**Hadoop 101: The Most Important Terms, Explained**

March 27, 2014 by **Victoria Garment**

<http://www.plottingsuccess.com/hadoop-101-important-terms-explained-0314/>

In the world of business intelligence (BI) and big data, Apache Hadoop receives quite a bit of attention and buzz. Despite this extensive coverage, however, Hadoop remains a gray area for many. Unless you work with Hadoop daily, you may have a vague idea of what it is and what it does, but beyond that, you’re likely in over your head.

To eliminate the confusion and help professionals understand Hadoop once and for all, we enlisted the help of three experts to create this plain English guide. Here, we explain exactly what Hadoop is, how it works and the most important terms associated with it that you need to know.

Our experts are:

* **Jesse Anderson**, curriculum developer at [Cloudera](http://www.cloudera.com/content/cloudera/en/home.html), a company that provides Hadoop-based software, support and training to businesses.
* **Alejandro Caceres**, founder of [Hyperion Gray](http://www.hyperiongray.com/), a small research and development company that creates open-source software.
* **Elliot Cordo**, chief architect at [Caserta Concepts](http://www.casertaconcepts.com/), a New York-based innovative technology and consulting firm that specializes in big data analytics, business intelligence and data warehousing solutions.

**What Is Hadoop?**

At the most basic level, Hadoop is an open-source software platform designed to store and process quantities of data that are too large for just one particular device or server. Hadoop’s strength lies in its ability to scale across thousands of commodity servers that don’t share memory or disk space.

Hadoop delegates tasks across these servers (called “worker nodes” or “slave nodes”), essentially harnessing the power of each device and running them together simultaneously. This is what allows massive amounts of data to be analyzed: splitting the tasks across different locations in this manner allows bigger jobs to be completed faster.

Hadoop can be thought of as an ecosystem—it’s comprised of many different components that all work together to create a single platform. There are two key functional components within this ecosystem: The storage of data (Hadoop Distributed File System, or HDFS) and the framework for running parallel computations on this data (MapReduce). Let’s take a closer look at each.

**Hadoop Distributed File System (HDFS)**

HDFS is the “secret sauce” that enables Hadoop to store huge files. It’s a scalable file system that distributes and stores data across all machines in a Hadoop cluster (a group of servers). Each HDFS cluster contains the following:

* **NameNode:** Runs on a “master node” that tracks and directs the storage of the cluster.
* **DataNode:** Runs on “slave nodes,” which make up the majority of the machines within a cluster. The NameNode instructs data files to be split into blocks, each of which are replicated three times and stored on machines across the cluster. These replicas ensure the entire system won’t go down if one server fails or is taken offline—known as “fault tolerance.”
* **Client machine:** neither a NameNode or a DataNode, Client machines have Hadoop installed on them. They’re responsible for loading data into the cluster, submitting MapReduce jobs and viewing the results of the job once complete.

**MapReduce**

MapReduce is the system used to efficiently process the large amount of data Hadoop stores in HDFS. Originally created by Google, its strength lies in the ability to divide a single large data processing job into smaller tasks. All MapReduce jobs are written in Java, but other languages can be used via the Hadoop Streaming API, which is a utility that comes with Hadoop.

Once the tasks have been created, they’re spread across multiple nodes and run simultaneously (the “map” step). The “reduce” phase combines the results together.

Imagine, for example, that an entire MapReduce job is the equivalent of building a house. Each job is broken down into individual tasks (e.g. lay the foundation, put up drywall) and assigned to various workers, or “mappers” and “reducers.” Completing each task results in a single, combined output: the house is complete.

This delegation of tasks is handled by two “daemons,” the JobTracker and TaskTracker. The technical definition of a daemon is “a process that is long-lived.” In our house example, a daemon can be thought of as a foreman: the jobs may change (new houses must be built), workers will come and go, but the foreman is always there to oversee the job and delegate tasks.

* **JobTracker:** The JobTracker oversees how MapReduce jobs are split up into tasks and divided among nodes within the cluster.
* **TaskTracker:** The TaskTracker accepts tasks from the JobTracker, performs the work and alerts the JobTracker once it’s done. TaskTrackers and DataNodes are located on the same nodes to improve performance.

A good way to understand how MapReduce works is with playing cards. *(Note: this example was provided by Jesse Anderson. The full video can be found* [*here*](https://www.youtube.com/watch?v=bcjSe0xCHbE)*.)*

Each deck of cards contains the following:

* *Four suits:* diamonds, clubs, hearts and spades.
* *Numeric cards:* 2, 3, 4, 5, 6, 7, 8, 9, 10.
* *Non-numeric cards:* ace, king, queen, jack and the jokers.

To understand how MapReduce works, imagine that your goal is to add up the card numbers for a single suit. To do this, you must sort through an entire deck of cards, one by one. As you do so, you will “map” them by separating the cards into piles according to suit.

In this example, non-numeric cards will represent “bad data,” or data we want to exclude from the results of this particular MapReduce job. As you go through the deck, you ignore the “bad data” by setting it aside from the data you do want to include.

Since MapReduce works by splitting up jobs into tasks performed on multiple nodes, we’ll illustrate this by using two pieces of paper to represent the two DataNodes this example job will be performed on.

To begin, we split the deck in half, “mapping” one half on each node by separating the numeric cards by suit and setting the non-numeric cards aside:



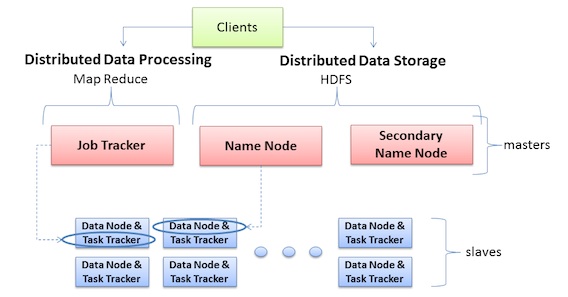
This mapping process results in four separate piles on each node—one for each suit. You’ll also have one “discard” pile containing the non-numeric cards you’re excluding from this job.

Once all the data has been mapped out, the next stage is the “reduce” stage. To simulate this, we’re going to combine each pile of hearts, diamonds, spades and clubs to create two piles on each node:



Each node has thus mapped the data (or cards) independently to create eight piles of cards, then reduced this data down to four piles of cards. Each node can now process two entire suit piles at once to create the single output we’re looking for in this example: the total sum of the numbers in each suit pile.

Now that you understand how MapReduce works, let’s look at a visual representation of what the entire Hadoop ecosystem looks like:



*Image courtesy of* [*Brad Hedlund*](http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/)

**Data locality:** An important concept with HDFS and MapReduce, data locality can best be described as “bringing the compute to the data.” In other words, whenever you use a MapReduce program on a particular part of HDFS data, you always want to run that program on the node, or machine, that actually stores this data in HDFS. Doing so allows processes to be run much faster, since it prevents you from having to move large amounts of data around.

When a MapReduce job is submitted, part of what the JobTracker does is look to see which machines the blocks required for the task are located on. This is why, when the NameNode splits data files into blocks, each one is replicated three times: the first is stored on the same machine as the block, while the second and third are each stored on separate machines.

Storing the data across three machines thus gives you a much higher chance of achieving data locality, since it’s likely that at least one of the machines will be freed up enough to process the data stored at that particular location.

**Yet Another Resource Negotiator (YARN)**

YARN is an updated way of handling the delegation of resources for MapReduce jobs. It takes the place of the JobTracker and TaskTracker. In our house example, if JobTracker and TaskTracker can be thought of as the foreman, YARN is a foreman with an MBA—it’s a more advanced way of carrying out MapReduce jobs.

It also gives you added abilities, such as the ability to work with frameworks other than MapReduce and to translate jobs developed in languages other than Java.

**HBase**

HBase is a columnar database management system that is built on top of Hadoop and runs on HDFS. Like MapReduce, HBase applications are written in Java, as well as other languages via their Thrift database, which is a framework that allows cross-language services development. The key difference between MapReduce and HBase is that HBase is intended to work with random workloads.

For example, if you have regular files that need to be processed, MapReduce works just fine. But if you have a table that is a petabyte in size and you need to process a single row from a random location within this table, you would use HBase. Another benefit of HBase is the extremely low latency, or time delay, it provides.

It’s important to note, however, that HBase and MapReduce are not mutually exclusive. In fact, you can often run them together—MapReduce can run against an HBase table or a file, for example.

**Hive**

MapReduce jobs are often written in Java. But not everyone using Hadoop knows Java—the preferred syntax is SQL, which is essentially the “lingua franca” between all programming languages in the BI/big data space.

Hive allows users who aren’t familiar with programming to access and analyze big data in a less technical way, using a SQL-like syntax called Hive Query Language (HiveQL). HiveQL is used to create programs that run just like MapReduce would on a cluster.

In a very general sense, Hive is used for complex, long-running tasks and analyses on large sets of data, e.g. analyzing the performance of every store within a particular region for a chain retailer.

**Impala**

Like Hive, Impala also uses SQL syntax instead of Java to access data. What distinguishes Hive and Impala is speed: While a query using Hive may take minutes, hours or longer, a query using Impala usually take seconds (or less).

Impala is used for analyses that you want to run and return quickly on a small subset of your data, e.g. analyzing company finances for a daily or weekly report. Since Impala is meant to be used as an analytic tool on top of prepared, more structured data, it’s not ideal if you’re in the process of data preparation and complex data manipulation, e.g. ingesting raw machine data from log files.

A good way to think about Hive and Impala is to compare them to a screwdriver and drill bit: both can do the same—or similar—jobs, but the drill (Impala) is much faster.

**Apache Pig**

Like Hive and Impala, Pig is a high-level platform used for creating MapReduce programs more easily. The programming language Pig uses is called Pig Latin, and it allows you to extract, transform and load (ETL) data at a very high level—meaning something that would require several hundred lines of Java code can be expressed in, say, 10 lines of Pig.

While Hive and Impala require data to be more structured in order to be analyzed, Pig allows you to work with unstructured data. In other words, while Hive and Impala are essentially query engines used for more straightforward analysis, Pig’s ETL capability means it can perform “grunt work” on unstructured data, cleaning it up and organizing it so that queries can be run against it.

**Hadoop Common**

Usually only referred to by programmers, Hadoop Common is a common utilities library that contains code to support some of the other modules within the Hadoop ecosystem. When Hive and HBase want to access HDFS, for example, they do so using JARs (Java archives), which are libraries of Java code stored in Hadoop Common.

**Apache Spark**

While not yet part of the Hadoop ecosystem, Apache Spark is frequently mentioned along with Hadoop, so we’ll take a moment to touch on it here. Spark is an alternative way to perform the type of batch-oriented processing that MapReduce does. (Batch-oriented means that it will take a certain amount of time for a result to be returned, as opposed to returning it in real-time.)

While MapReduce jobs use data that have been replicated and stored on-disk within a cluster, Spark allows you to leverage the memory space on servers, performing in-memory computing. This allows for real-time data processing that is up to 100 times faster than MapReduce in some instances.