Practical Hadoop Ecosystem

A Definitive Guide to Hadoop-Related Frameworks and Tools

Deepak Vohra

Foreword by John Yeary

Apress®

Practical Hadoop Ecosystem

A Definitive Guide to Hadoop-Related Frameworks and Tools

Deepak Vohra

Apress®

Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools

Deepak Vohra

ISBN-13 (pbk): 978-1-4842-2198-3 ISBN-13 (electronic): 978-1-4842-2199-0
DOI 10.1007/978-1-4842-2199-0

Library of Congress Control Number: 2016954636

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewers: John Yeary and Simon Bisson
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Contents at a Glance

Contents

About the Author

Deepak Vohra is a consultant and a principal member of the NuBean.com software company. Vohra is a Sun-certified Java programmer and web component developer. He has worked in the fields of XML, Java programming, and Java EE for over seven years. Vohra is the coauthor of *Pro XML Development with Java Technology* (Apress, 2006). He is also the author of the *JDBC 4.0* and *Oracle JDeveloper for J2EE Development, Processing XML Documents with Oracle JDeveloper 11g, EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g,* and *Java EE Development in Eclipse IDE* (Packt Publishing). He also served as the technical reviewer on *WebLogic: The Definitive Guide (O'Reilly Media, 2004)* and *Ruby Programming for the Absolute Beginner* (Cengage Learning PTR, 2007).

About the Technical Reviewer

John Yeary is a Principal Software Engineer on Epiphany CRM Marketing at Infor Global Solutions. John has been a Java evangelist and has been working with Java since 1995. Yeary is a technical blogger with a focus on Java Enterprise Edition technology, NetBeans, and GlassFish. He is currently the Founder of the Greenville Java Users Group (GreenJUG). He is as instructor, mentor, and a prolific open source contributor.

John graduated from Maine Maritime Academy with a B.Sc. Marine Engineering with a concentration in mathematics. He is a merchant marine officer and has a number of licenses and certifications. When he is not doing Java and F/OSS projects, he likes to hike, sail, travel, and spend time with his family. Yeary is also the Assistant Scoutmaster in the Boy Scouts of America (BSA) Troop 767, and Outdoor Ethics Master Trainer in the Blue Ridge Council of the BSA.

Organizations and projects:

- Java User Groups Community Leader (`Java.net`)
- Java Enterprise Community Leader (`Java.net`)
- JavaOne Technical Content Reviewer: Java Tools and Emerging Languages (Four Years)
- JavaOne Technical Content Reviewer: Java EE Platform and Frameworks (Three Years)
- JavaOne 2011, 2013 Planning Committee
- Duke's Choice Award Selection Committee (2010-2013)
- JavaOne Speaker (2009-2013)
- JCP Member
- Project Woodstock Committer and Maintainer (`http://java.net/projects/woodstock`)
- Java Boot Camp Project Owner (`http://java.net/projects/certbootcamp`)

Foreword

I want to welcome you to the world of Hadoop. If you are novice or an expert looking to expand your knowledge of the technology, then you have arrived at the right place. This book contains a wealth of knowledge that can help the former become the latter. Even most experts in a technology focus on particular aspects. This book will broaden your horizon and add valuable tools to your toolbox of knowledge.

When Deepak asked me to write the foreword, I was honored and excited. Those of you who know me usually find I have no shortage of words. This case was no exception, but I found myself thinking more about what to say, and about how to keep it simple.

Every few years, technology has a period of uncertainty. It always seems we are on the cusp of the next "great" thing. Most of the time, we find that it is a fad that is soon replaced by the next shiny bauble. There are some moments that have had an impact, and some that leave the community guessing. Let's take a look at a couple of examples to make a point.

Java appeared like manna from the heavens in 1995. Well, that is perhaps a bit dramatic. It did burst on to the scene and made development easier because you didn't need to worry about memory management or networking. It also had this marketing mantra, which was "write once, run anywhere". It turned out to be mostly true. This was the next "great" thing.

Rolling ahead to 1999 and the release of J2EE. Again, we encounter Java doing all the right things. J2EE technologies allowed, in a standard way, enterprises to focus on business value and not worry about the technology stack. Again, this was mostly true.

Next we take a quantum leap to 2006. I attended JavaOne 2005 and 2006 and listened to numerous presentations of where J2EE technology was going. I met a really passionate developer named Rod Johnson who was talking about Spring. Some of you may have heard of it. I also listened as Sun pushed Java EE 5, which was the next big change in the technology stack. I was also sold on a new component-based web UI framework called Woodstock, which was based on JavaServer Faces. I was in a unique position; I was in charge of making decisions for a variety of business systems at my employer at the time. I had to make a series of choices. On the one hand I could use Spring, or on the other, Java EE 5. I chose Java EE 5 because of the relationships I had developed at Sun, and because I wanted something based on a "standard". Woodstock, which I thought was the next "great" thing, turned out to be flash in the pan. Sun abandoned Woodstock, and well… I guess on occasion I maintain it along with some former Sun and Oracle employees. Spring, like Java EE 5, turned out to be a "great" thing.

Next was the collapse of Sun and its being acquired by the dark side. This doomsday scenario seemed to be on everyone's mind in 2009. The darkness consumed them in 2010. What would happen to Java? It turned out everyone's initial assessment was incorrect. Oracle courted the Java community initially, spent time and treasure to fix a number of issues in the Java SE stack, and worked on Java EE as well. It was a phenomenal wedding, and the first fruits of the union were fantastic—Java SE 7 and Java EE 7 were "great". They allowed a number of the best ideas to become reality. Java SE 8, the third child, was developed in conjunction with the Java community. The lambda, you would have thought, was a religious movement.

While the Java drama was unfolding, a really bright fellow named Doug Cutting came along in 2006 and created an Apache project called Hadoop. The funny name was the result of his son's toy elephant. Today it literally is the elephant in the room. This project was based on the Google File System and Map Reduce. The baby elephant began to grow. Soon other projects with cute animal names like Pig, or more even more apt, Zookeeper, came along. The little elephant that could soon was “the next great thing”.

Suddenly, Hadoop was the talk of the Java community. In 2012, I handed the Cloudera team a Duke’s Choice Award for Java technology at JavaOne. Later that year, version 1.0 was released. It was the culmination of hard work for all the folks who invested sweat and tears to make the foundation of what we have today.

As I sit here wondering about Java EE 8 and its apparent collapse, I am reminded that there is still innovation going on around me. The elephant in the room is there to remind me of that.

Some of you may be wondering what Hadoop can do for you. Let’s imagine something fun and how we might use Hadoop to get some answers. Where I live in South Carolina, we have something called a Beer BBQ 5K. It is a 5K run that includes beer and BBQ at the end, along with music. Some folks will just do the beer and BBQ. That is fine, but in my case I need to do the run before. So we have data coming in on the registration; we have demographic data like age and gender. We have geographic data: where they call home. We have timing data from the timing chips. We have beer and BBQ data based on wristband scans. We have multiple races in a year.

Hmm… what can we do with that data? One item that comes to mind is marketing, or planning. How many women in which age groups attended and what beer did they drink? How many men? Did the level of physical activity have any effect on the consumption of BBQ and beer? Geographically, where did attendees come from? How diverse were the populations? Do changing locations and times of the year have different effects? How does this compare with the last three years? We have incomplete data for the first year, and the data formats have changed over time. We have become more sophisticated as the race and the available data have grown. Can we combine data from the race with publicly accessible information like runner tracking software data? How do we link the data from a provider site with our data?

Guess what? Hadoop can answer these questions and more. Each year, the quantity of data grows for simple things like a Beer BBQ 5K. It also grows in volumes as we become more connected online. Is there a correlation between Twitter data and disease outbreak and vector tracking? The answer is yes, using Hadoop. Can we track the relationship between terrorists, terrorist acts, and social media? Yes, using…. well, you get the point.

If you have read this far, I don’t think I need to convince you that you are on the right path. I want to welcome you to our community, and if you are already a member, I ask you to consider contributing if you can. Remember “a rising tide raises all boats,” and you can be a part of the sea change tide.

The best way to learn any technology is to roll up your sleeves and put your fingers to work. So stop reading my foreword and get coding!

—John Yeary
NetBeans Dream Team
Founder Greenville Java Users Group
Java Users Groups Community Leader
Java Enterprise Community Leader

PART I

Fundamentals

CHAPTER 1

Introduction

Apache Hadoop is the de facto framework for processing and storing large quantities of data, what is often referred to as "big data". The Apache Hadoop ecosystem consists of dozens of projects providing functionality ranging from storing, querying, indexing, transferring, streaming, and messaging, to list a few. This book discusses some of the more salient projects in the Hadoop ecosystem.

Chapter 1 introduces the two core components of Hadoop—HDFS and MapReduce. Hadoop Distributed Filesystem (HDFS) is a distributed, portable filesystem designed to provide high-throughput streaming data access to applications that process large datasets. HDFS's main benefits are that it is fault-tolerant and designed to be run on commodity hardware. Hadoop MapReduce is a distributed, fault-tolerant framework designed for processing large quantities of data stored in HDFS, in parallel on large clusters using commodity hardware.

Chapter 2 introduces Apache Hive, which is a data warehouse for managing large datasets stored in HDFS. Hive provides a HiveQL language, which is similar to SQL but does not follow the SQL-92 standard fully, for querying the data managed by Hive. While Hive can be used to load new data, it also supports projecting structure onto data already stored. The data could be stored in one of the several supported formats, such as Avro, ORC, RCFile, Parquet, and SequenceFile. The default is TextFile.

Chapter 3 introduces the Hadoop database called Apache HBase. HBase is a distributed, scalable NoSQL data store providing real-time access to big data; NoSQL implies that HBase is not based on the relational data model. HBase stores data in HDFS, thus providing a table abstraction for external clients. An HBase table is unlike a relational database table in that it does not follow a fixed schema. Another difference of RDBMS is the scale of data stored; an HBase table could consist of millions of rows and columns. As with the other Hadoop ecosystem projects, HBase can run on clusters of commodity software.

Chapter 4 introduces Apache Sqoop, a tool for bulk transfer of data between relational databases such as Oracle database and MySQL database and HDFS. Sqoop also supports bulk transfer of data from RDBMS to Hive and HBase.

Chapter 5 introduces Apache Flume, a distributed, fault-tolerant framework for collecting, aggregating, and streaming large datasets, which are typically log data, but could be from other data sources such as relational databases.

Chapter 6 discusses Apache Avro, a schema based data serialization system providing varied data structures. Avro provides a compact, fast, binary data format typically used to store persistent data.

Chapter 7 discusses another data format called Apache Parquet. Parquet is a columnar data storage format providing complex data structures and efficient compression and encoding on columnar data to any Hadoop ecosystem project.

Chapter 8 introduces Apache Kafka, a distributed publish-subscribe messaging system that is fast, scalable, and durable. As with the other Hadoop ecosystem projects, Kafka is designed to process large quantities of data and provide high throughput rates.

Chapter 9 discusses another distributed, scalable, and fault-tolerant framework called Apache Solr. Solr provides indexing and data integration for large datasets stored in HDFS, Hive, HBase, or relational databases. Solr is Lucene-based and one of the most commonly used search engines.

D. Vohra, *Practical Hadoop Ecosystem*, DOI 10.1007/978-1-4842-2199-0_1

Chapter 10 introduces Apache Mahout, a framework for machine learning applications. Mahout supports several machine-learning systems, such as classification, clustering, and recommender systems.

The different Apache Hadoop ecosystem projects are correlated, as shown in Figure 1-1. MapReduce processes data stored in HDFS. HBase and Hive store data in HDFS by default. Sqoop could be used to bulk transfer data from a relational database management system (RDBMS) to HDFS, Hive, and HBase. Sqoop also supports bulk transfer of data from HDFS to a relational database. Flume, which is based on sources and sinks, supports several kinds of sources and sinks with the emphasis being on streaming data in real time in contrast to one-time bulk transferring with Sqoop. Flume is typically used to stream log data and the sink could be HDFS, Hive, HBase, or Solr, to list a few. Solr could be used to index data from HDFS, Hive, HBase, and RDBMS. HDFS stores data in a disk filesystem and is in fact an abstract filesystem on top of the disk filesystem. Solr also stores data in a disk filesystem by default, but can also store the indexed data in HDFS.

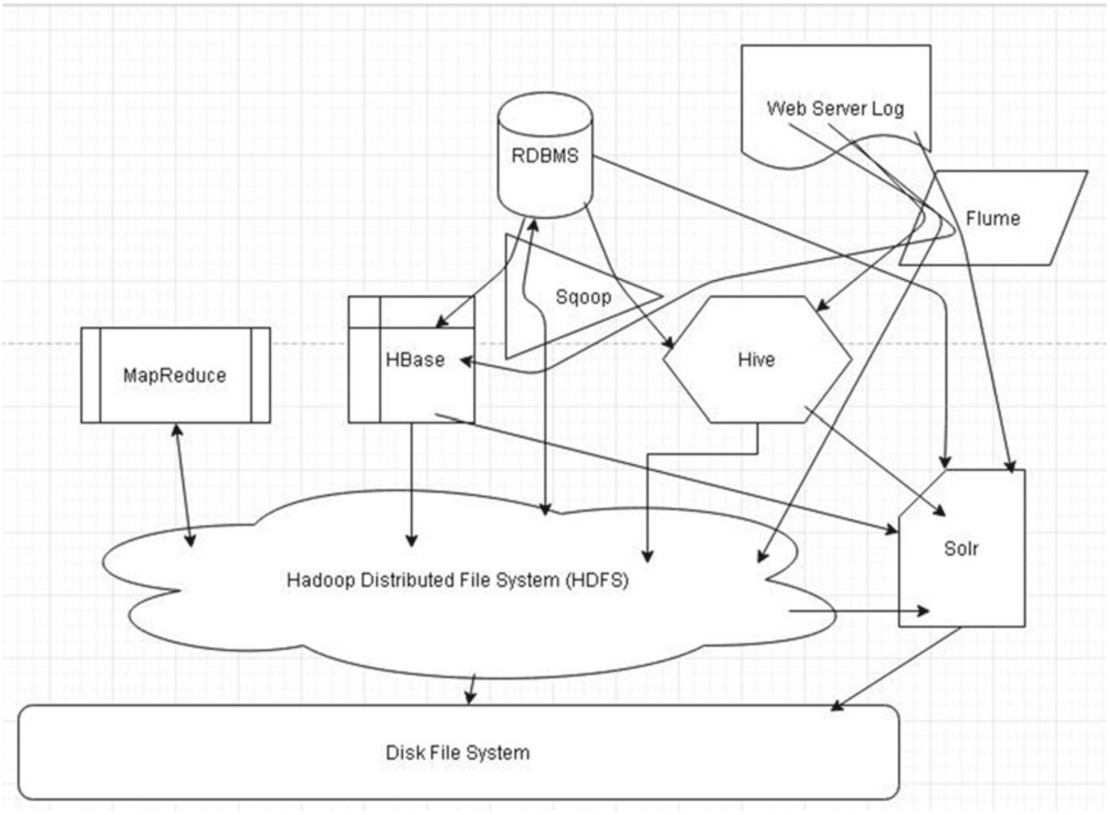

Figure 1-1. *The Apache Hadoop ecosystem*

Next, we introduce some of the concepts used by Apache Hadoop's core components HDFS and MapReduce, and discuss why Hadoop is essential for web-scale data processing and storage.

Core Components of Apache Hadoop

Hadoop has two main components: the Hadoop Distributed Filesystem (HDFS) and MapReduce. The HDFS is a data storage and data processing filesystem. HDFS is designed to store and provide parallel, streaming access to large quantities of data (up to 100s of TB). HDFS storage is spread across a cluster of nodes; a single large file could be stored across multiple nodes in the cluster. A file is broken into blocks, which is an abstraction over the underlying filesystem, with default size of 64MB. HDFS is designed to store large files and lots of them.

MapReduce is a distributed data processing framework for processing large quantities of data, distributed across a cluster of nodes, in parallel. MapReduce processes input data as key/value pairs. MapReduce is designed to process medium-to-large files. MapReduce has two phases: the map phase and the reduce phase. The map phase uses one or more mappers to process the input data and the reduce phase uses zero or more reducers to process the data output during the map phase. The input files is converted to key/value pairs before being input to the map phase. During the map phase, input data consisting of key/value pairs is mapped to output key/value pairs using a user-defined `map()` function. The map output data is partitioned and sorted for input to the reduce phase. The map output data is partitioned such that values associated with the same key are partitioned to the same partition and sent to the same reducer. During the reduce phase, the data output from the map phase is processed to reduce the number of values associated with a key, or using some other user-defined function. The Google implementation of the MapReduce programming model is discussed at http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

Why Apache Hadoop?

Large-scale computation models have been necessitated by the increasing scale of datasets being generated and processed. Examples of big data scenarios include large-scale market data being generated for marketable products, consumer preferences data, market trends data, social media data, and search engine data.

Shortcomings in Current Computing Systems

Different distributed systems overcome the challenges to large-scale computational models differently, prioritizing some issues more than others, and in the process making trade-offs.

NFS (Network FileSystem) is the most commonly used distributed filesystem. The design of NFS is very constrained; it provides remote access to a single logical volume on a single machine. A client has access to only a portion of the NFS filesystem, which the client can mount on its local filesystem and access as if it were a local filesystem. One of the main advantages of NFS is that the filesystem is transparent to the client. NFS has the following disadvantages:

- The data on NFS is stored on a single machine. If the machine fails, the entire filesystem becomes unavailable.
- All clients access the same remote filesystem, which could overload the NFS.
- Clients must copy the data to the local filesystem before being able to use the data.

When designing a large-scale system, these basic assumptions that apply to traditional systems have to be disregarded:

- Hardware must be reliable.
- Machines have identities with explicit communication between them.

- A dataset can be stored on a single machine.
- All data is structured data.

The scale of data involved in big data scenarios is many orders of magnitude larger than in traditional computational models. Whereas traditional computational models process data in the range of a few MB to 100s of MB, large-scale computational models process data in the range of 100s of GB to 100s of TB and even several PB. At such a large scale, the input data won't fit into a single machine's memory or disk drive. Individual machines have limited resources in terms of CPU processor time, RAM, hard disk space, and network bandwidth. An individual dataset won't fit into a single machine and won't be able to be processed on a single machine. A single disk drive won't be able to store even the temporary output from processing large quantities of data, much less store the input data. With multi-gigabyte datasets being transmitted, network switches will become saturated and the available network bandwidth won't be sufficient for the large datasets. With machines on different racks, the network bandwidth between machines on different racks would be less than the network bandwidth between nodes on the same rack. A large-scale computational model must be able to manage the resources efficiently. Additional resources may be required to implement features designed for a large-scale model. For example, if data is to be replicated for durability, additional disk space becomes a requirement. Synchronizing between multiple machines is an issue unique to large-scale computational models. Efficient network communication protocols have to be used to communicate between the different nodes on a large-scale cluster.

Failure recovery has to be implemented to recover from partial failures. For example, if a machine fails, the processing of a section of dataset fails. The computation has to be restarted on a different machine, thus incurring some computation time loss. The computation capacity, proportional to the number of machines lost, is lost. Failure in some of the components changes the network topology and starting recomputation in the event of a failure may not be trivial. Some of the related computation previously completed successfully but not yet used may also be lost in the event of a machine failure.

Data durability is an issue in a large-scale computational model. If a machine is lost, the data stored on the machine is also lost. If replicas of the lost data are available, the data is recoverable, but if no replicas are available, the data is not durable.

With large datasets, the probability of some sections of datasets getting corrupted increases. Data corruption detection has to be implemented to provide data reliability and data availability. If data corruption is not detected, computation could be erroneous. If data corruption is not addressed, some data may be lost and not available for computation.

Large-scale computation requires that the input data be distributed across multiple machines and processed in parallel on different machines. One of the results of using multiple machines is that the probability of failure increases; the more machines there are in the cluster, the more likely one of those machines will fail. Failures could get cascaded. Machine failure and program recovery is an issue in large-scale models only, as with a single-machine model recovery of the program is not an option; if the machine fails, the program fails.

Failure of some of the components in a large-scale system is inevitable and the failure is also likely to happen more often. Examples of partial failures include failure of network switches and routers, high network traffic, hard drive failure on individual machines, running out of memory and disk space, and data corruption. In a large-scale computational model, the rest of the components are expected to continue to function.

Scalability of a large-scale computation model is another issue. If more and more computational units are added, as in a distributed system, the computational model should scale well.

How Is Hadoop Better than Other Distributed Computing Systems?

A distributed system in the context of data computation is a system in which computation is distributed across a set of machines instead of using a single machine. Why are distributed systems required? If the scale of data has increased, why can't the capacities of individual machines also increase proportionally? New processors with multi-CPU cores (dual-core, quad-core, or more) provide multiple threads to process more data in parallel, but the processor speed diminishes slightly, due to thermal requirements, with an increase in the number of CPU cores per processor. For example, a dual-core processor may have a speed of 3.5GHz, while a quad-core processor has 3.0GHz. Even if a single-machine with thousands of CPU cores is built, it would not be feasible for individual hard drives to read data fast enough to be processed in parallel on the multi-CPU core processors. An individual hard disk drive has a limited read speed in the range of 60-100MB/sec. For example, if a 12-core processor is used on a single machine with multiple I/O channels, data is input to the machine at the rate of 1200MB/sec (assuming an upper range rate). A 10TB dataset would take 2.3 hours to read the data. If 100 similar machines are used, the 10TB is read in 1.38 minutes. This demonstrates the advantage of using multiple machines in a distributed system.

Grid computing is not new. Other distributed grid systems such as MPI, PVM, and Condor have been used in the past. The grid emphasis is to distribute the workload. Data stored in NetApp filer or SAN drives used in conjunction with several compute nodes. Sharing is slow on most of the single-storage based distributed systems. In contrast, Hadoop distributes the data on several nodes instead of storing it in a single file.

Some reliability demands on distributed systems that support large-scale computational models are:

- *Support partial failure.* Must support graceful decline in application performance rather than a full halt.
- *Data recoverability.* If components fail, their workload must be taken up by the functioning machines.
- *Individual recoverability.* A full restart should not be required. If a node fails, it should be able to restart and rejoin the cluster.
- *Consistency*. Concurrent operations or partial failures should not cause externally visible nondeterminism.
- *Scalability*. The distributed system should be able to take an increased load without performance degradation or complete failure. A graceful decline in performance is still preferred over complete failure. An increase in the number of machines should provide a proportionate increase in the capacity.

Hadoop is a large-scale distributed processing system designed for a cluster consisting of hundreds or thousands of nodes, each with multi-processor CPU cores. Hadoop is designed to distribute and process large quantities of data across the nodes in the cluster. Hadoop does not require any special hardware and is designed for commodity hardware. Hadoop may be used with any type of data, structured or unstructured. Hadoop is not a database, nor does Hadoop replace traditional database systems. Hadoop is designed to cover the scenario of storing and processing large-scale data, which most traditional systems do not support or do not support efficiently. Hadoop can join and aggregate data from many different sources and deliver results to other enterprise systems for further analysis.

Hadoop is designed to process web-scale data in the order of hundreds of GB to 100s of TB, even to several PB. Hadoop provides a distributed filesystem that breaks up the input data and distributes the data across several nodes in a cluster. Hadoop processes the distributed data in parallel using all the machines (nodes) in the cluster. Data is processed on the same node as the data is stored if feasible, as a result providing data locality. Not having to move large amounts of data for computation reduces the network bandwidth usage. Hadoop is designed for commodity hardware in which component failure is expected rather than an exception, and Hadoop provides automatic detection and recovery of failure and other design implementations such as replicating data across the cluster for durability. If one machine fails, a data replica on another machine can be used.

Considering the differences in the network bandwidth between nodes on the same rack and nodes on different racks, Hadoop uses a rack-aware placement policy when creating a new file, reading from a file, or processing input data.

To make data durable, Hadoop replicates stored data. If a machine fails and the data on the machine is lost, a replica of the data is used, and the data is re-replicated to its replication level. Making redundant copies of data adds to the disk space requirements, and as a result only a fraction of disk space is actually being used to store a single copy of the data. For data reliability, Hadoop uses checksum verification on the data stored in its filesystem. If corrupt data is found, it is not used and a replica of the data is used instead. Corrupt data replicas are replaced with non-corrupt data and the corrupt data is removed from the filesystem. A data scanner runs periodically and automatically to scan for corrupt data.

Hadoop provides recovery in the event of computation failure. A failed computation is performed again, sometimes even having to re-compute related computation. If some of the disk drives fail, computation may be able to continue with the other disk drives.

Other distributed systems such as HTCondor (formerly Condor) also provide a high-throughput, parallel computational model. Hadoop is unique in its simplified programming model (MapReduce), which allows quick development and running of distributed applications. Hadoop is also unique in its ability to efficiently and automatically distribute data and computation code across the multiple machines in the cluster, as a result utilizing the parallelism provided by multi-CPU cores effectively. HTCondor does not distribute data automatically and a separate SAN (Storage Area Network) is required for the purpose. While the programming model in Hadoop does not require communication between independent processes running in parallel on a subset of the dataset, in HTCondor communication between multiple commute nodes must be managed with a communication system such as MPI (Message Passing Interface). Individual nodes in Hadoop also communicate with each other, but the communication is implicit in contrast to the more traditional distributed systems in which data has to be marshaled explicitly over sockets or through MPI buffers using application code. In Hadoop, explicit communication between nodes is not permitted. In Hadoop, partitioning of data is implemented using a built-in partitioner and shuffling of data between different phases of data processing is also implemented implicitly without user-defined application code.

Individual machines in Hadoop are more independent of each other than in other distributed systems, which makes failure recovery more favorable, since a failure of a single machine does not affect other machines in the distributed system. Multiple user programs do not need to communicate with each other and failure of a process does not affect other processes running in parallel on other machines (in machine failure) or even the same machine (in process failure).

Hadoop provides a linear scalability as the scale of data stored and processed increases and as additional resources are added. A Hadoop application developed for a 12-machine cluster can be scaled nearly linearly to hundreds and thousands of machines without much modification. The performance of the Hadoop platform does not degrade with an increase in the scale of data or an increase in the number of machines.

Other distributed programming systems, such as HTCondor, MPI, and PVM, may perform better or comparably at a small scale, but do not scale well with an increased dataset load or increased number of machines. For example, MPI performs better than Hadoop with a small number of machines up to 12, but if the number of machines is increased to several tens or hundreds and the data processed is increased, a lot of refactoring such as modifying the application program is required. The performance of other distributed systems such as MPI degrades (or does not increase linearly) at a large scale. Other distributed systems may also have some design limitations at a large scale, which makes their scaling limited.

While MapReduce MRv1 is designed to run only MapReduce applications, YARN (Yet Another Resource Negotiator) or MRv2 supports running other applications besides MapReduce. Motivations for MapReduce are data processing of large datasets (> 1TB) , massively parallel (hundreds or thousands of CPUs), and easy to use without having to code communication between nodes. The reduce phase does not start until the map phase has completed, which could put a limit on the job progress rate if a few of the map processes are slow. If some slow map phase processes are slowing down, the whole job the master daemon runs redundant copies of slow moving processes and uses the results from the redundant process that completes first. In MapReduce, data processing is divided into two phases: the map phase and the reduce phase. The only communication between different processes is copying the output from the map phase to the reduce phase. The master daemon assigns tasks based on location of data; the map phase tasks run on the same machine (or the same rack) as the machine with the data. Map task input is divided into input splits, which are sized based on the block size. Individual input splits are processed by separate processes in parallel. Processes are independent of each other. The master daemon detects failure in a process and reruns failed processes. Restarting processes does not require communication with other processes. MapReduce is functional programming with distributed computing. MR factors out reliability concerns from application logic. With MapReduce, a user or developer does not have to implement system level details of synchronization, concurrency, hardware failure, inter-node communication, process failure, and failure recovery.

What Kind of Computations Is Hadoop Suitable For?

Hadoop is suitable for iterative jobs such as graph algorithms. Each iteration must read or write data to disk. I/O and latency cost of an iteration is high.

What Kind of Computations Is Hadoop Not Suitable For?

Hadoop is not suitable for:

- Applications that need shared state or coordination. MapReduce tasks are independent and are shared-nothing. Shared state requires scalable state store.
- Low-latency applications.
- Small datasets.
- Finding individual records.

HDFS Daemons

The HDFS daemons are the `NameNode`, `Secondary NameNode`, and `DataNode`.

NameNode

The `NameNode` is the master daemon in the HDFS. `NameNode`'s function is to maintain the HDFS namespace metadata, which includes the filenames, directory names, file permissions, directory permissions, file-to-block mapping, block IDs, and block locations in RAM. The metadata is kept in RAM for fast access. `NameNode` stores the metadata information in a `fsimage` file in the `NameNode`'s local filesystem. The stored namespace state does not include the block locations. The block locations are kept only in memory and when the `NameNode` starts, the block locations are constructed from the block lists sent by `DataNodes` when they join the `NameNode` and also in periodic reports. `NameNode` does not directly read or write to the HDFS. When a client requests a file to read, the `NameNode` provides the client with the block locations of the file and the client reads the file directly. When a client creates a new file, the `NameNode` provides the client with a list of block locations ordered by distance from the client and the client writes to the blocks directly.

`NameNode` also keeps an edit log of all the transactions made in the HDFS namespace that would alter the namespace, such as creating a file, deleting a file, and creating block replicas. The edits log is also stored by the `NameNode` local filesystem as `EditLog` file. The modifications information in `NameNode` RAM is flushed to the edit log on the `NameNode` disk periodically. When the `NameNode` starts, it starts in Safe mode, during which the edit log is applied to the image file `fsimage` to create a new `fsimage` file, which is stored. The `NameNode` exits the safe mode and starts with an empty edits file. The edit log is also check-pointed periodically by the `Secondary NameNode`.

In CDH5, the NameNode namespace URI is configured in the configuration property fs.defaultFS in core-default.xml.

```
<property>
 <name>fs.defaultFS</name>
 <value> hdfs://<namenode host>:<namenode port>/</value>
</property>
```

Another important property that must be configured is dfs.permissions.superusergroup in hdfs-site.xml. It sets the UNIX group whose users are to be the superusers for HDFS.

```
<property>
 <name>dfs.permissions.superusergroup</name>
 <value>hadoop</value>
</property>
```

The NameNode stores the namespace metadata fsimage file and the edit log file in directories configured using the dfs.namenode.name.dir property in hdfs-site.xml. At least two directories are recommended to be configured for the NameNode, preferably one on the NFS mount.

```
<property>
 <name>dfs.namenode.name.dir</name>
 <value>/data/1/dfs/nn,/nfsmount/dfs/nn</value>
</property>
```

The dfs.namenode.http-address on the NameNode specifies the HTTP address for the NameNode web UI to listen on.

```
<property>
  <name>dfs.http.address</name>
  <value>0.0.0.0:50070</value>
</property>
```

If HDFS high availability is not used, the NameNode is a single point of failure (SPOF) in the HDFS. The failure of the NameNode causes the HDFS become unavailable until the NameNode is restored. If HDFS high availability is not used, a Secondary NameNode is typically used.

Secondary NameNode

Since all the file modifications that alter the HDFS namespace are kept in the edits log, which is check-pointed by the NameNode only once at startup, the edits file could get very large during the operation of NameNode and take up extra disk space. Another disadvantage of a large edits log file is that when the NameNode starts and checkpoints the edits log to the fsimage file, the check-pointing could take a long time, resulting in delay in the NameNode starting in functional mode.

The Secondary NameNode's function is to make periodic checkpoints of the edits log to the fsimage file while the NameNode is running. When an edits log file has been check-pointed, it is cleared and as result occupies less disk space on the NameNode. With periodic check-pointing of the edits log, the edits log does not grow in size as much. When NameNode starts, it does not take as much time to checkpoint the edits log to the fsimage image, as the edits log is relatively smaller. As a result, the NameNode startup is faster.

The Secondary NameNode makes a new checkpoint using an interval configured in the hdfs-default.xml property dfs.namenode.checkpoint.period, which has a default setting of 3600 secs (1 hour). Another checkpoint setting that overrides dfs.namenode.checkpoint.period is dfs.namenode.checkpoint.txns, which specifies the number of transactions (the default value is 1,000,000 transactions) after which a checkpoint must be made regardless of whether the dfs.namenode.checkpoint.period interval has been exceeded or not. Secondary NameNode polls the NameNode periodically to find the number of uncheck-pointed transactions, as configured in dfs.namenode.checkpoint.check.period; the default is every 60 seconds. When a checkpoint is to be made, Secondary NameNode downloads the fsimage file and the edits log from the NameNode and stores the temporary copies of the two in two sets of directories configured by the dfs.namenode.checkpoint.dir property and the dfs.namenode.checkpoint.edits.dir property respectively in hdfs-default.xml.

The default value of both of these configuration properties is file://${hadoop.tmp.dir}/dfs/namesecondary. The number of copies of the fsimage file and the edits log the Secondary NameNode keeps is configured in the dfs.namenode.num.checkpoints.retained property; the default is two copies. The maximum network bandwidth for image transfer between the NameNode and the Secondary NameNode may be limited using the dfs.image.transfer.bandwidthPerSec property in hdfs-default.xml. A value of 0 (the default) implies that throttling is disabled. The image transfer bandwidth is limited so that the NameNode can carry out its normal operation. The image transfer may be timed out using the dfs.image.transfer.timeout property with a default value of 600000ms (10 minutes). Secondary NameNode applies the checkpoint and uploads the fsimage file back to the NameNode, as shown in Figure 1-2.

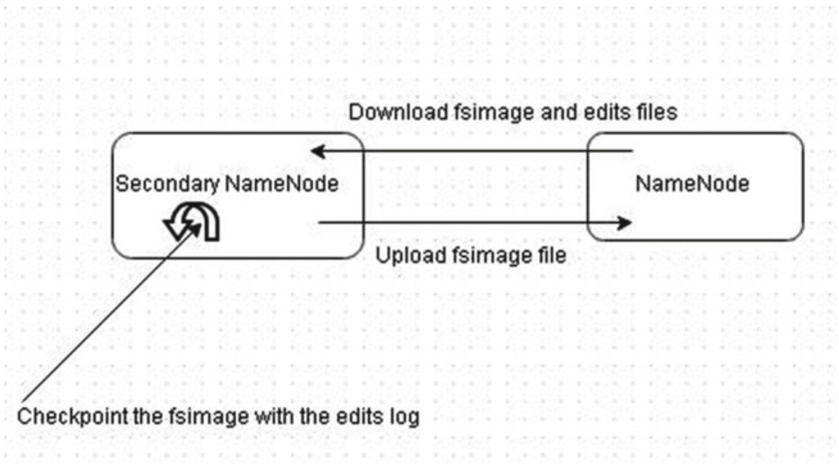
Download fsimage and edits files
Secondary NameNode
NameNode
Upload fsimage file
Checkpoint the fsimage with the edits log

Figure 1-2. *Check-pointing by secondary NameNode*

As mentioned, a NameNode applies a checkpoint of the edits log to the fsimage file on NameNode startup. The checkpoint on NameNode startup is based on the number of transactions stored in the edits log. The number of transactions stored in the edits log are not exactly those that represent previously uncheck-pointed transactions. Some extra transactions are stored as configured in the dfs.namenode.num.extra.edits.retained property, the default extra transactions being 1000000. Another configuration property that affects extra edit logs stored is the dfs.namenode.max.extra.edits.segments.retained, which specifies the number of extra edit log segments stored. The default is 10000. If the number of extra transactions as set in dfs.namenode.num.extra.edits.retained makes the extra edit segments exceed the setting in dfs.namenode.max.extra.edits.segments.retained, the value set in the latter is used to limit the number of extra transactions stored.

Secondary NameNode check-pointing could fail when loading the fsimage file or when applying the edits log. If the Secondary NameNode checkpoint fails, it retries *x* number of times as configured in dfs.namenode.checkpoint.max-retries. The default is three retries.

A cluster can have only one NameNode and one Secondary NameNode. On a production cluster, the Secondary NameNode should be located on a different node than the NameNode for two reasons:

- The Secondary NameNode on a separate node does not take up the much-needed RAM required by the NameNode.
- The Secondary NameNode on a separate node makes the HDFS more fault-tolerant because if the NameNode fails, the fsimage file can be re-created from the copy stored on the Secondary NameNode node.

To configure the Secondary NameNode, add the name of the Secondary NameNode host to the conf/slaves file and set a NameNode address for the Secondary NameNode to connect to for check-pointing. Use the following setting for dfs.namenode.http-address on the Secondary NameNode node.

```
<property>
  <name>dfs.namenode.http-address</name>
  <value><namenode.host.address>:50070</value>
</property>
```

If the NameNode loses all copies of the fsimage file and edits log, it may import the latest checkpoint stored in the dfs.namenode.checkpoint.dir directory using the following command.

```
hadoop namenode  -importCheckpoint
```

An empty dfs.namenode.name.dir directory is required to import the checkpoint prior to running the preceding command.

DataNodes

The DataNode is a slave daemon for storing the HDFS data. Data is broken into blocks and stored on the DataNodes. The blocks are replicated using a replication factor, as configured in the dfs.replication property in hdfs-default.xml; the default replication is 3. The block size is configured in the dfs.blocksize property in hdfs-default.xml, with the default value being 128MB. The HDFS blocks are an abstraction over the underlying filesystem of the DataNode. The directories in which the DataNode stores the blocks are configured in the dfs.datanode.data.dir property in hdfs-default.xml.

```
<property>
 <name>dfs.datanode.data.dir</name>
 <value>/data/1/dfs/dn,/data/2/dfs/dn,/data/3/dfs/dn</value>
</property>
```

By default, if any of the data volumes (directories) fails, the DataNode shuts down, but a certain number of failed volumes may be tolerated using the dfs.datanode.failed.volumes.tolerated property, which has a default value of 0.

```
<property>
    <name>dfs.datanode.failed.volumes.tolerated</name>
    <value>1</value>
    <final>true</final>
  </property>
```