Chapter 2
Getting Started with Hadoop

Apache Hadoop is a software framework that allows distributed processing of large datasets across clusters of computers using simple programming constructs/models. It is designed to scale-up from a single server to thousands of nodes. It is designed to detect failures at the application level rather than rely on hardware for high-availability thereby delivering a highly available service on top of cluster of commodity hardware nodes each of which is prone to failures [2]. While Hadoop can be run on a single machine the true power of Hadoop is realized in its ability to scale-up to thousands of computers, each with several processor cores. It also distributes large amounts of work across the clusters efficiently [1].

The lower end of Hadoop-scale is probably in hundreds of gigabytes, as it was designed to handle web-scale of the order of terabytes to petabytes. At this scale the dataset will not even fit a single computer's hard drive, much less in memory. Hadoop's distributed file system breaks the data into chunks and distributes them across several computers to hold. The processes are computed in parallel on all these chunks, thus obtaining the results with as much efficiency as possible.

The Internet age has passed and we are into the data age now. The amount of data stored electronically cannot be measured easily; IDC estimates put the total size of the digital universe at 0.18 Zetabytes in 2006 and it is expected to grow tenfold by 2011 to 1.8 Zeta-bytes [9]. A Zetabyte is 10^{21} bytes, or equivalently 1000 Exabytes, 1,000,000 Petabytes or 1bn Terabytes. This is roughly equivalent to one disk drive for every person in the world [10]. This flood of data comes from many sources. Consider the following:

- The New York Stock Exchange generates about one terabyte of trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage. Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.
- The Internet Archive stores around 2 petabytes of data, and is growing at a rate of 20 terabytes per month.

K.G. Srinivasa and A.K. Muppalla, *Guide to High Performance Distributed Computing*, Computer Communications and Networks,
DOI 10.1007/978-3-319-13497-0_2

- The Large Hadron Collider near Geneva, Switzerland, produces around 15 petabytes of data per year.

2.1 A Brief History of Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, a widely used text search library. The Apache Nutch project, an open source web search engine, had a significant contribution to building Hadoop [1].

Hadoop is not an acronym; it is a made-up name. The project creator, Doug Cutting, explains how the name came about:

> The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such.

It is ambitious to build a web search engine from scratch as it is not only challenging to build a software required to crawl and index websites, but also a challenge to run without a dedicated operations team as there are so many moving parts. It is estimated that a 1 billion page index would cost around $500,000 to build and monthly $30,000 for maintenance [4]. Nevertheless, this goal is worth pursuing as Search engine algorithms are opened to the world for review and improvement.

The Nutch project was started in 2002, with the crawler and search system being quickly developed. However, they soon realized that their system would not scale to a billion pages. Timely publication from Google in 2003, the architecture of Google FileSystem, called the GFS came in very handy [5]. The GFS or something like that was enough to solve their storage needs for the very large files generated as part of the web crawl and indexing process. The GFS particularly frees up the time spent on maintaining the storage nodes. This effort gave way to the Nutch Distributed File System (NDFS) .

Google produced another paper in 2004 that would introduce **MapReduce** to the world. By early 2005 the Nutch developers had a working MapReduce implementation in Nutch and by the middle of that year most of the Nutch Algorithms were ported to MapReduce and NDFS.

NDFS and MapReduce implementation in Nutch found applications in areas beyond the scope of Nutch; in Feb 2006 they were moved out of Nutch to form their own independent subproject called Hadoop. Around the same time Doug Cutting joined Yahoo! which gave him access to a dedicated team and resources to turn Hadoop into a system that ran at web-scale. This ability of Hadoop was announced that its production search index was generated by the 10,000 node Hadoop cluster [6].

In January 2008, Hadoop was promoted to a top level project at Apache, confirming its success and its diverse active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times.

The capability of Hadoop was demonstrated and publicly put at the epitome of the distributed computing sphere when The New York Times used Amazon's EC2 compute cloud to crunch through four terabytes of scanned archives from the paper converting them into PDFs for the Web [7]. The project came at the right time with great publicity toward Hadoop and the cloud. It would have been impossible to try this project if not for the popular pay-by-the-hour cloud model from Amazon. The NYT used a large number of machines, about a 100 and Hadoop's easy-to-use parallel programming model to process the task in 24 hours.

Hadoop's successes did not stop here, it went on to break the world record to become the fastest system to sort a terabyte of data in April 2008. It took 209 seconds to sort a terabyte of data on a 910 node cluster, beating the previous year's winner of 297 seconds. It did not end here, later that year Google reported that its MapReduce implementation sorted one terabyte in 68 seconds [8]. Later, Yahoo! reported to have broken Google's record by sorting one terabyte in 62 seconds.

2.2 Hadoop Ecosystem

Hadoop is a generic processing framework designed to execute queries and other batch read operations on massive datasets that can scale from tens of terabytes to petabytes in size. HDFS and MapReduce together provide a reliable, fault-tolerant software framework for processing vast amounts of data in parallel on large clusters of commodity hardware (potentially scaling to thousands of nodes).

Hadoop meets the needs of many organizations for flexible data analysis capabilities with an unmatched price-performance curve. The flexibility in data analysis feature applies to data in a variety of formats, from unstructured data, such as raw text, to semi-structured data, such as logs, to structured data with a fixed schema.

In environments where massive server farms are used to collect data from a variety of sources, Hadoop is able to process parallel queries as background batch jobs on the same server farm. Thus, the requirement for an additional hardware to process data from a traditional database system is eliminated (assuming such a system can scale to the required size). The effort and time required to load data into another system is also reduced since it can be processed directly within Hadoop. This overhead becomes impractical in very large datasets [14].

The Hadoop ecosystem includes other tools to address particular needs:

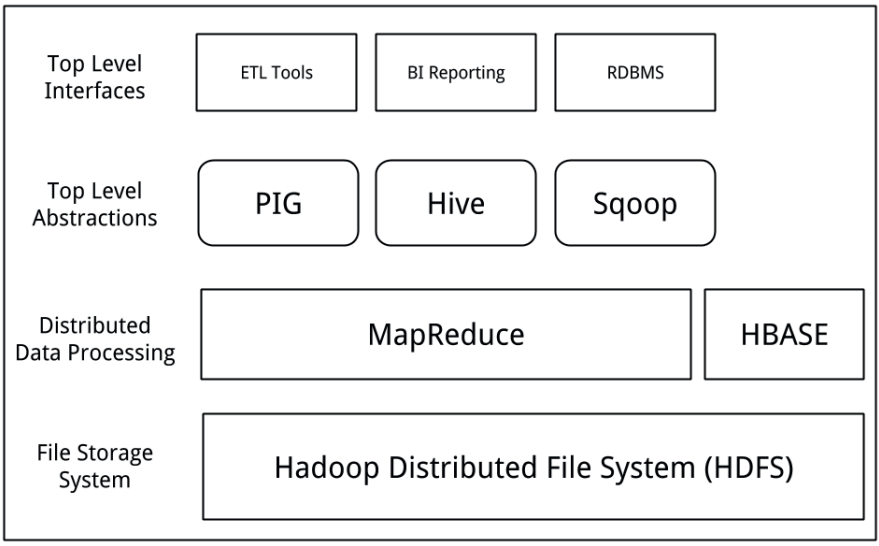


Fig. 2.1: Hadoop Ecosystem [14]

Common: A set of components and interfaces for distributed filesystems and general I/O (serialization, Java RPC, persistent data structures).

Avro: A serialization system for efficient, cross-language RPC and persistent data storage.

MapReduce: A distributed data processing model and execution environment that runs on large clusters of commodity machines.

HDFS: A distributed filesystem that runs on large clusters of commodity machines.

Pig: A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters [6].

Hive: A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data [7].

HBase: A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads) [18].

ZooKeeper: A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications [19].

Sqoop: A tool for efficiently moving data between relational databases and HDFS.

Cascading: MapReduce is very powerful as a general-purpose computing framework, but writing applications in the Hadoop Java API for MapReduce is daunting, due to the low-level abstractions of the API, the verbosity of Java, and the relative inflexibility of the MapReduce for expressing many common algorithms. Cascading is the most popular high-level Java API that hides many of the complexities of MapReduce programming behind more intuitive pipes and data flow abstractions.

Twitter Scalding: Cascading is known to provide close abstraction to the daunting Java API using pipes and custom data flows. It still suffers from limitations and verbosity of Java. Scalding is Scala API on top of cascading that indents to remove most of the boilerplate Java code and provides concise implementations of common data analytics and manipulation functions similar to SQL and Pig. Scalding also provides algebra and matrix models, which are useful in implementing machine learning and other linear algebra-dependent algorithms.

Cascalog: Cascalog is similar to Scalding in the way it hides the limitations of Java behind a powerful Clojure API for cascading. Cascalog includes logic programming constructs inspired by Datalog. The name is derived from Cascading + Datalog.

Impala: It is a scalable parallel database technology to Hadoop, which can be used to launch SQL queries on the data stored in HDFS and Apache HBase without any data movement or transformation. It is a massively parallel processing engine that runs natively on Hadoop.

Apache BigTop: It was originally a part of the Cloudera's CDH distribution, which is used to test the Hadoop ecosystem.

Apache Drill: It is an open-source version of Google Drell . It is used for interactive analysis on large-scale datasets. The primary goals of Apache Drill are real-time querying of large datasets and scaling to clusters bigger than 10,000 nodes. It is designed to support nested data, but also supports other data schemes like Avro and JSON. The primary language, *DrQL*, is SQL like [20].

Apache Flume: It is responsible data transfer between "source" and "sink", which can be scheduled or triggered upon an event. It is used to harvest, aggregate, and move large amounts of data in and out of Hadoop. Flume allows different data formats for sources, Avro, files, and sinks, HDFS and HBase. It also has a querying

engine so that the user can transform any data before it is moved between sources and sinks.

Apache Mahout: it is a collection of scalable data mining and machine learning algorithms implemented in Java. Four main groups of algorithms are:

- Recommendations, a.k.a. collective filtering
- Classification, a.k.a categorization
- Clustering
- Frequent itemset mining, a.k.a parallel frequent pattern mining

It is not merely a collection of algorithms as many machine learning algorithms are non-scalable, the algorithms in Mahout are written to be distributed in nature and use the MapReduce paradigm for execution.

Oozie: it is used to manage and coordinate jobs executed on Hadoop.

2.3 Hadoop Distributed File System

The distributed file system in Hadoop is designed to run on commodity hardware. Although it has many similarities with other distributed file systems, the differences are significant. It is highly fault-tolerant and is designed to run on low-cost hardware. It also provides high-throughput to stored data, hence can be used to store and process large datasets. To enable this streaming of data it relaxes some POSIX standards. HDFS was originally built for the Apache Nutch project and later forked in to an individual project under Apache [21].

HDFS by design is able to provide reliable storage to large datasets, allowing high-bandwidth data streaming to user applications. By distributing the storage and computation across several servers, the resource can scale up and down with demand while remaining economical.

HDFS is different from other distributed file systems in the sense that it uses a write-once-read-many model that relaxes concurrency requirements, provides simple data coherency, and enables high-throughput data access [22]. HDFS prides on the principle and proves to be more efficient when the processing is done near the data rather than moving the data to the applications space. The data writes are restricted to one writer at a time. The bytes are appended to the end of the stream and are stored in the order written. HDFS has many notable goals:

- Ensuring fault tolerance by detecting faults and applying quick recovery methods.

- MapReduce streaming for data access.
- Simple and robust coherency model.
- Processing is moved to the data, rather than data to processing.
- Support heterogeneous commodity hardware and operating systems.
- Scalability in storing and processing large amounts of data.
- Distributing data and processing across clusters economically.
- Reliability by replicating data across the nodes and redeploying processing in the event of failures.

2.3.1 Characteristics of HDFS

Hardware Failure: Hardware failure is fairly common in clusters. A Hadoop cluster consists of thousands of machines, each of which stores a block of data. HDFS consists of a huge number of components and with that there is a good chance of failure among them at any point of time. The detection of these failures and the ability to quickly recover is part of the core architecture.

Streaming Data Access: Applications that run on the Hadoop HDFS need access to streaming data. These applications cannot be run on general-purpose file systems. HDFS is designed to enable large-scale batch processing, which is enabled by the high-throughput data access. Several POSIX requirements are relaxed to enable these special needs of high throughput rates.

Large Data Sets: The HDFS-based applications feed on large datasets. A typical file size is in the range of high gigabytes to low terabytes. It should provide high data bandwidth and support millions of files across hundreds of nodes in a single cluster.

Simple Coherency Model: The write-once-read-many access model of files enables high throughput data access as the data once written need not be changed, thus simplifying data coherency issues. A MapReduce-based application takes advantage of this model.

Moving compute instead of data: Any computation is efficient if it executes near the data because it avoids the network transfer bottleneck. Migrating the computation closer to the data is a cornerstone of HDFS-based programming. HDFS provides all the necessary application interfaces to move the computation close to the data prior to execution.

Heterogeneous hardware and software portability: HDFS is designed to run on commodity hardware, which hosts multiple platforms. This features helps widespread adoption of this platform for large-scale computations.

Drawbacks:

Low-latency data access: The high-throughput data access comes at the cost of latency. Latency sensitive applications are not suitable for HDFS. HBase has shown promise in handling low-latency applications along with large-scale data access.

Lots of small files: The metadata of the file system is stored in the namenode's memory(master node). The limit to the number of files in the file-system is dependent on the namenode memory. Typically, each file, directory, and block takes about 150 bytes. For example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Storing millions of files seems possible, but hardware is incapable of accommodating billions of files [24].

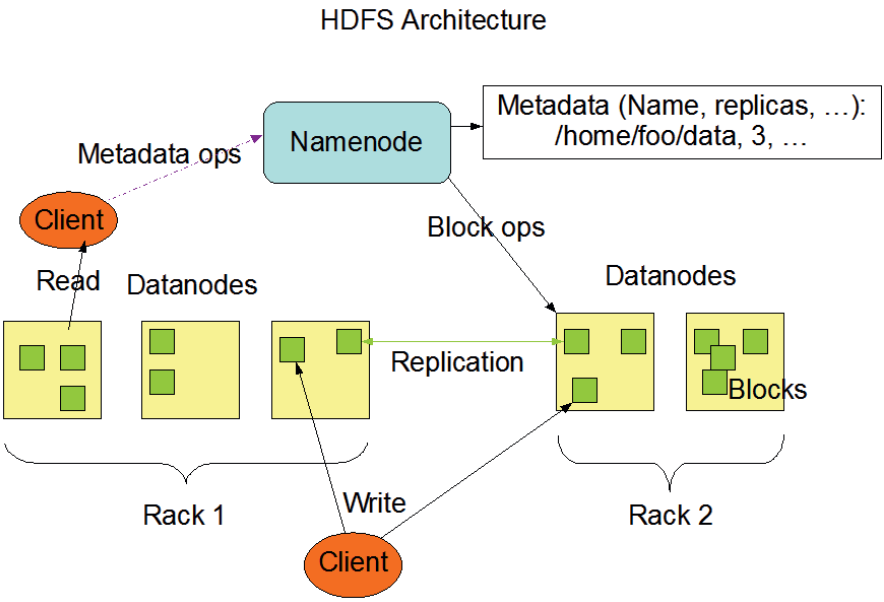


Fig. 2.2: HDFS Architecture [1]

2.3.2 Namenode and Datanode

The HDFS architecture is built in tandem with the popular master/slave architecture. An HDFS cluster consists of a master server that manages the file-system namespace and regulates files access, called the *Namenode*. Analogous to slaves, there are a number of *Datanodes*. These are typically one per cluster node and manage the data stored in the nodes. The HDFS file-system exists independently of the host file-system and allows data to be stored in its own namespace . The Namenode allows typical file-system operations like opening, closing, and renaming files and directories. It also maintains data block and Datanode mapping information. The Datanodes handle the read and write requests. Upon Namenode's instructions the Datanodes perform block creation, deletion, and replications operations. HDFS architecture is given in Figure 2.2

Namenode and Datanodes are software services provided by HDFS that are designed to run on heterogeneous commodity machines. These applications typically run on Unix/Linux-based operating systems. Java programming language is used to build these services. Any machine that supports the Java runtime environment can run Namenode and Datanode services. Given the highly portable nature of Java programming language HDFS can be deployed on wide range of machines. A typical cluster installation has one dedicated machine that acts as a master running the Namenode service. Other machines in the cluster run one instance of Datanode service per node. Although you can run multiple Datanodes on one machine, this practice is rare in real-world deployments.

A single instance of Namenode/master simplifies the architecture of the system. It acts as an arbitrator and a repository to all the HDFS meta-data. The system is designed such that there is data flow through the Namenode. Figure 2.3 shows how the Hadoop ecosystem interacts together.

2.3.3 File System

The file organization in HDFS is similar to the traditional hierarchical type. A user or an application can create and store files in directories. The namespace hierarchy is similar to other file-systems in the sense that one can create, remove, and move files from one directory to another, or even rename a file. HDFS does not support hard-links or soft-links.

Any changes to the file-system namespace is recorded by the Namenode. An application can specify the replication factor of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor.

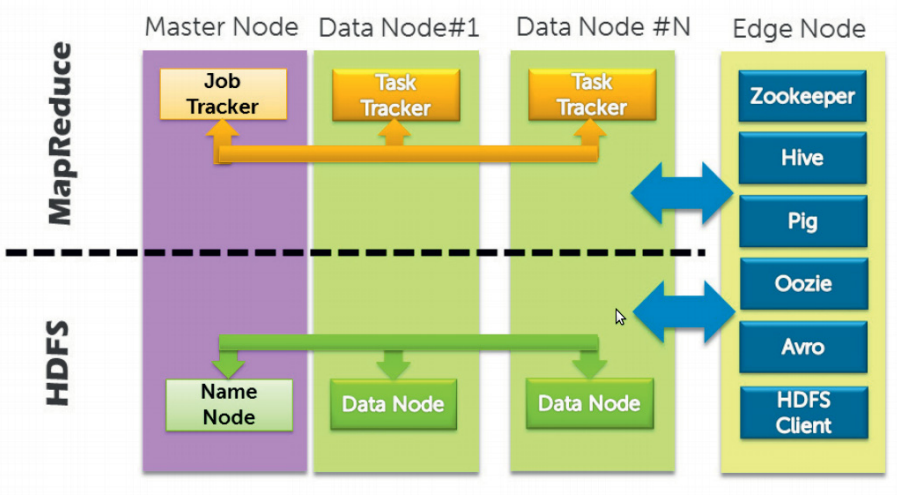


Fig. 2.3: Interaction between HDFS and MapReduce [1]

2.3.4 Data Replication

HDFS provides a reliable storage mechanism even though the cluster spans thousands of machines. Each file is stored as a sequence of blocks, where each block except the last one is of the same size. Fault tolerance is ensured by replicating the file blocks. The *block size* and *replication factor* is configurable for each file either by the user or an application. The replications factor can be set at file creation and can be modified later. Files in HDFS are write-once and strictly adhere to a single writer at a time property (Figure 2.4).

Namenode, acting as the master, takes all the decisions regarding data block replication. It receives a heartbeat, check Figure 2.5 on how it works, and block report from the Datanodes in the cluster. Heartbeat implies that the Datanode is functional and the block report provides a list of all the blocks in a Datanode.

Replication is vital to HDFS reliability and performance is improved by optimizing the placement of these replicas. This optimized placement contributes significantly to the performance and distinguishes itself from the rest of the file-systems. This feature requires much tuning and experience to get it right. Using a rack-aware replication policy improves data availability, reliability, and efficient network bandwidth utilization. This policy is the first of its kind and has seen much attention with better and sophisticated policies to follow.

Typically large HDFS instances span across multiple racks. Communication between these racks go through switches. Network bandwidth between machines in different racks is less than machines within the same rack.

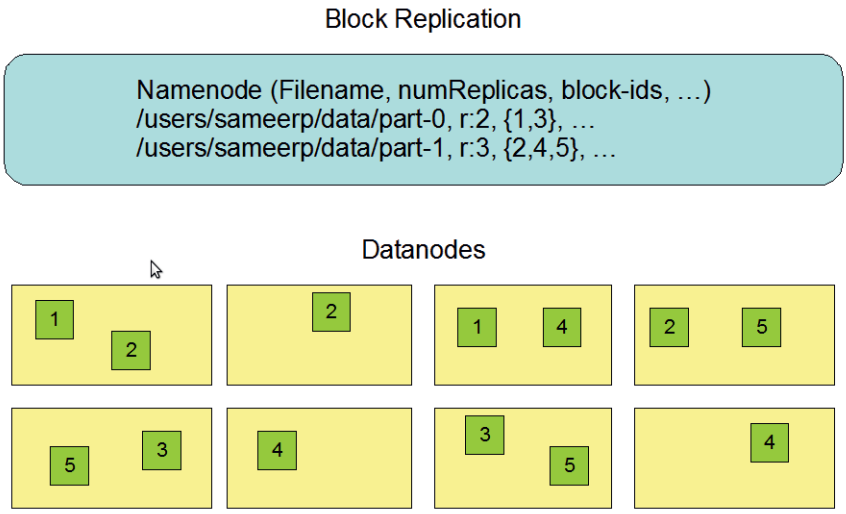


Fig. 2.4: Data Replication in Hadoop [22]

The Namenode is aware of the rack-ids each Datanode belongs to. A simple policy can be implemented by placing unique replicas in individual racks; in this way even if an entire rack fails, although unlikely, the replica on another rack is still available. However, this policy can be costly as the number of writes between racks increases.

When the replication factor is, say 3, the placement policy in HDFS is put a replica in one node per rack. This policy cuts the inter-rack write, which improves performance. The chance of rack failure is far less than node failure. This policy significantly reduces the network bandwidth used when reading data as a replica is placed in only two unique racks instead of three. One-third of replicas are on one node and one-third in another rack, while another third is evenly distributed across remaining racks. This policy improves performance along with data reliability and read performance.

Replica Selection: Selecting a replica that is closer to the reader significantly reduces the global bandwidth consumption and read latency. If a replica is present on the same rack then it is chosen instead of another rack, similarly if the replica spans data centers then the local data center hosting the replica is chosen.

Safe-mode: When HDFS services are started, the Namenode enters a safe-mode. The Namenode receives heartbeats and block reports from the Datanodes. Each block has a specified number of replicas and Namenode checks if these replicas are present. It also checks with Datanodes through the heartbeats. Once a good per-

centage of blocks are present the Namenode exits the safe-mode (takes about 30s). It completes the remaining replication in other Datanodes.

2.3.5 Communication

The TCP/IP protocols are foundations to all the communication protocols built in HDFS. Remote Procedure Calls (RPCs) are designed around the client protocols and the Datanode protocols. The Namenode does not initiate any procedure calls, it only responds to the request from the clients and the Datanodes.

Robustness: The fault-tolerant nature of HDFS ensures data reliability. The three common type of failures are Namenode, Datanode, and network partitions.

Data Disk Failure, Heartbeats, and Re-replication: Namenode receives periodic heartbeats from the Datanodes. A network partition, failure of a switch, can cause all the Datanodes connected via that network to be invisible to the Namenode. Namenode used heartbeats to detect the condition of nodes in the cluster, it marks them dead if there are no recent heartbeats and does not forward any I/O requests. Any data part of the dead Datanode is not available and the Namenode keeps track of these blocks and triggers replications whenever necessary. There are several reasons to re-replicate a data block: Datanode failure, corrupted data block, storage disk failure, increased replication factor .

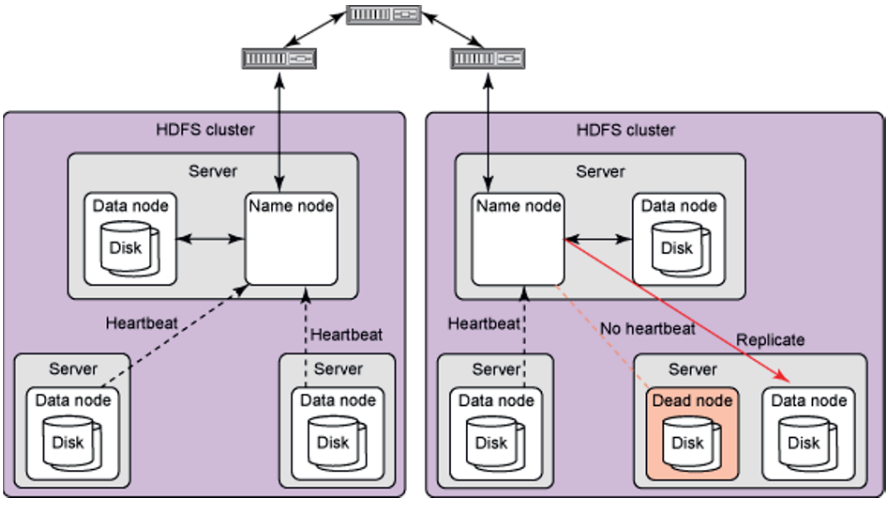


Fig. 2.5: Heart Beat Mechanism in Hadoop [22]

Cluster Rebalancing: Data re-balancing schemes are compatible with the HDFS architecture. It automatically moves the data from a Datanode when its free space falls below a certain threshold. In case a file is repeatedly used in an application, a scheme might create additional replicas of the file to re-balance the thirst for the data on the file in the cluster.

Data Integrity: Data corruption can occur for various reasons like storage device failure, network faults, buggy software, etc. To recognize corrupted data blocks HDFS implements a checksum to check on the contents of the retrieved HDFS files. Each block of data has an associated checksum that is stored in a separate hidden file in the same HDFS namespace. When a file is retrieved the quality of the file is checked with the checksum stored, if not then the client can ask for that data block from another Datanode (Figure 2.6).

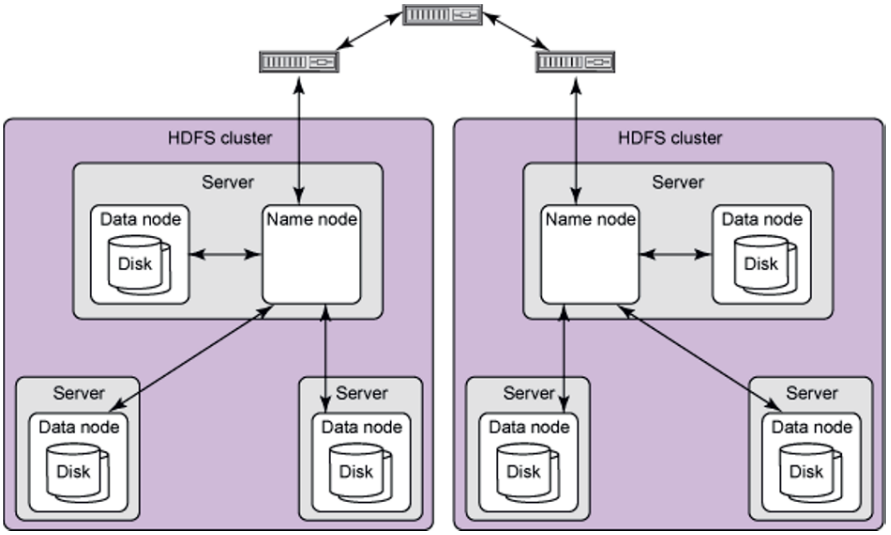


Fig. 2.6: Each cluster contains one Namenode. This design facilitates a simplified model for managing each Namespace and arbitrating data distribution [22]

2.3.6 Data Organization

Data Blocks: HDFS by design supports very large files. Applications written to work on HDFS write their data once and read many times, with reads at streaming speeds. A typical block size used by HDFS is 64 MB. Each file is chopped into 64 MB chunks and replicated.

Staging: A request to create a file does not reach the Namenode immediately. Initially, The HDFS client caches the file data into a temporary local file. Application writes are redirected to this temporary files. When the local file accumulates content over the block size, the client contacts the Namenode. The Namenode creates a file in the file-system and allocates a data block for it. The Namenode responds to the client with the Datanode and data block identities. The client flushes the block of data from the temporary file to this data block. If the file is closed the Namenode is informed, and it starts a file creation operation into a persistent store. Suppose, if the Namenode dies before commit the file is lost.

The benefits of the above approach is to allow streaming write to files. If a client writes a file directly to a remote file without buffering, the network speed and network congestion impacts the throughput considerably. There are earlier file-systems that successfully use client side caching to improve performance. In case of HDFS a few POSIX rules have been relaxed to achieve high performance data uploads.

Replication Pipelining: Local file caching mechanism described in the previous section is used for writing data to an HDFS file. Suppose the HDFS file has a replication factor of three. In such event the local file accumulates a data block of data, and the client receives a list of Datanodes from the Namenode that will host the replica of the data block. The client then flushes the data block in the first Datanode. Each Datanode in the list receives data block in smaller chunks of 4KB, the first Datanode persists this chunk in its repository after which it is flushed to the second Datanode. The second Datanode in turn persists this data chunk in its repository and flushes to the third Datanode, which persists the chunk in its repository. Thus the Datanodes receive data from the previous nodes in a *pipeline* and at the same time forwarding to the next Datanode in the pipeline.

2.4 MapReduce Preliminaries

Functional Programming: MapReduce framework facilitates computing with large volumes of data in parallel. This requires the workload to be divided across several machines. This model scales because there is no intrinsic sharing of data as the communication overhead needed to keep the data synchronized prevents the cluster from performing reliably and efficiently at large scale.

All the data elements of MapReduce cannot be updated, i.e., immutable. If you change the (key, value) pair in a mapping task, the change is not persisted. A new output (key, value) pair is created before it is forwarded by the Hadoop system into the next phase of execution.

MapReduce: Hadoop *MapReduce* is a simple software framework used to process vast amounts of data in the scale of higher terabytes in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. A MapReduce job splits the dataset into independent chunks that are processed by many *map* tasks in parallel. The framework then sorts the output of this *map* which are then input to the *reduce* task. The input and output of the job are stored on the file-system. The framework also takes care of scheduling, monitoring, and re-launching failed tasks. Typically, in Hadoop the storage and compute nodes are the same, the MapReduce framework and HDFS run on the same set of machines. This allows for the framework to schedule jobs on nodes where data are already present, resulting in high-throughput across the cluster.

The framework consists of one *JobTracker*on the master/Namenode and one *TaskTracker* per cluster node/Datanodes. The master is responsible for scheduling jobs, which are executed as tasks on the Datanodes. The Namenode is responsible for monitoring, re-executing failed tasks.

List Processing: MapReduce as paradigm transforms a list of input data into another set of output lists. The list processing idioms are *map* and *reduce*. These are principles of several functional programming languages like Scheme, LISP, etc.

Mapping Lists: A list of items provided one at a time to a function called the *mapper*, which transforms these elements one at a time to an output data element. For example, a function like *toUpper* converts a string into its uppercase version, and is applied to all the items in the list; the input string is not modified but a new transformed string is created and returned (Figure 2.7).

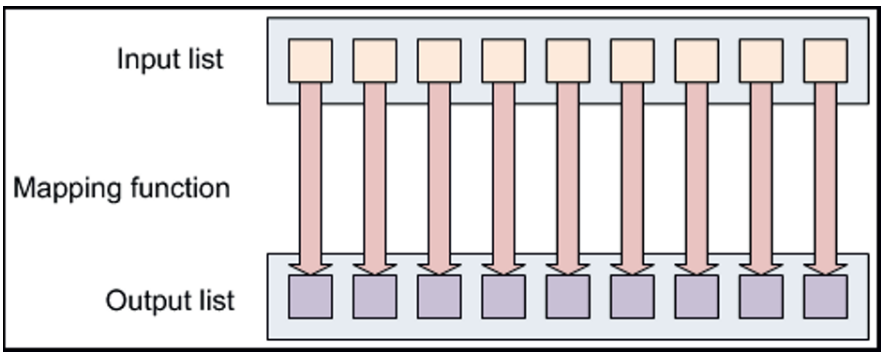


Fig. 2.7: Mapping creates a new output list by applying a function to individual elements of an input list [22].

Reducing Lists: The intermediate outputs of the mappers are sent to a *reducer* function that receives an iterator over input values. It combines these values returning a single output (Figure 2.8).

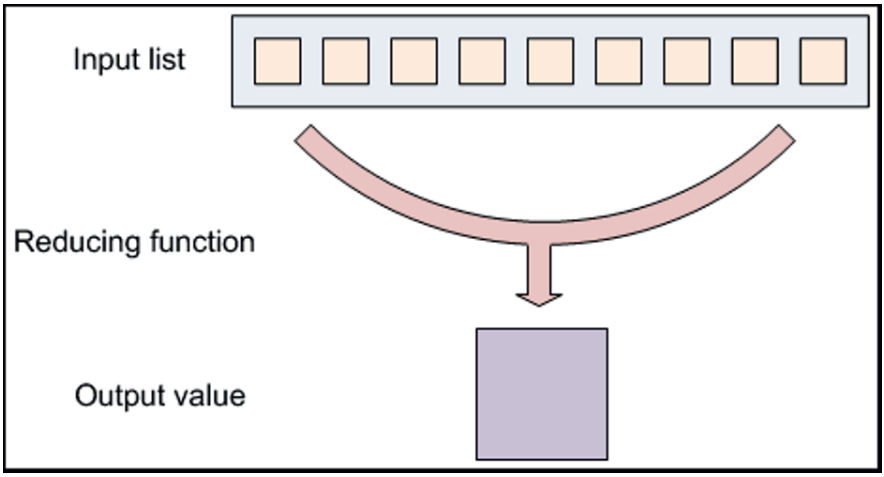


Fig. 2.8: reducing a list iterates over the input values to produce an aggregate value as output [22].

Keys and Values: : Every value in an input for MapReduce program has a key associated to it. Keys identify related values. A collection of timestamped speedometer readings, the value, from different cars has the license plate number, the key, associated to it.

```
AAA-123 65mph, 12:00pm
ZZZ-789 50mph, 12:02pm
AAA-123 40mph, 12:05pm
CCC-456 25mph, 12:15pm
...
```

The mapping and reducing functions receive not just values but (key, value) pairs. The output of each of these functions is the same: both a key and a value must be emitted to the next list in the data flow.

MapReduce is also less strict than other languages about how the Mapper and Reducer work. In more formal functional mapping and reducing settings, a mapper must produce exactly one output element for each input element, and a reducer must produce exactly one output element for each input list. In MapReduce, an arbitrary number of values can be output from each phase; a mapper may map one input into zero, one, or one hundred outputs. A reducer may compute over an input list and emit one or a dozen different outputs.

Keys divide the reduce space : A reducing function turns a large list of values into one (or a few) output values. In MapReduce, all of the output values are not usually reduced together. All of the values with the same key are presented to a single reducer together. This is performed independently of any reduce operations occurring on other lists of values, with different keys attached (Figure 2.9).

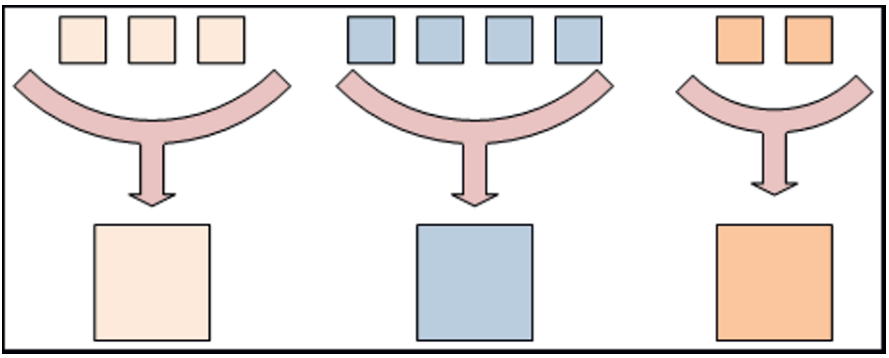


Fig. 2.9: Different colors represent different keys. All values with the same key are presented to a single reduce task [22].

2.5 Prerequisites for Installation

Hadoop cluster installation requires the following software packages:

1. Linux-based operating system, preferably Ubuntu
2. Java version 1.6 or higher
3. Secure Shell Access across the machines

Step 1: Check for Java in the systems:

```
user@ubuntu:~$ java -version
java version "1.6.0_27"
OpenJDK Runtime Environment (IcedTea6 1.12.6) (6b27-1.12.6-1
    ubuntu0.12.04.2)
OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)
```

If the above command does not yield the correct output.

```
user@ubuntu:~$ sudo apt-get install openjdk-6-jdk
```

The installation is available in:

```
/usr/lib/jvm/java-6-openjdk-amd64
```

Step 2: The communication between nodes in the cluster happens via *ssh*. In a multi-node cluster setup of communication is between individual nodes, while in sing-node cluster, localhost acts as server.

Execute the following command to install ssh:

```
user@ubuntu:~$ sudo apt-get install openssh-server openssh-
    client
```

After the installation, generate an ssh key:

```
user@ubuntu:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Created directory '/home/hduser/.ssh'.
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
9b:82:ea:58:b4:e0:35:d7:ff:19:66:a6:ef:ae:0e:d2 hduser@ubuntu
The key's randomart image is:
[...snipp...]
user@ubuntu:~$
```

Note : If there is an error you must check your ssh installation, repeat Step 2.

Step 3: In order to establish password-less communication, the public key is transfered to the slave machines from the master, which will be discussed in the following sections. In the case of single-node cluster, the slave and master are in the same machine but communication is over the localhost server.

```
user@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/
    authorized_keys
```

Step 4: Test the password-less connection by creating an ssh connection to localhost.

```
user@ubuntu:~$ ssh localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is d7:87:25:47:ae:02:00:eb:1d:75:4f:bb:44:f9
    :36:26.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of
    known hosts.
Linux ubuntu 2.6.32-22-generic #33-Ubuntu SMP Wed Apr 28
    13:27:30 UTC 2010 i686 GNU/Linux
Ubuntu 10.04 LTS
[...snipp...]
user@ubuntu:~$
```

If the connection should fail, these general tips might help:

- Enable debugging with `ssh -vvv localhost` and investigate the error in detail.
- Check the SSH server configuration in `/etc/ssh/sshd_config`, in particular the options `PubkeyAuthentication` (which should be set to yes) and `AllowUsers` (if this option is active, add the user user to it). If you made any changes to the SSH server configuration file, you can force a configuration reload with `sudo /etc/init.d/ssh reload`.

2.6 Single Node Cluster Installation

Step 1: Download and extract Hadoop source

There are several ways to install Hadoop on your system:

- Ubuntu Hadoop Package(.deb)
- Hadoop Source Code
- Third Party Hadoop Distributions (Cloudera, Hortonworks etc)

We are going to install hadoop from its source. Download Hadoop 1.0.3 from the link below:

```
user@ubuntu:~$ wget https://dl.dropboxusercontent.com/u
    /26579166/hadoop-1.0.3.tar.gz
```

Unpack the Hadoop source code (I have assumed your current working directory as home directory, you can pick your own directory)

```
user@ubuntu:~$ sudo tar xzf hadoop-1.0.3.tar.gz
user@ubuntu:~$ sudo mv hadoop-1.0.3 hadoop
```

Hadoop source is present in path: `/home/user/hadoop`

Step 2: Update the system `.bashrc` file found in the `/home` folder. Set the following environment variables for Hadoop use:

- Set `HADOOP_HOME` to the extracted folder, as shown in Step 1.
- Set `JAVA_HOME` to the installed Java path, shown in Step 1 of Section 2.5.
- Append HADOOP_HOME/bin to the system path, so that the executables are visible system-wide.

```
# Set Hadoop-related environment variables
export HADOOP_HOME=/home/user/hadoop
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
export PATH=$PATH:$HADOOP_HOME/bin
```

Step 3: Configuring Hadoop is simple, in the sense the changes have to be made in the following files found in `/home/user/hadoop/conf`:

1. `hadoop-env.sh`: Set the `JAVA_HOME` environmental variable, this is specific to Hadoop environment settings.

   Change:

```
# The java implementation to use.  Required.
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

   to:

```
# The java implementation to use.  Required.
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
```

2. `core-site.xml`: This file sets the HDFS path, where the file-system is installed and the data is stored. Create a folder `/home/user/hadoop_tmp`, which will be used as the storage location. Since the file is of `xml` format, the variables are set as `property`.

   Change:

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>Enter absolute path here</value>
  <description>A base for other temporary directories.
      </description>
</property>
```

   To:

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/user/hadoop_tmp</value>
  <description>A base for other temporary directories.
      </description>
</property>
```

   Final `core-site.xml` looks like this:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/user/hadoop_tmp</value>
  <description>A base for other temporary directories.
      </description>
</property>
```

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
  <description>The name of the default file system. A URI
      whose scheme and authority determine the FileSystem
      implementation. The uri's scheme determines the config
      property (fs.SCHEME.impl) naming the FileSystem
      implementation class.  The uri's authority is used to
      determine the host, port, etc. for a filesystem.
  </description>
</property>
</configuration>
```

3. `mapred-site.xml`: This file contains the configuration paramters set/unset for MapReduce jobs. One such parameter is the URL for `JobTracker`.

   Add the following code to the file:

```
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
  <description>The host and port that the MapReduce job
      tracker runs at. If "local", then jobs are run in-
      process as a single map and reduce task.
  </description>
</property>
```

4. `hdfs-site.xml`: The replication factor is set/unset in the file by adding the following configuration setting:

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>Default block replication. The actual number
      of replications can be specified when the file is
      created. The default is used if replication is not
      specified in create time.
  </description>
</property>
```

Step 4: Formatting file-system

So far the environmental variables and configuration files have been changed to suit the needs of a cluster. Installing the Hadoop Distributed File System(HDFS) means to format the installation path, as shown in Step 3, that is, `/home/user/hadoop_tmp`.

Execute the following command:

```
user@ubuntu:~$ ./hadoop/bin/hadoop namenode -format
```

The expected output of successful format looks like below:

```
user@ubuntu:~$ ./hadoop/bin/hadoop namenode -format
13/10/18 10:16:39 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = ubuntu/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 1.0.3
STARTUP_MSG:   build = https://svn.apache.org/repos/asf/hadoop/
    common/branches/branch-1.0 -r 1335192; compiled by 'hortonfo'
     on Tue May  8 20:31:25 UTC 2012
************************************************************/
13/10/18 10:16:39 INFO util.GSet: VM type       = 64-bit
13/10/18 10:16:39 INFO util.GSet: 2% max memory = 17.77875 MB
13/10/18 10:16:39 INFO util.GSet: capacity      = 2^21 = 2097152
    entries
13/10/18 10:16:39 INFO util.GSet: recommended=2097152, actual
    =2097152
13/10/18 10:16:39 INFO namenode.FSNamesystem: fsOwner=user
13/10/18 10:16:39 INFO namenode.FSNamesystem: supergroup=
    supergroup
13/10/18 10:16:39 INFO namenode.FSNamesystem: isPermissionEnabled
    =true
13/10/18 10:16:39 INFO namenode.FSNamesystem: dfs.block.
    invalidate.limit=100
13/10/18 10:16:39 INFO namenode.FSNamesystem:
    isAccessTokenEnabled=false accessKeyUpdateInterval=0 min(s),
    accessTokenLifetime=0 min(s)
13/10/18 10:16:39 INFO namenode.NameNode: Caching file names
    occuring more than 10 times
13/10/18 10:16:39 INFO common.Storage: Image file of size 109
    saved in 0 seconds.
13/10/18 10:16:40 INFO common.Storage: Storage directory /home/
    user/hadoop_tmp/dfs/name has been successfully formatted.
13/10/18 10:16:40 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1
************************************************************/
user@ubuntu:~$
```

Note: Always stop the cluster before formatting.

Step 5: After the file-system is installed, start the cluster to check if all the framework daemons are working.

Run the command:

```
user@ubuntu:~$ /usr/local/hadoop/bin/start-all.sh
```

This will startup a *Namenode*, *Datanode*, *Jobtracker* and a *Tasktracker* on your machine as daemon services.

```
user@ubuntu:~$ ./hadoop/bin/start-all.sh
starting namenode, logging to /home/user/hadoop/libexec/../logs/
    hadoop-user-namenode-ubuntu.out
localhost: starting datanode, logging to /home/user/hadoop/
    libexec/../logs/hadoop-user-datanode-ubuntu.out
localhost: starting secondarynamenode, logging to /home/user/
    hadoop/libexec/../logs/hadoop-user-secondarynamenode-ubuntu.
    out
starting jobtracker, logging to /home/user/hadoop/libexec/../
    logs/hadoop-user-jobtracker-ubuntu.out
localhost: starting tasktracker, logging to /home/user/hadoop/
    libexec/../logs/hadoop-user-tasktracker-ubuntu.out
user@ubuntu:~$
```

To check if all the daemon services are functional, use the built-in Java process tool: `jps`

```
user@ubuntu:~$ jps
21128 Jps
20328 DataNode
20596 SecondaryNameNode
20689 JobTracker
20976 TaskTracker
19989 NameNode
```

Note: If any of above services are missing, debug by looking at the logs in `/home/user/hadoop/logs`.

Step 6: Stopping the cluster

To stop all the daemons execute the following command:

```
user@ubuntu:~$ ./hadoop/bin/stop-all.sh
```

output:

```
user@ubuntu:~$ ./hadoop/bin/stop-all.sh
stopping jobtracker
localhost: stopping tasktracker
stopping namenode
localhost: stopping datanode
localhost: stopping secondarynamenode
user@ubuntu:~$
```

2.7 Multi-node Cluster Installation

To enable simple installation of a multi-node cluster, it is recommended that every node in the proposed cluster have Hadoop installed and configured as per the single-node installation instructions in Section 2.6. In the case of single node both master and slave are on the same machine, after installation it is only a matter of identifying the ”master” and ”slave” machines.

Here we try to create a multi-node cluster with two nodes, as shown in Figure 2.10. We will modify the Hadoop configuration to make one Ubuntu box the ”master” (which will also act as a slave) and the other Ubuntu box a ”slave”.

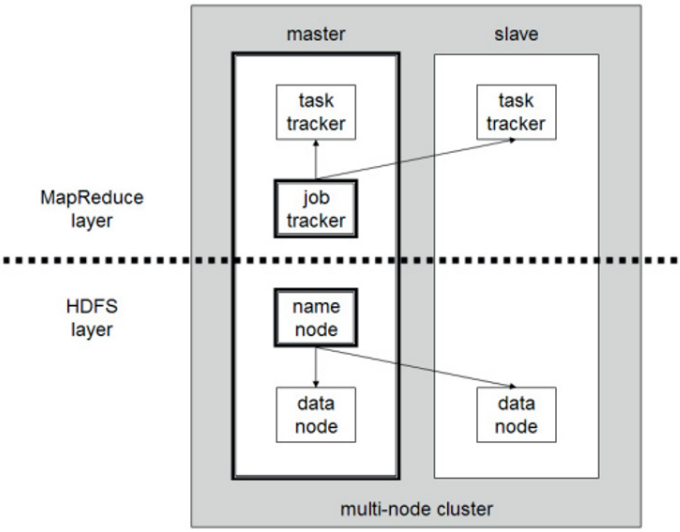


Fig. 2.10: Multi-node Cluster Overview

Note 1: We will call the designated master machine just the ”`master`” from now on and the slave-only machine the ”`slave`”. We will also give the two machines these respective hostnames in their networking setup, most notably in `/etc/hosts`. If the hostnames of your machines are different (e.g. ”node01”) then you must adapt the settings in this section as appropriate.

Note 2: Shutdown each single-node cluster with `bin/stop-all.sh` before continuing if you haven’t done so already.

Step 1: Networking

While creating a multi-node cluster it is assumed that all the nodes in the proposed cluster are reachable over the network. Ideally, these are connected to one hub or switch. For example, machines are connected to a switch and are identified by the address sequence `192.168.0.x/24`.

Since we have a two-node cluster, IP addresses `192.168.0.1` and `192.168.0.2` are assigned to nodes respectively. Update `/etc/hosts` on both machines with the following lines:

```
192.168.0.1    master
192.168.0.2    slave
```

Step 2: Secure Shell Access across the network.

In Sections 2.6 and 2.5 password-less logins are created by generating an ssh key and adding that key to `authorized` key listing. Similarly, the key generated on the `master` node is transfered to each of the `slave` nodes. The keys must be transfered so that Hadoop jobs can be started and stopped without the user authentication.

execute the following command from the `master` node.

```
user@master:~$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub user@slave
```

This will prompt you to enter the login password for user `user` on the `slave` machine, copy the `master` public key and assign correct directory and permissions.

```
user@master:~$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub user@slave
```

After successful transfer of the keys, test the connection to each of the `slave` nodes from the `master`. The step is also needed to save slaves host key fingerprint to the `hduser@masters known_hosts` file.

`master` to `master`:

```
user@master:~$ ssh master
The authenticity of host 'master (192.168.0.1)' can't be
    established.
RSA key fingerprint is 3b:21:b3:c0:21:5c:7c:54:2f:1e:2d:96:79:eb
    :7f:95.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'master' (RSA) to the list of known
    hosts.
Linux master 2.6.20-16-386 #2 Fri Oct 18 20:16:13 UTC 2013 i686
...
user@master:~$
```

`master` to `slave`:

```
user@master:~$ ssh slave
The authenticity of host 'slave (192.168.0.2)' can't be
    established.
RSA key fingerprint is 74:d7:61:86:db:86:8f:31:90:9c:68:b0
    :13:88:52:72.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'slave' (RSA) to the list of known
    hosts.
Ubuntu 12.04
...
user@slave:~$
```

Step 3: Multi-node configuration

With reference Step 3 in Section 2.6 the following files in the `/home/user/hadoop/conf` directory need to be changed:

1. `core-site.xml` on all machines

   Change the `fs.default.name` parameter, which specifies the NameNode (the HDFS master) host and port. In our case, this is the master machine.

   Change:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
</property>
```

   To:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://master:54310</value>
</property>
```

2. `hdfs-site.xml` on all machines

   change the |dfs.replication— parameter (in |conf/hdfs-site.xml—) which specifies the default block replication. It defines how many machines a single file should be replicated to before it becomes available.

   The default value of `dfs.replication` is `3`. However, we have only two nodes available, so we set `dfs.replication` to `2`.

   Change:

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
```

To:

```
<property>
  <name>dfs.replication</name>
  <value>2</value>
</property>
```

3. `mapred-site.xml` on all machines

   Change the `mapred.job.tracker` parameter which specifies the `Job Tracker` (MapReduce master) host and port. Again, this is the `master` in our case.

   Change:

```
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
  </property>
```

   To:

```
<property>
  <name>mapred.job.tracker</name>
  <value>master:54311</value>
</property>
```

4. `masters` file, on the `master` node

   This file identifies the machines as the `Namenode` and `Secondary Namenodes` associated by the IP addresses. Enter the hostname that has been allocated as the `master` in the `/etc/hosts` file.

```
master
```

5. `slaves` file, on the `master` node

   This file lists the hosts, one per line, where the Hadoop slave daemons (DataNodes and TaskTrackers) will be run. Instead of wasting the compute resource of the master we add `master` as a slave so that it can share the load of cluster.

```
master
slave
```

   If you have additional slave nodes, just add them to the `slaves` file, one hostname per line.

```
master
slave
anotherslave01
anotherslave02
anotherslave03
```

Note: Typically, one machine in the cluster is designated as the `NameNode` and another machine the as `JobTracker`, exclusively. These are the actual "master nodes". The rest of the machines in the cluster act as both `DataNode` and `TaskTracker`. These are the slaves or "worker nodes".

Additional Configuration Parameters: There are some other configuration options worth studying

In file `conf/mapred-site.xml`:

- *mapred.local.dir*

  Determines where temporary MapReduce data is written. It also may be a list of directories.

- *mapred.map.tasks*

  As a rule of thumb, use 10x the number of slaves (i.e., number of TaskTrackers).

- *mapred.reduce.tasks*

  As a rule of thumb, use

  `num_tasktrackers * num_reduce_slots_per_tasktracker * 0.99`.

  If `num_tasktrackers` is small

  use (`num_tasktrackers - 1`) * `num_reduce_slots_per_tasktracker`.

Step 4: Starting multi-node cluster

Starting the cluster is performed in two steps.

- We begin with starting the HDFS daemons: the `Namenode` daemon is started on `master`, and `DataNode` daemons are started on all slaves (here: `master` and `slave`).

  Run the command `bin/start-dfs.sh` on the machine you want the (primary) NameNode to run on. This will bring up HDFS with the NameNode running on the machine you ran the previous command on, and DataNodes on the machines listed in the `conf/slaves` file.

```
user@master:~$ ./hadoop/bin/start-dfs.sh
starting namenode, logging to /usr/local/hadoop/bin/../logs/
    hadoop-hduser-namenode-master.out
slave: Ubuntu 10.04
slave: starting datanode, logging to /usr/local/hadoop/bin
    /../logs/hadoop-hduser-datanode-slave.out
master: starting datanode, logging to /usr/local/hadoop/bin
    /../logs/hadoop-hduser-datanode-master.out
```

```
master: starting secondarynamenode, logging to /usr/local/
    hadoop/bin/../logs/hadoop-hduser-secondarynamenode-master.
    out
user@master:~$
```

At this point, the following Java processes should run on `master`

```
user@master:~$ jps
16017 Jps
14799 NameNode
14880 DataNode
14977 SecondaryNameNode
hduser@master:~$
```

and the following Java processes should run on every `slave` node.

```
user@master:~$ jps
15183 DataNode
16284 Jps
hduser@master:~$
```

- Then we start the MapReduce daemons: the `JobTracker` is started on master, and `TaskTracker` daemons are started on all slaves (here: `master` and `slave`).

```
user@master:~$ ./hadoop/bin/start-mapred.sh
```

At this point, the following Java processes should run on `master`

```
user@master:~$ jps
16017 Jps
14799 NameNode
15686 TaskTracker
14880 DataNode
15596 JobTracker
14977 SecondaryNameNode
hduser@master:~$
```

and the following Java processes should run on every `slave` node.

```
user@master:~$ jps
15183 DataNode
15897 TaskTracker
16284 Jps
hduser@master:~$
```

Step 5: Stopping the cluster

Similar to Step 4, the cluster is stopped in two steps:

- Stop the MapReduce daemons: the JobTracker is stopped on `master`, and TaskTracker daemons are stopped on all slaves (here: `master` and `slave`).

  Run the command `bin/stop-mapred.sh` on the `JobTracker` machine. This will shut down the MapReduce cluster by stopping the JobTracker daemon running on the machine you ran the previous command on, and `TaskTrackers` on the machines listed in the `conf/slaves` file.

  In our case, we will run bin/stop-mapred.sh on master:

```
user@master:~$./hadoop/bin/stop-mapred.sh
stopping jobtracker
slave: Ubuntu 10.04
master: stopping tasktracker
slave: stopping tasktracker
user@master:~$
```

  At this point, the following Java processes should run on `master`

```
user@master:~$ jps
16017 Jps
14799 NameNode
14880 DataNode
14977 SecondaryNameNode
hduser@master:~$
```

  and the following Java processes should run on `slave`

```
user@master:~$ jps
15183 DataNode
16284 Jps
hduser@master:~$
```

- Stop the HDFS daemons: the NameNode daemon is stopped on `master`, and DataNode daemons are stopped on all slaves (here: `master` and `slave`).

  Run the command `bin/stop-dfs.sh` on the NameNode machine. This will shut down HDFS by stopping the NameNode daemon running on the machine you ran the previous command on, and DataNodes on the machines listed in the `conf/slaves` file.

  In our case, we will run `bin/stop-dfs.sh` on `master`:

```
user@master:~$./hadoop/bin/stop-dfs.sh
stopping namenode
slave: Ubuntu 10.04
slave: stopping datanode
master: stopping datanode
master: stopping secondarynamenode
user@master:~$
```

  At this point, only the following Java processes should run on `master`

```
user@master:~$ jps
18670 Jps
user@master:~$
```

and the following on `slave`.

```
user@slave:~$ jps
18894 Jps
user@slave:~$
```

2.8 Hadoop Programming

Running `wordcount` Hadoop MapReduce job. This program reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count of how often it occurred, separated by a tab.

Download input data: The Notebooks of Leonardo Da Vinci

```
user@ubuntu:~$ wget http://www.gutenberg.org/cache/epub/5000/
    pg5000.txt
```

output

```
user@ubuntu:~$ wget http://www.gutenberg.org/cache/epub/5000/
    pg5000.txt
--2013-10-18 11:02:36--  http://www.gutenberg.org/cache/epub
    /5000/pg5000.txt
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47
Connecting to www.gutenberg.org (www.gutenberg.org)
    |152.19.134.47|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1423803 (1.4M) [text/plain]
Saving to: 'pg5000.txt'

100%[======================================>] 14,23,803  241K/s
    in 5.8s

2013-10-18 11:02:43 (241 KB/s) - 'pg5000.txt' saved
    [1423803/1423803]
```

Restart the Hadoop cluster: using script `start-all.sh`

```
user@ubuntu:~$ ./hadoop/bin/start-all.sh
```

Copy local example data to HDFS: using `copyFromLocal` dfs option.

```
user@ubuntu:~$ ./hadoop/bin/hadoop dfs -copyFromLocal pg5000.txt
     input
user@ubuntu:~$ ./hadoop/bin/hadoop dfs -ls
Found 1 items
-rw-r--r--   1 user supergroup    1423803 2013-10-18 11:35 /user
    /user/input
```

Run wordcount program: The wordcount program has already been compiled and stored as a jar file. This jar file comes with the downloaded Hadoop source. The wordcount example can be found in the `hadoop-examples-1.0.3.jar`.

```
user@ubuntu ./hadoop/bin/hadoop jar hadoop-examples-1.0.3.jar
    wordcount input output
```

This command will read all the files in the HDFS directory `/user/user/input`, process it, and store the result in the HDFS directory `/user/user/output`.

```
user@ubuntu:~$ ./hadoop/bin/hadoop jar hadoop-examples-1.0.3.jar
     wordcount input output
13/10/18 13:47:34 INFO input.FileInputFormat: Total input paths
    to process : 1
13/10/18 13:47:34 INFO util.NativeCodeLoader: Loaded the native-
    hadoop library
13/10/18 13:47:34 WARN snappy.LoadSnappy: Snappy native library
    not loaded
13/10/18 13:47:34 INFO mapred.JobClient: Running job:
    job_201310181134_0001
13/10/18 13:47:35 INFO mapred.JobClient:  map 0% reduce 0%
13/10/18 13:47:48 INFO mapred.JobClient:  map 100% reduce 0%
13/10/18 13:48:00 INFO mapred.JobClient:  map 100% reduce 100%
13/10/18 13:48:05 INFO mapred.JobClient: Job complete:
    job_201310181134_0001
.
.
.
13/10/18 13:48:05 INFO mapred.JobClient:     Virtual memory
    (bytes) snapshot=4165484544
13/10/18 13:48:05 INFO mapred.JobClient:     Map output records
    =251352
```

Check if the result is successfully stored in HDFS directory `/user/user/output`

```
user@ubuntu:~$ ./hadoop/bin/hadoop dfs -ls output
Found 3 items
-rw-r--r--   1 user supergroup          0 2013-10-18 13:48 /user
    /user/output/_SUCCESS
drwxr-xr-x   - user supergroup          0 2013-10-18 13:47 /user
    /user/output/_logs
-rw-r--r--   1 user supergroup     337648 2013-10-18 13:47 /user
    /user/output/part-r-00000
```

Inspect if the output has been correctly populated.

```
user@ubuntu:~$ hadoop dfs -cat /user/mak/output/part-r-00000
"(Lo)cra"   1
"1490   1
"1498," 1
"35"    1
"40,"   1
"AS-IS".    1
"A_ 1
"Absoluti   1
"Alack! 1
"Alack!"    1
.
.
.
```

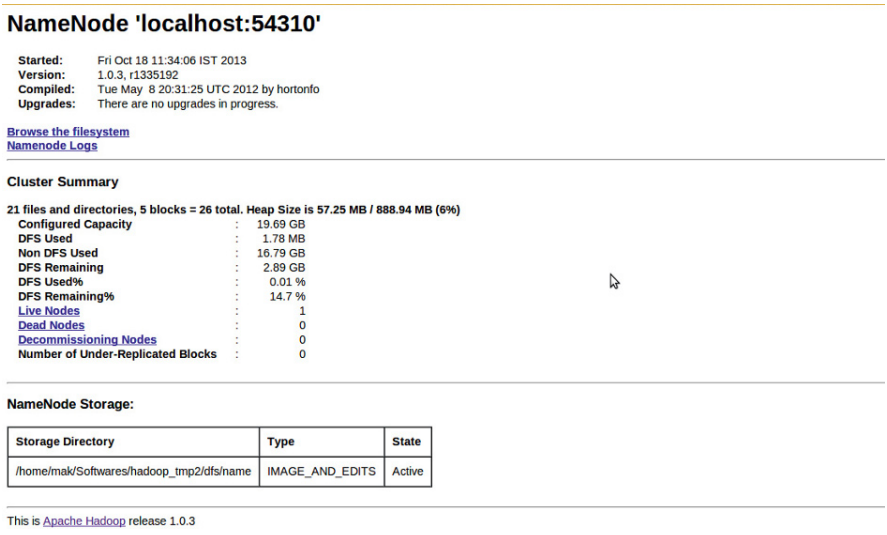
Hadoop Web Interfaces: The progress of the Hadoop job can be monitored using the built-in web interfaces. These interfaces provide a graphical representation of the jobs.

`http://localhost:50070/` — web UI of the `NameNode` daemon
`http://localhost:50030/` — web UI of the `JobTracker` daemon
`http://localhost:50060/` — web UI of the `TaskTracker` daemon

NameNode Web Interface (HDFS layer): It shows details of the storage capacity and usage, number of live nodes and dead nodes, access to logs, browse file-system, etc (Figure 2.11).

JobTracker Web Interface (MapReduce layer): It provides information about job statistics, job statuses if running/completed/failed and also job history logs (Figure 2.12).

TaskTracker Web Interface (MapReduce layer): It shows the status of running and non-running tasks. It also gives access to the tasktracker log files of that task.



NameNode 'localhost:54310'

Started: Fri Oct 18 11:34:06 IST 2013
Version: 1.0.3, r1335192
Compiled: Tue May 8 20:31:25 UTC 2012 by hortonfo
Upgrades: There are no upgrades in progress.

Browse the filesystem
Namenode Logs

Cluster Summary

21 files and directories, 5 blocks = 26 total. Heap Size is 57.25 MB / 888.94 MB (6%)

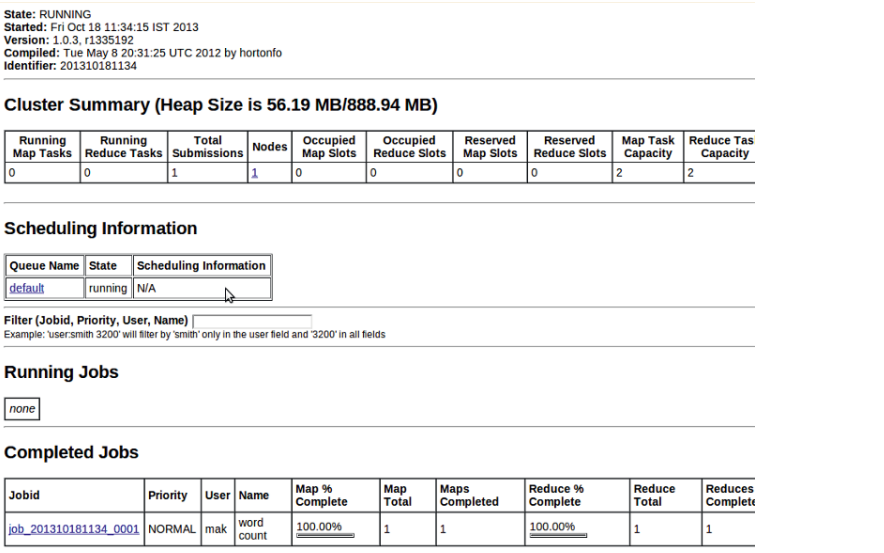
Configured Capacity	:	19.69 GB
DFS Used	:	1.78 MB
Non DFS Used	:	16.79 GB
DFS Remaining	:	2.89 GB
DFS Used%	:	0.01 %
DFS Remaining%	:	14.7 %
Live Nodes	:	1
Dead Nodes	:	0
Decommissioning Nodes	:	0
Number of Under-Replicated Blocks	:	0

NameNode Storage:

Storage Directory	Type	State
/home/mak/Softwares/hadoop_tmp2/dfs/name	IMAGE_AND_EDITS	Active

This is Apache Hadoop release 1.0.3

Fig. 2.11: NameNode Web Interface (HDFS layer)



State: RUNNING
Started: Fri Oct 18 11:34:15 IST 2013
Version: 1.0.3, r1335192
Compiled: Tue May 8 20:31:25 UTC 2012 by hortonfo
Identifier: 201310181134

Cluster Summary (Heap Size is 56.19 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Tas Capacity
0	0	1	1	0	0	0	0	2	2

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Complete
job_201310181134_0001	NORMAL	mak	word count	100.00%	1	1	100.00%	1	1

Fig. 2.12: JobTracker Web Interface (MapReduce layer)

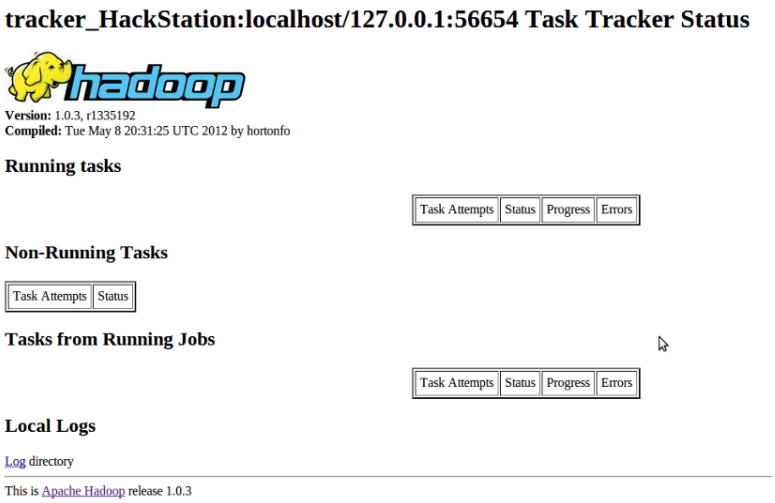


Fig. 2.13: TaskTracker Web Interface (MapReduce layer).

2.9 Hadoop Streaming

Hadoop streaming is a utility that helps users to write and run MapReduce jobs using any executable or script as a mapper and/or reducer. Streaming is similar to a `pipe` operation in Linux. The text input is printed on the `stdin` stream, then passed to the Mapper script reading from `stdin`. The resulting output is written to the `stdout` stream which is then passed to Reducer. Finally, the reducer writes the output to a storage location, typically an HDFS directory.

The command line interface looks as follows.

```
$HADOOP_HOME/bin/hadoop jar \
    $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper /bin/cat \
    -reducer /bin/wc
```

Here the `-mapper` and `-reducer` are Linux executables that read and write to `stdin` and `stdout` respectively. The `hadoop-streaming.jar` utility creates a MapReduce job on the cluster and monitors its progress.

When the mapper tasks are executed, the input is converted into lines and fed into the stdin of the mapper task. The mapper processes these lines as they come and converts them to (key, value) pairs, which are then sent out to the stdout. These (key, value) pairs are then fed into the stdin of the reducer task launched after the completion of the mapper. The (key, value) pairs are reduced by the key and

operation is performed on the values. The results are then pushed to the stdout that is written to persistent storage. This is how streaming works with the MapReduce framework.

Consequently, this is not limited to scripting languages. Java classes can also be plugged in as mapper classes.

```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-
   streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper org.apache.hadoop.mapred.lib.IdentityMapper \
    -reducer /bin/wc
```

Note: Streaming tasks exiting with nonzero status are considered failed.

Streaming Command Line Interface: Streaming supports streaming command options as well as generic command options. The general command line syntax is shown below.

Note: Be sure to place the generic options before the streaming options, otherwise the command will fail.

```
bin/hadoop command [genericOptions] [streamingOptions]
```

The Hadoop generic command options you can use with streaming are listed here [28]:

Commands	Description
-input directoryname or filename	(Required) Input location for mapper
-output directoryname	(Required) Output location for reducer
-mapper JavaClassName	(Required) Mapper executable
-reducer JavaClassName	(Required) Reducer executable
-file filename	Make the mapper, reducer, or combiner executable available locally on the compute nodes
-inputformat JavaClassName	Class you supply should return key/value pairs of Text class. If not specified, TextInputFormat is used as the default
-outputformat JavaClassName	Class you supply should take key/value pairs of Text class. If not specified, TextOutputformat is used as the default
-partitioner JavaClassName	Class that determines which reduce a key is sent to
-combiner JavaClassName	Combiner executable for map output

Any executable can be specified as a mapper or a reducer, if the file is not present in the nodes then `-file` option is used to tell Hadoop to package this file as part of the job and send across the cluster nodes.

```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper myPythonScript.py \
    -reducer /bin/wc \
    -file myPythonScript.py
```

In addition to executables, other files like configuration files, dictionaries, etc., can also be shipped to the nodes while job submission using `-file` option.

```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper myPythonScript.py \
    -reducer /bin/wc \
    -file myPythonScript.py \
    -file myDictionary.txt
```

Hadoop Streaming with Python: Python-based mapper and reducer are written as executables. They will read from and write to `sys.stdin` and `sys.stdout` respectively, similar to the discussion above. Here a simple wordcount program is implemented in python as follows:

Step 1: Map Step

The mapper executable reads the lines from `sys.stdin`, splits them into words, and prints out each word as a key and associated count 1 as the value. This file is stored as `/home/user/mapper.py`.

Note: Make sure the file has execution permission `chmod +x /home/hduser/mapper.py`

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print word, "1"
```

Step 2: Reduce step

The output of the mapper is passed as the input to the reducer. The reducer groups the words as keys and sums the value associated with each word, thus counting the

frequency of words in the input file. This file is stored as `/home/user/reducer.py`.

Make sure the file has execution permission

```
chmod +x /home/user/reducer.py.
```

```
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()

    word, count = line.split()

    try:
        count = int(count)
    except ValueError:
        continue

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print current_word, current_count
        current_count = count
        current_word = word

if current_word == word:
    print current_word, current_count
```

Note: The input file must be transfered to the HDFS before executing the streaming job.

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/
    streaming/hadoop-*streaming*.jar \
  -file /home/user/mapper.py \
  -mapper /home/user/mapper.py \
  -file /home/user/reducer.py  \
  -reducer /home/user/reducer.py \
  -input input -output output
```

If you want to modify some Hadoop settings on the fly like increasing the number of Reduce tasks, you can use the `-D` option:

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/
    streaming/hadoop-*streaming*.jar -D mapred.reduce.tasks=16
    ...
```

References

1. Tom White, 2012, Hadoop: The Definitive Guide, O'reilly
2. Hadoop Tutorial, Yahoo Developer Network, http://developer.yahoo.com/hadoop/tutorial
3. Mike Cafarella and Doug Cutting, April 2004, Building Nutch: Open Source Search, ACM Queue, http://queue.acm.org/detail.cfm?id=988408.
4. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leun, g, October 2003, The Google File System, http://labs.google.com/papers/gfs.html.
5. Jeffrey Dean and Sanjay Ghemawat, December 2004, MapReduce: Simplified Data Processing on Large Clusters, http://labs.google.com/papers/mapreduce.html
6. Yahoo! Launches World?s Largest Hadoop Production Application, 19 February 2008, http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html.
7. Derek Gottfrid, 1 November 2007, Self-service, Prorated Super Computing Fun!, http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.
8. Google, 21 November 2008, Sorting 1PB with MapReduce, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
9. From Gantz et al., March 2008, The Diverse and Exploding Digital Universe, http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf
10. http://www.intelligententerprise.com/showArticle.jhtml?articleID=207800705, http://mashable.com/2008/10/15/facebook-10-billion-photos/, http://blog.familytreemagazine.com/insider/Inside+Ancestrycoms+TopSecret+Data+Center.aspx, and http://www.archive.org/about/faqs.php, http://www.interactions.org/cms/?pid=1027032.
11. David J. DeWitt and Michael Stonebraker, In January 2007 ?MapReduce: A major step backwards? http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards
12. Jim Gray, March 2003, Distributed Computing Economics, http://research.microsoft.com/apps/pubs/default.aspx?id=70001
13. Apache Mahout, http://mahout.apache.org/
14. Think Big Analytics, http://thinkbiganalytics.com/leading_ big_ data_ dtechnologies/hadoop/
15. Jeffrey Dean and Sanjay Ghemawat, 2004, MapReduce: Simplified Data Processing on Large Clusters. Proc. Sixth Symposium on Operating System Design and Implementation.
16. Olston, Christopher, et al. "Pig latin: a not-so-foreign language for data processing." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.
17. Thusoo, Ashish, et al. "Hive: a warehousing solution over a map-reduce framework." Proceedings of the VLDB Endowment 2.2 (2009): 1626-1629.
18. George, Lars. HBase: the definitive guide. " O'Reilly Media, Inc.", 2011.
19. Hunt, Patrick, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." USENIX Annual Technical Conference. Vol. 8. 2010.
20. Hausenblas, Michael, and Jacques Nadeau. "Apache drill: interactive Ad-Hoc analysis at scale." Big Data 1.2 (2013): 100-104.
21. Borthakur, Dhruba. "HDFS architecture guide." HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfs design. pdf (2008).
22. [Online] IBM DeveloperWorks, http://www.ibm.com/developerworks/library/waintrohdfs/
23. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, May 2010, The Hadoop Distributed File System, Proceedings of MSST2010, http://storageconference.org/2010/Papers/MSST/Shvachko.pdf
24. [Online] Konstantin V. Shvachko, April 2010, HDFS Scalability: The limits to growth, pp. 6?16 http://www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf
25. [Online] Micheal Noll, Single Node Cluster, http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/
26. [Online] Micheal Noll, Multi Node Cluster, http://www.michaelnoll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

27. [Online] Micheal Noll, Hadoop Streaming:Python, http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
28. Hadoop, Apache. "Apache Hadoop." 2012-03-07]. http://hadoop.apache.org (2011).

http://www.springer.com/978-3-319-13496-3

Guide to High Performance Distributed Computing
Case Studies with Hadoop, Scalding and Spark
Srinivasa, K.G.; Muppalla, A.K.
2015, XVII, 304 p. 43 illus., Hardcover
ISBN: 978-3-319-13496-3