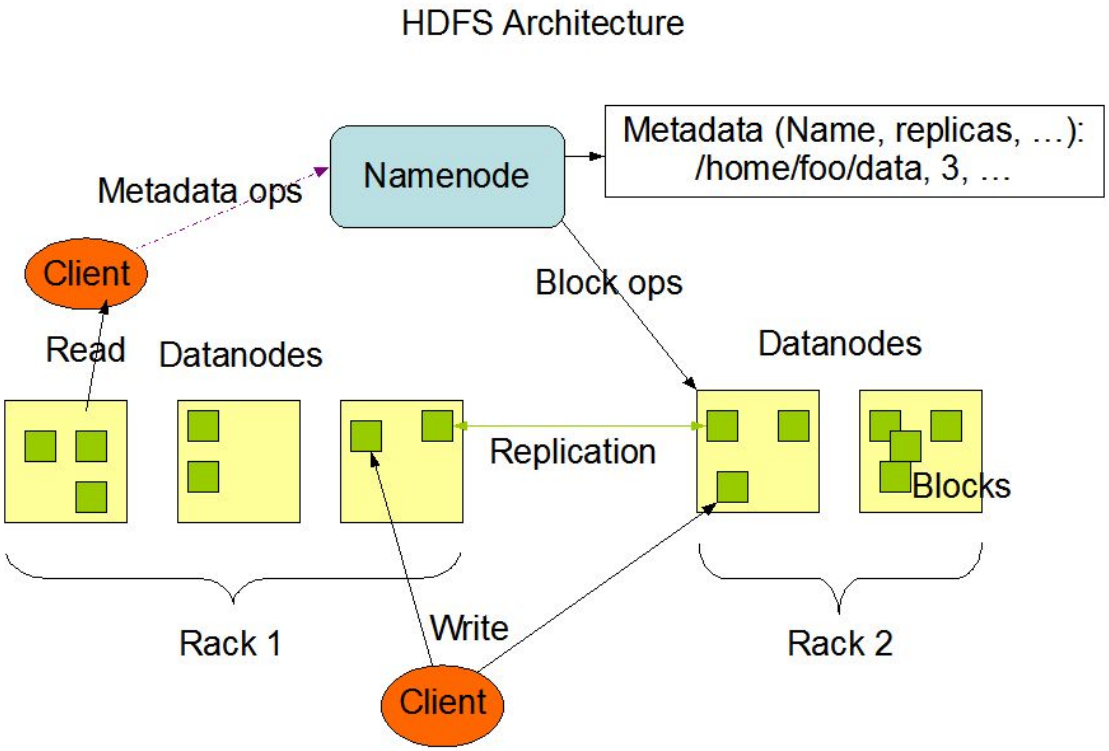

Figure 1: *HDFS Architecture* [figure]. (2017). Retrieved from

http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

A typical deployment has a dedicated and high-availability machine that runs only

the NameNode software. Each of the other machines in the HDFS cluster runs one instance of the DataNode software. The NameNode is the master, arbitrator and repository for all HDFS metadata. The DataNodes are responsible for serving read and write requests from the file system clients. (HDFS Architecture, 2017)

Map-Reduce

MapReduce is a programming model for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all the intermediate values associated with the same intermediate key. Many real-world tasks are expressible in this model. The MapReduce model simplifies parallel processing by abstracting away the complexities involved in working with distributed systems. It hides complexities like parallelization, fault-tolerance, locality optimization, and load balancing. (Jeffrey & Sanjay, 2008)

The□ diagram□ below □illustrates□ a sample MapReduce data processing pipeline.

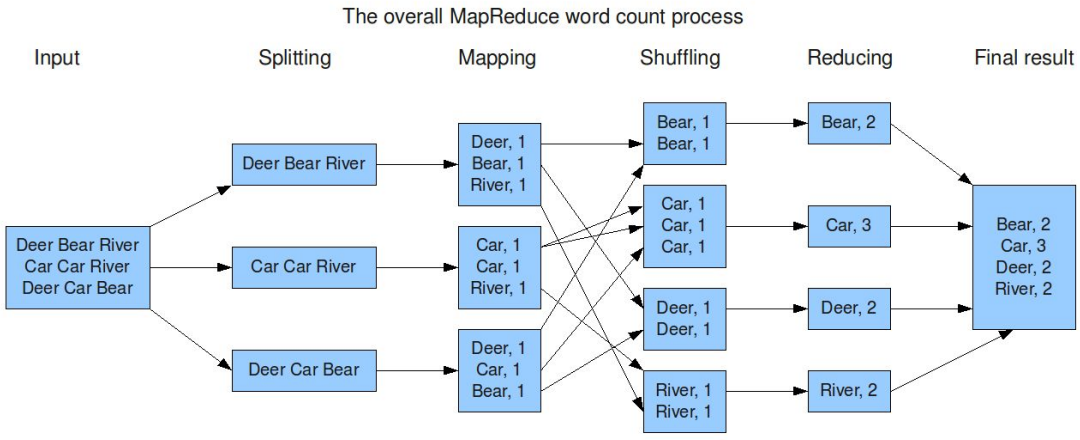

Figure 2: The overall MapReduce word count process, retrieved from http://blog.trifork.com/wp-content/uploads/2009/08/MapReduceWordCountOverview1.png

Apache Spark

Apache Spark is a fast cluster computing framework, heavily relying on in-memory processing of large-scale data. Spark does not use MapReduce as an execution engine; instead it uses its own distributed runtime. However, Spark has many parallels with MapReduce, in terms of both API and runtime. Spark is closely integrated with Hadoop; it can run on YARN and works with Hadoop file formats and storage backends like HDFS. MapReduce is considered mostly for batch Big Data processing applications, and Spark because of its fast computing framework it can be used for iterative algorithms, interactive queries and stream processing. In Spark, applications can be written in multiple languages like Java, Scala, or Python.

At the core of Spark is the notion of a Resilient Distributed Dataset (RDD), which is an immutable collection of objects that are partitioned and distributed across multiple physical nodes of a YARN cluster and that can be operated in parallel. Typically, RDDs are instantiated by loading data from a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat on a YARN cluster. Once an RDD is instantiated, one can apply two types of operations: *Transformations* and *Actions*. “Transformation” operations create new datasets from existing RDD and build out the processing Directed Acyclic Graph (DAG) that can then be applied on the partitioned dataset across the YARN cluster. An “Action” operation executes DAG and returns a value. (Apache Spark Introduction, 2017)

Spark Ecosystem provides Spark Core APIs, and additional libraries for Big Data Analytics and Machine Learning applications, these include:

- **Spark Core** - A general execution engine, providing in-memory computing and referencing datasets in external storage systems as RDDs.
- **Spark SQL** - A new data abstraction called SchemaRDD, which supports structured and semi-structured data.
- **Spark Streaming** - leverages Spark Core's fast scheduling capability to perform streaming analytics.
- **MLlib (Machine Learning Library)** - because of in-memory processing Machine Learning framework is several times faster than disk-based Apache Mahout.
- **GraphX** - distributed graph processing framework.

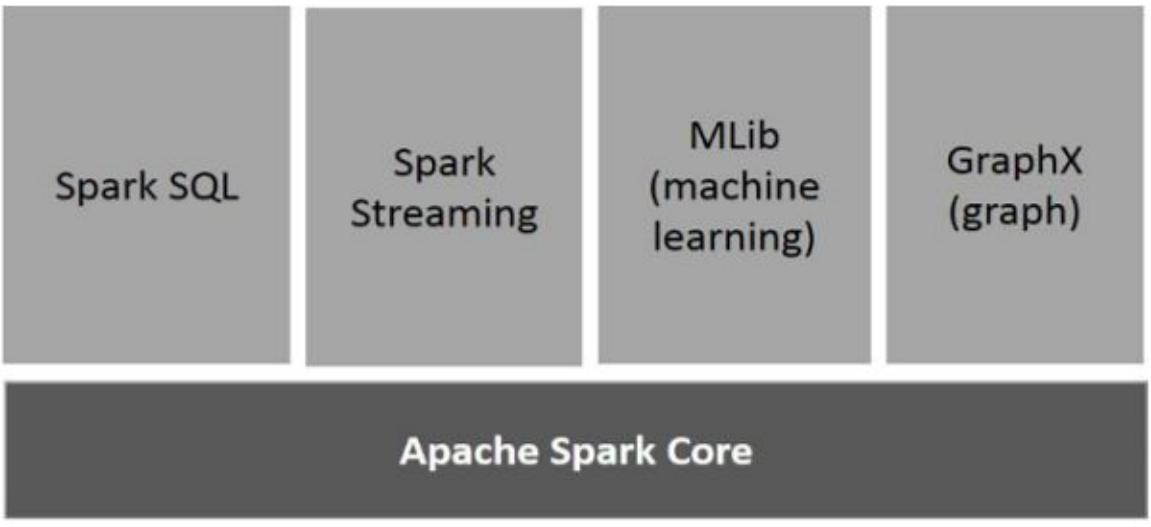

Figure 3: Components of Spark - retrieved from https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

Spark Deployment Models

Spark can run both by itself in standalone mode, or over some existing cluster managers, like YARN/Mesos and Hadoop MapReduce.

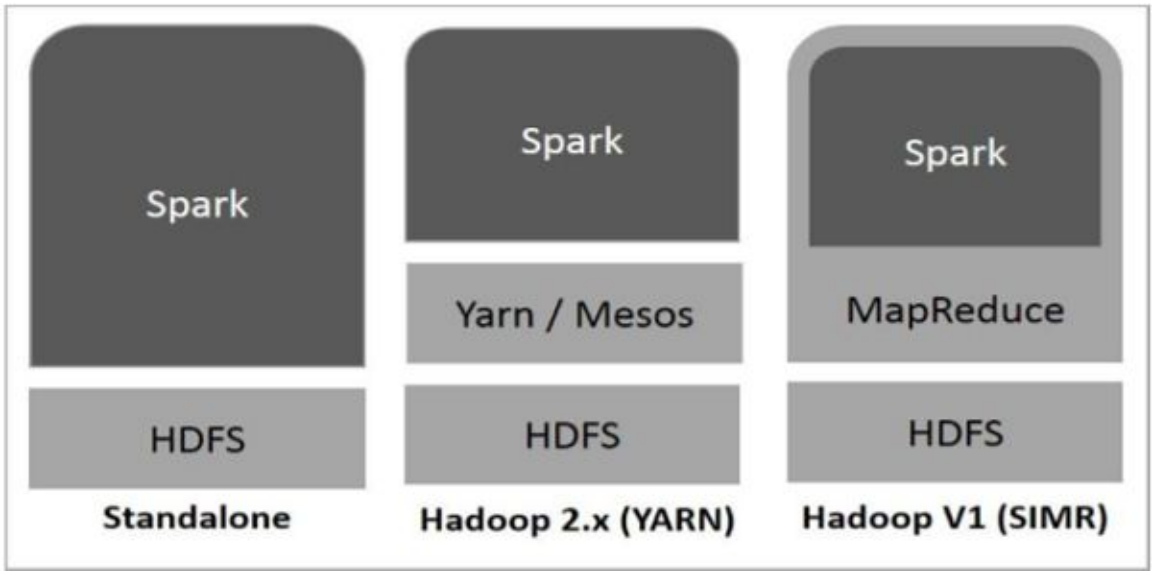

Figure 4: Spark Deployment Models - retrieved from https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

Lambda Architecture

Hadoop on its own is considered mostly for Batch processing applications using HDFS, and Apache Storm or Spark on its own is considered for real-time stream processing applications using in-memory RDDs, to address the user's low-latency requirements Nathan Marz at Twitter designed this generic Lambda architecture addressing the following common requirements for Big Data.

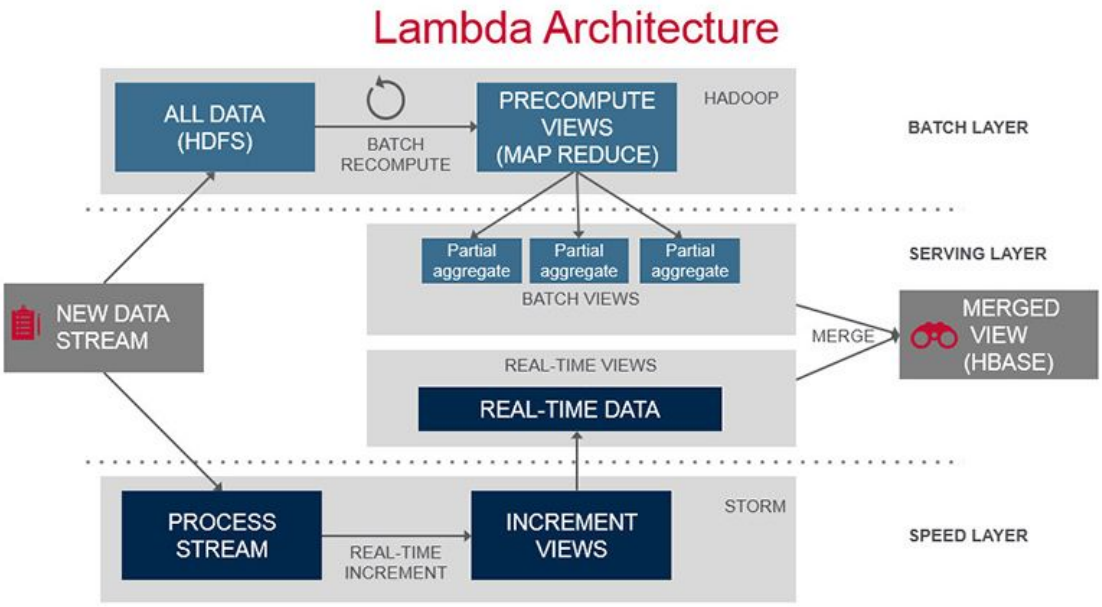

Figure 6: *Overview of the Lambda Architecture*, retrieved from https://www.mapr.com/developercentral/lambda-architecture

The Lambda architecture consists of three layers, the Batch Layer, the Serving Layer and the Speed Layer. Each of these layers can be realized using various big data technologies. For instance, the Batch layer datasets can be in a distributed file system like HDFS, while MapReduce can be used to create batch views that can be fed to the Serving layer. The Serving layer can be implemented using NoSQL technologies such as HBase, while querying can be implemented by technologies such as Apache Drill or Impala. Finally, the Speed layer can be realized with data streaming technologies such as Apache Storm or Spark Streaming. (Lambda Architecture, 2017)

Lambda architecture addresses these common Big Data processing requirements:

- *Fault-tolerance against isolated HW failures and human errors* - the Batch layer provides this by managing the master dataset, an immutable, append-only set of raw data. If a failure occurs, and the original application goes down, a new

instance of the application can pick up the data stream within seconds of where the original application instance dropped off. An added advantage of this architecture is the availability of streaming data for batch as well as the serving layers.

- *Support for low latency querying and updates use-cases* - The Speed layer accommodates all requests that are subject to low latency requirements. Using fast and incremental algorithms, the speed layer deals with recent data only. The Serving layer indexes the batch views so that they can be queried in ad-hoc with low latency.
- *Linear scale-out capabilities*, meaning that throwing more machines at the problem should help with getting the job done.
- *Extensibility* so that the system is manageable and can accommodate newer features easily. The applications can use the growing feature set in the Hadoop or Apache Storm or Spark ecosystems.

Lambda architecture has some drawbacks (Jay, 2014):

- Difficult to maintain code that needs to produce the same result in two complex distributed systems. Inevitably, code ends up being specifically engineered toward the framework it runs on.
- Increasingly we need to build complex, low-latency processing systems. Combining a scalable high-latency batch system that can process historical data and a low-latency stream processing system that can't reprocess results, as defined in the Lambda architecture is a temporary solution, driven by the current

limitation of off-the-shelf tools.

Jay Kreps in his July 2, 2014 "*Questioning the Lambda Architecture"* online article advises,

> These days, my advice is to use a batch processing framework like MapReduce if you aren't latency sensitive, and use a stream processing framework if you are, but not to try to do both at the same time unless you absolutely must.

Conclusion

Systems that support large volumes of both structured and unstructured data will continue to rise. The market will demand platforms that can perform Big Data analysis with increasingly lower latency and higher complexity of analysis. We explored the HDFS/MapReduce Hadoop system, then a low-latency/interactive in-memory Apache Spark ecosystem, and then analyzed the Lambda architecture, which combines the two architectures. Big Data processing tools and technologies are growing leaps-and-bounds, and companies are trying to simplify as well as try to solve the growing and complex requirements of Data Analytics.

References

[1] Alex Holmes. 2012. *Hadoop in Practice*. Manning Publications Co., Greenwich, CT, USA

[2] HDFS Architecture. (2017). Retrieved February 16, 2017, from http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,

Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. *Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12). USENIX Association, Berkeley, CA, USA, 2-2

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. *MapReduce: simplified data processing on large clusters*. Commun. ACM 51, 1 (January 2008), 107-113. DOI=http://dx.doi.org/10.1145/1327452.13274924

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. *The Google file system*. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 29-43. DOI=http://dx.doi.org/10.1145/945445.945450

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. *The Hadoop Distributed File System*. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10). IEEE Computer Society, Washington, DC, USA, 1-10. DOI=http://dx.doi.org/10.1109/MSST.2010.5496972

[7] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. *Spark: cluster computing with working sets*. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, Berkeley, CA, USA, 10-10.

[8] *Apache Spark Introduction*. (2017). Retrieved February 16, 2017, from

https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

[9] Michael Hausenblas. (2017) *Lambda Architecture*. Retrieved February 16, 2017, from https://www.mapr.com/developercentral/lambda-architecture

[10] Jay Kreps. (July 2, 2014). *Questioning the Lambda Architecture*. Retrieved Feburary 19, 2017 from https://www.oreilly.com/ideas/questioning-the-lambda-architecture