listing veracity as one of the dimensions of big data amounts to saying that data coming into the so-called big data applications have a variety of trustworthiness, and therefore before we accept the data for analytical or other applications, it must go through some degree of quality testing and credibility analysis. Many sources of data generate data that is uncertain, incomplete, and inaccurate, therefore making its veracity questionable.

We now turn our attention to the technologies that are considered the pillars of big data technologies. It is anticipated that by 2016, more than half of the data in the world may be processed by Hadoop-related technologies. It is therefore important for us to trace the MapReduce/Hadoop revolution and understand how this technology is positioned today. The historical development starts with the programming paradigm called MapReduce programming.

25.2 Introduction to MapReduce and Hadoop

In this section, we will introduce the technology for big data analytics and data processing known as Hadoop, an open source implementation of the MapReduce programming model. The two core components of Hadoop are the MapReduce programming paradigm and HDFS, the Hadoop Distributed File System. We will briefly explain the background behind Hadoop and then MapReduce. Then we will make some brief remarks about the Hadoop ecosystem and the Hadoop releases.

25.2.1 Historical Background

Hadoop has originated from the quest for an open source search engine. The first attempt was made by the then Internet archive director Doug Cutting and University of Washington graduate student Mike Carafella. Cutting and Carafella developed a system called Nutch that could crawl and index hundreds of millions of Web pages. It is an open source Apache project.[8] After Google released the Google File System[9] paper in October 2003 and the MapReduce programming paradigm paper[10] in December 2004, Cutting and Carafella realized that a number of things they were doing could be improved based on the ideas in these two papers. They built an underlying file system and a processing framework that came to be known as Hadoop (which used Java as opposed to the C++ used in MapReduce) and ported Nutch on top of it. In 2006, Cutting joined Yahoo, where there was an effort under way to build open source technologies using ideas from the Google File System and the MapReduce programming paradigm. Yahoo wanted to enhance its search processing and build an open source infrastructure based on the Google File System and MapReduce. Yahoo spun off the storage engine and the processing parts of Nutch as **Hadoop** (named after the stuffed elephant toy of Cutting's son). The

[8]For documentation on Nutch, see http:nutch.apache.org

[9]Ghemawat, Gbioff, and Leung (2003).

[10]Dean and Ghemawat (2004).

initial requirements for Hadoop were to run batch processing using cases with a high degree of scalability. However, the circa 2006 Hadoop could only run on a handful of nodes. Later, Yahoo set up a research forum for the company's data scientists; doing so improved the search relevance and ad revenue of the search engine and at the same time helped to mature the Hadoop technology. In 2011, Yahoo spun off Hortonworks as a Hadoop-centered software company. By then, Yahoo's infrastructure contained hundreds of petabytes of storage and 42,000 nodes in the cluster. In the years since Hadoop became an open source Apache project, thousands of developers worldwide have contributed to it. A joint effort by Google, IBM, and NSF used a 2,000-node Hadoop cluster at a Seattle data center and helped further universities' research on Hadoop. Hadoop has seen tremendous growth since the 2008 launch of Cloudera as the first commercial Hadoop company and the subsequent mushrooming of a large number of startups. IDC, a software industry market analysis firm, predicts that the Hadoop market will surpass $800 million in 2016; IDC predicts that the big data market will hit $23 billion in 2016. For more details about the history of Hadoop, consult a four-part article by Harris.[11]

An integral part of Hadoop is the MapReduce programming framework. Before we go any further, let us try to understand what the MapReduce programming paradigm is all about. We defer a detailed discussion of the HDFS file system to Section 25.3.

25.2.2 MapReduce

The MapReduce programming model and runtime environment was first described by Jeffrey Dean and Sanjay Ghemawat (Dean & Ghemawat (2004)) based on their work at Google. Users write their programs in a functional style of *map* and *reduce* tasks, which are automatically parallelized and executed on large clusters of commodity hardware. The programming paradigm has existed as far back as the language LISP, which was designed by John McCarthy in late 1950s. However, the reincarnation of this way of doing parallel programming and the way this paradigm was implemented at Google gave rise to a new wave of thinking that contributed to the subsequent developments of technologies such as Hadoop. The runtime system handles many of the messy engineering aspects of parallelization, fault tolerance, data distribution, load balancing, and management of task communication. As long as users adhere to the **contracts** laid out by the MapReduce system, they can just focus on the logical aspects of this program; this allows programmers without distributed systems experience to perform analysis on very large datasets.

The motivation behind the MapReduce system was the years spent by the authors and others at Google implementing hundreds of special-purpose computations on large datasets (e.g., computing inverted indexes from Web content collected via Web crawling; building Web graphs; and extracting statistics from Web logs, such as frequency distribution of search requests by topic, by region, by type of user, etc.). Conceptually, these tasks are not difficult to express; however, given the scale

[11]Derreck Harris : 'The history of Hadoop: from 4 nodes to the future of data," at https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/

of data in billions of Web pages and with the data spread over thousands of machines, the execution task was nontrivial. Issues of program control and data management, data distribution, parallelization of computation, and handling of failures became critically important.

The MapReduce programming model and runtime environment was designed to cope with the above complexity. The abstraction is inspired by the map and reduce primitives present in LISP and many other functional languages. An underlying model of data is assumed; this model treats an object of interest in the form of a unique key that has associated content or value. This is the key-value pair. Surprisingly, many computations can be expressed as applying a map operation to each logical "record" that produces a set of intermediate key-value pairs and then applying a reduce operation to all the values that shared the same key (the purpose of sharing is to combine the derived data). This model allows the infrastructure to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance. The idea of providing a restricted programming model so that the runtime can parallelize computations automatically is not new. MapReduce is the enhancement of those existing ideas. As it is understood today, MapReduce is a fault-tolerant implementation and a runtime environment that scales to thousands of processors. The programmer is spared the worry of handling failures. In subsequent sections, we will abbreviate MapReduce as **MR**.

The MapReduce Programming Model In the following description, we use the formalism and description as it was originally described by Dean and Ghemawat (2010).[12] The map and reduce functions have the following general form:

map[K1,V1] which is (key, value) : List[K2,V2] and
reduce(K2, List[V2]) : List[K3,V3]

Map is a generic function that takes a key of type **K1** and a value of type **V1** and returns a list of key-value pairs of type **K2** and **V2**. **Reduce** is a generic function that takes a key of type **K2** and a list of values of type **V2** and returns pairs of type **(K3,V3)**. In general, the types K1, K2, K3, etc., are different, with the only requirement that the output types from the Map function must match the input type of the Reduce function.

The basic execution workflow of MapReduce is shown in Figure 25.1.

Assume that we have a document and we want to make a list of words in it with their corresponding frequencies. This ubiquitous *word count* example quoted directly from Dean and Ghemawat (2004) above goes as follows in pseudocode:

Map (String key, String value):
for each word w in value Emitintermediate (w, "1");

Here key is the document name, and value is the text content of the document.

[12]Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI (2004).

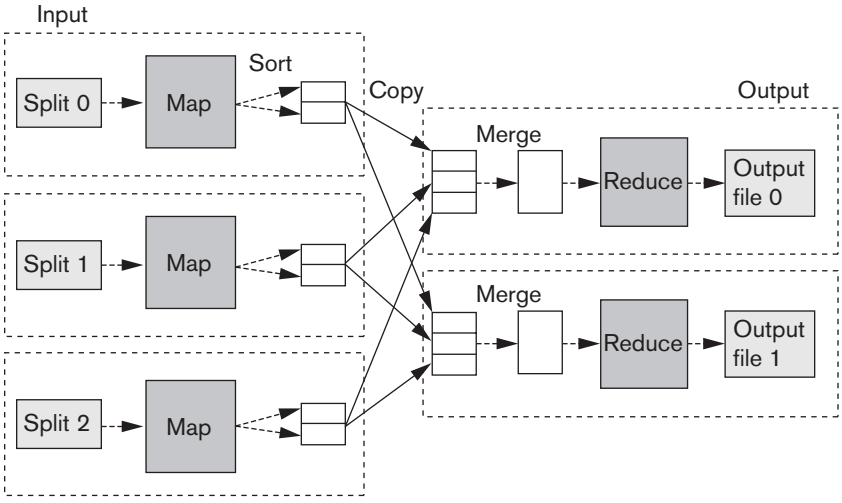

Figure 25.1
Overview of MapReduce execution. (Adapted from T. White, 2012)

Then the above lists of (word, 1) pairs are added up to output total counts of all words found in the document as follows:

```
Reduce (String key, Iterator values) : // here the key is a word and values are
lists of its counts //
    Int result =0;
    For each v in values :
        result += Parseint (v);
    Emit (key, Asstring (result));
```

The above example in MapReduce programming appears as:

```
map[LongWritable,Text](key, value) : List[Text, LongWritable] = {
    String[] words = split(value)
    for(word : words) {
        context.out(Text(word), LongWritable(1))
    }
}
reduce[Text, Iterable[LongWritable]](key, values) : List[Text, LongWritable] = {
    LongWritable c = 0
    for( v : values) {
        c += v
    }
    context.out(key,c)
    }
```

The data types used in the above example are LongWritable and Text. Each MapReduce job must register a Map and Reduce function. The Map function receives each key-value pair and on each call can output 0 or more key-value pairs. The signature of the Map function specifies the data types of its input and output

key-value pairs. The Reduce function receives a key and an iterator of values associated with that key. It can output one or more key-value pairs on each invocation. Again, the signature of the Reduce function indicates the data types of its inputs and outputs. The output type of the Map must match the input type of the Reduce function. In the wordcount example, the map function receives each line as a value, splits it into words, and emits (via the function context.out) a row for each word with frequency 1. Each invocation of the Reduce function receives for a given word the list of frequencies computed on the Map side. It adds these and emits each word and its frequency as output. The functions interact with a *context.* The context is used to interact with the framework. It is used by clients to send configuration information to tasks; and tasks can use it to get access to HDFS and read data directly from HDFS, to output key-value pairs, and to send status (e.g., task counters) back to the client.

The MapReduce way of implementing some other functions based on Dean and Ghemawat (2004) is as follows:

Distributed Grep

Grep looks for a given pattern in a file. The Map function emits a line if it matches a supplied pattern. The Reduce function is an identity function that copies the supplied intermediate data to the output. This is an example of a *Map only task*; there is no need to incur the cost of a **Shuffle**. We will provide more information when we explain the MapReduce runtime.

Reverse Web-Link Graph

The purpose here is to output (target URL, source URL) pairs for each link to a target page found in a page named source. The Reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair <target, list(source)>.

Inverted Index

The purpose is to build an inverted index based on all words present in a document repository. The Map function parses each document and emits a sequence of (word, document_id) pairs. The Reduce function takes all pairs for a given word, sorts them by document_id and emits a (word, list (document_id)) pair. The set of all these pairs forms an inverted index.

These illustrative applications give a sense of the MapReduce programming model's broad applicability and the ease of expressing the application's logic using the Map and Reduce phases.

A **Job** in MapReduce comprises the code for the Map and Reduce (usually packaged as a jar) phases, a set of artifacts needed to run the tasks (such as files, other jars, and archives) and, most importantly, a set of properties specified in a configuration. There are hundreds of properties that can be specified, but the core ones are as follows:

- the Map task
- the Reduce task

- the Input that the Job is to run on: typically specified as an HDFS path(s)
- the Format(Structure) of the Input
- the Output path
- the Output Structure
- the Reduce-side parallelism

A Job is submitted to the **JobTracker**, which then schedules and manages the execution of the Job. It provides a set of interfaces to monitor running Jobs. See the Hadoop Wiki[13] for further details about the workings of the JobTracker.

25.2.3 Hadoop Releases

Since the advent of Hadoop as a new distributed framework to run MapReduce programs, various releases have been produced:

The 1.x releases of Hadoop are a continuation of the original 0.20 code base. Subreleases with this line have added Security, additional HDFS and MapReduce improvements to support HBase, a better MR programming model, as well as other improvements.

The 2.x releases include the following major features:

- YARN (Yet Another Resource Navigator) is a general resource manager extracted out of the JobTracker from MR version1.
- A new MR runtime that runs on top of YARN.
- Improved HDFS that supports federation and increased availability.

At the time of this writing, Hadoop 2.0 has been around for about a year. The adoption is rapidly picking up; but a significant percentage of Hadoop deployments still run on Hadoop v1.

25.3 Hadoop Distributed File System (HDFS)

As we said earlier, in addition to MapReduce, the other core component of Hadoop is the underlying file system HDFS. In this section, we will first explain the architecture of HDFS, then describe the file input/output operations supported in HDFS, and finally comment on the scalability of HDFS.

25.3.1 HDFS Preliminaries

The Hadoop Distributed File System (HDFS) is the file system component of Hadoop and is designed to run on a cluster of commodity hardware. HDFS is patterned after the UNIX file system; however, it relaxes a few POSIX (portable operating system interface) requirements to enable streaming access to file system data. HDFS provides high-throughput access to large datasets. HDFS stores file system

[13]Hadoop Wiki is at http://hadoop.apache.org/

metadata and application data separately. Whereas the metadata is stored on a dedicated server, called the NameNode, the application data is stored on other servers, called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. To make data durable, the file content is replicated on multiple DataNodes, as in the Google File System. This not only increases reliability, but it also multiplies the bandwidth for data transfer and enables colocation of computation with data. It was designed with the following assumptions and goals:

Hardware failure: Using commodity hardware, failure of hardware is the norm rather than an exception. Therefore, with thousands of nodes, automatic detection and recovery from failures becomes a must.

Batch processing: HDFS has been primarily designed for batch rather than interactive use. High throughput is emphasized over low latency of data access. Full scans of files are typical.

Large datasets: HDFS was designed to support huge files in the hundreds of gigabytes to terabytes range.

Simple coherency model: HDFS applications need a one writer and many reader access models for files. File content cannot be updated, but only appended. This model alleviates coherency issues among copies of data.

25.3.2 Architecture of HDFS

HDFS has a master-slave architecture. The master server, called the **NameNode**, manages the file system storage area or namespace**;** Clients access the namespace through the Namenode. The slaves called **DataNodes** run on a cluster of commodity machines, usually one per machine. They manage the storage attached to the node that they run on. The namespace itself comprises Files and Directories. The Namenodes maintain *inodes* (index nodes) about File and Directories with attributes like ownership, permissions, creation and access times, and disk space quotas. Using inodes, the mapping of File blocks to DataNodes is determined. DataNodes are responsible for serving read and write requests from clients. DataNodes perform block creation, deletion, and replication operations as instructed by the NameNode. A cluster can have thousands of DataNodes and tens of thousands of HDFS clients simultaneously connected.

To read a file, a client first connects to the NameNode and obtains the locations of the data blocks in the file it wants to access; it then connects directly with the DataNodes that house the blocks and reads the data.

The architecture of HDFS has the following highlights:

1. HDFS allows a decoupling of metadata from data operations. Metadata operations are fast whereas data transfers are much slower. If the location of metadata and transfer of data are not decoupled, speed suffers in a distributed environment because data transfer dominates and slows the response.

2. Replication is used to provide reliability and high availability. Each block is replicated (default is three copies) to a number of nodes in the cluster. The highly contentious files like MapReduce job libraries would have a higher number of replicas to reduce network traffic.
3. The network traffic is kept to a minimum. For reads, clients are directed to the closest DataNode. As far as possible, a local file system read is attempted and involves no network traffic; the next choice is a copy on a node on the same rack before going to another rack. For writes, to reduce network bandwidth utilization, the first copy is written to the same node as the client. For other copies, travel across racks is minimized.

NameNode. The NameNode maintains an **image** of the file system comprising *i*-nodes and corresponding block locations. Changes to the file system are maintained in a Write-ahead commit log (see the discussion of Write-ahead logs in Chapter 22) called the **Journal**. Checkpoints are taken for purposes of recovery; they represent a persistent record of the image without the dynamic information related to the block placement. Block placement information is obtained from the DataNodes periodically as described below. During Restart, the image is restored to the last checkpoint and the journal entries are applied to that image. A new checkpoint and empty journal are created so that the NameNode can start accepting new client requests. The startup time of a NameNode is proportional to the Journal file's size. Merging the checkpoint with the Journal periodically reduces restart time.

Note that with the above architecture, it is catastrophic to have any corruption of the Checkpoint or the Journal. To guard against corruption, both are written to multiple directories on different volumes.

Secondary NameNodes. These are additional NameNodes that can be created to perform either the checkpointing role or a backup role. A Checkpoint node periodically combines existing checkpoint and journal files. In backup mode, it acts like another storage location for the Journal for the primary NameNode. The backup NameNode remains up-to-date with the file system and can take over on failure. In Hadoop V1, this takeover must be done manually.

DataNodes: Blocks are stored on a DataNode in the node's native file system. The NameNode directs clients to the DataNodes that contain a copy of the block they want to read. Each block has its representation in two files in the native file system: a file containing the data and a second file containing the metadata, which includes the checksums for the block data and the block's generation stamp. DataNodes and NameNodes do not communicate directly but via a so-called **heartbeat mechanism**, which refers to a periodic reporting of the state by the DataNode to the NameNode; the report is called a Block Report. The report contains the block id, the generation stamp, and the length for each block. The block locations are not part of the namespace image. They must be obtained from the block reports, and they change as blocks are moved around. The MapReduce Job Tracker, along with the

NameNode, uses the latest block report information for scheduling purposes. In response to a heartbeat from the DataNode, the NameNode sends one of the following types of commands to the DataNode:

- Replicate a block to another node.
- Remove a block replica.
- Reregister the node or shut down the node.
- Send an immediate block report.

25.3.3 File I/O Operations and Replica Management in HDFS

HDFS provides a single-writer, multiple-reader model. Files cannot be updated, but only appended. A file consists of blocks. Data is written in 64-KB packets in a **write pipeline**, which is set up to minimize network utilization, as we described above. Data written to the last block becomes available only after an explicit hflush operation. Simultaneous reading by clients is possible while data is being written. A checksum is generated and stored for each block and is verified by the client to detect corruption of data. Upon detection of a corrupt block, the Namenode is notified; it initiates a process to replicate the block and instructs the Datanode to remove the corrupt block. During the read operation, an attempt is made to fetch a replica from as close a node as possible by ordering the nodes in ascending order of distance from the client. A read fails when the Datanode is unavailable, when the checksum test fails, or when the replica is no longer on the Datanode. HDFS has been optimized for batch processing similar to MapReduce.

Block Placement. Nodes of a Hadoop cluster are typically spread across many racks. They are normally organized such that nodes on a rack share a switch, and rack switches are connected to a high-speed switch at the upper level. For example, the rack level may have a 1-Gb switch, whereas at the top level there may be a 10-Gb switch. HDFS estimates the network bandwidth between Datanodes based on their distance. Datanodes on the same physical node have a distance of 0, on the same rack are distance 2 away, and on different racks are distance 4 away. The default HDFS block placement policy balances between minimizing the write cost and maximizing data reliability and availability as well as aggregate read bandwidth. Network bandwidth consumed is estimated based on distance among DataNodes. Thus, for DataNodes on the same physical node, the distance is 0, whereas on the same rack it is 2 and on a different rack it is 4. The ultimate goal of block placement is to minimize the write cost while maximizing data availability and reliability as well as available bandwidth for reading. Replicas are managed so that there is at least one on the original node of the client that created it, and others are distributed among other racks. Tasks are preferred to be run on nodes where the data resides; three replicas gives the scheduler enough leeway to place tasks where the data is.

Replica Management. Based on the block reports from the DataNodes, the NameNode tracks the number of replicas and the location of each block. A replication priority queue contains blocks that need to be replicated. A background thread

monitors this queue and instructs a DataNode to create replicas and distribute them across racks. NameNode prefers to have as many different racks as possible to host replicas of a block. Overreplicated blocks cause some replicas to be removed based on space utilization of the DataNodes.

25.3.4 HDFS Scalability

Since we are discussing big data technologies in this chapter, it is apropos to discuss some limits of scalability in HDFS. Hadoop program management committee member Shvachko commented that the Yahoo HDFS cluster had achieved the following levels as opposed to the intended targets (Shvachko, 2010). The numbers in parentheses are the targets he listed. Capacity: 14 petabytes (vs. 10 petabytes); number of nodes: 4,000 (vs. 10,000); clients:15,000 (vs. 100,000); and files: 60 million (vs. 100 million). Thus, Yahoo had come very close to its intended targets in 2010, with a smaller cluster of 4,000 nodes and fewer clients; but Yahoo had actually exceeded the target with respect to total amount of data handled.

Some of the observations made by Shvachko (2010) are worth mentioning. They are based on the HDFS configuration used at Yahoo in 2010. We present the actual and estimated numbers below to give the reader a sense of what is involved in these gigantic data processing environments.

- The blocksize used was 128K, and an average file contained 1.5 blocks. NameNode used about 200 bytes per block and an additional 200 bytes for an *i*-node. 100 million files referencing 200 million blocks would require RAM capacity exceeding 60 GB.
- For 100 million files with size of 200 million blocks and a replication factor of 3, the disk space required is 60 PB. Thus a rule of thumb was proposed that 1 GB of RAM in NameNode roughly corresponds to 1 PB of data storage based on the assumption of 128K blocksize and 1.5 blocks per file.
- In order to hold 60 PB of data on a 10,000-node cluster, each node needs a capacity of 6 TB. This can be achieved by having eight 0.75-TB drives.
- The internal workload for the NameNode is block reports. About 3 reports per second containing block information on 60K blocks per report were received by the NameNode.
- The external load on the NameNode consisted of external connections and tasks from MapReduce jobs. This resulted in tens of thousands of simultaneous connections.
- The Client Read consisted of performing a block lookup to get block locations from the NameNode, followed by accessing the nearest replica of the block. A typical client (the Map job from an MR task) would read data from 1,000 files with an average reading of half a file each, amounting to 96 MB of data. This was estimated to take 1.45 seconds. At that rate, 100,000 clients would send 68,750 block-location requests per second to the NameNode. This was considered to be well within the capacity of the NameNode, which was rated at handling 126K requests per second.