



Practical Hadoop Ecosystem

A Definitive Guide to Hadoop-Related
Frameworks and Tools

Deepak Vohra

Foreword by John Yearly

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

Copyright © 2016 by Deepak Vohra

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

About the Authorxv

About the Technical Reviewerxvii

Forewordxix

■ Part I: Fundamentals 1

■ Chapter 1: Introduction 3

■ Chapter 2: HDFS and MapReduce 163

■ Part II: Storing & Querying 207

■ Chapter 3: Apache Hive 209

■ Chapter 4: Apache HBase 233

■ Part III: Bulk Transferring & Streaming 259

■ Chapter 5: Apache Sqoop 261

■ Chapter 6: Apache Flume 287

■ Part IV: Serializing..... 301

■ Chapter 7: Apache Avro 303

■ Chapter 8: Apache Parquet..... 325

■ **Part V: Messaging & Indexing 337**

■ **Chapter 9: Apache Kafka 339**

■ **Chapter 10: Apache Solr 349**

■ **Chapter 11: Apache Mahout 377**

Index..... 415

Contents

- About the Authorxv
- About the Technical Reviewerxvii
- Forewordxix
- Part I: Fundamentals 1
- Chapter 1: Introduction 3
 - Core Components of Apache Hadoop 5
 - Why Apache Hadoop? 5
 - Shortcomings in Current Computing Systems..... 5
 - How Is Hadoop Better than Other Distributed Computing Systems? 6
 - What Kind of Computations Is Hadoop Suitable For? 9
 - What Kind of Computations Is Hadoop Not Suitable For? 9
 - HDFS Daemons..... 9
 - NameNode 9
 - Secondary NameNode 10
 - DataNodes 12
 - MapReduce Daemons 14
 - JobTracker 14
 - TaskTracker 16
 - ResourceManager 16
 - NodeManager 16
 - ApplicationMaster..... 17
 - JobHistory Server 17

- Resource Allocations with YARN..... 17
- The Workflow of the MapReduce Job..... 25
- HDFS High-Availability Daemons..... 28
 - Active NameNode 28
 - Standby NameNode 29
 - JournalNodes..... 29
 - ZooKeeper 30
 - DataNodes 30
- Benefits and Challenges of HDFS..... 30
 - HDFS Is Distributed..... 31
 - Block Abstraction..... 31
 - Scalable..... 31
 - Fault-Tolerant 31
 - Data Locality 31
 - Data Coherency 31
 - Parallel Streaming Access to Data..... 32
 - Large Quantities of Data..... 32
 - Unstructured Data 32
 - Commodity Hardware 32
 - Portability 32
 - TCP/IP Protocol 32
 - Accessible 32
 - HDFS High Availability 32
 - Single Point of Failure 33
 - Write-Once Model..... 33
 - Not Optimized for Random Seek 33
 - Local Caching Not Supported 33
 - Commodity Hardware Failure 33
 - Detection of Hardware Failure and Recovery 33

File Sizes, Block Sizes, and Block Abstraction in HDFS.....	33
Block Sizes	33
Block Abstraction.....	37
File Sizes	37
HDFS Quotas.....	38
Hadoop Archive.....	38
Data Replication	38
Configuring Replication	39
Data Flow in Replication.....	41
Under Replication	43
Over Replication	44
Misplaced Replicas.....	44
Storage Requirements in Replication	44
NameNode Federation.....	45
Use Cases for HDFS Federation	48
Data Locality.....	49
Rack-Aware Placement Policy.....	51
Network Topology, Network Bandwidth, and Rack Placement Policy.....	52
How Does HDFS Store, Read, and Write Files?.....	55
Reading a File	55
Writing a File	58
Storing a File	62
Checksum Verification	63
Hadoop Cluster in Data Storage and in Data Processing	64
JVM Reusage.....	66
The Old and New APIs	70
Data Serialization Options	72
Writables	72
Avro	73
Comparing Sequence Files and Avro	74
Thrift.....	75

Protocol Buffers	75
Comparing Thrift and Protocol Buffers	76
Choosing a Serialization Mechanism	76
Filesystem Shell Commands for HDFS	77
Making a Directory	77
Listing Files and Directories	77
Putting Files in the HDFS	78
Creating a File	78
Changing Group Associations for Files and Directories	78
Changing Permissions of Files	79
Changing Owner of Files and Directories	79
Copying Files to the Local Filesystem	79
Appending to a File	80
Copying HDFS Files to STDOUT	80
Testing a File	80
Counting Files and Directories	81
Copying Files in HDFS	81
Moving Files Within the HDFS	82
Displaying File and Directory Sizes	82
Outputting a File in Text Format	82
Removing Files and Directories	82
Choosing Key and Value Types for MapReduce Jobs	83
The Lifecycle of a Mapper and a Reducer in a MapReduce Job	84
The Relationship of Input Keys to Output Keys	92
Mapper Input/Output Types	93
Mapper Input/Output Number of Keys/Values	95
Reducer Input/Output Types	96
Reducer Input/Output Number of Keys/Values	98
Sorting Keys and Values	99
Partitioners	101
Combiners	106

Sort and Shuffle Process.....	112
Shuffle	112
Sort.....	114
MapReduce Job Configuration and Submission.....	125
Performance-Optimized Configuration Parameters.....	134
Input and Output Data Paths in a MapReduce Job.....	139
Selecting InputFormat and OutputFormat in a MapReduce Job	141
Input Formats	142
Output Formats.....	145
The Order of Operations in a MapReduce Job.....	148
RecordReader, Sequence Files, and Compression	150
RecordReader	150
Sequence Files	154
Compression	159
Summary.....	162
■ Chapter 2: HDFS and MapReduce	163
Hadoop Distributed Filesystem	164
NameNode	164
DataNodes	164
MapReduce Framework	164
JobTracker	165
TaskTracker	165
ResourceManager	167
NodeManager	167
Job HistoryServer	167
Setting the Environment.....	167
Hadoop Cluster Modes	170
Standalone Mode.....	170
Pseudo-Distributed Mode.....	170
Fully-Distributed Mode	170

Running a MapReduce Job with the MR1 Framework	171
Running MR1 in Standalone Mode	171
Running MR1 in Pseudo-Distributed Mode.....	178
Running MapReduce with the YARN Framework.....	190
Running YARN in Pseudo-Distributed Mode	191
Running Hadoop Streaming	202
Summary	205
 ■ Part II: Storing & Querying	207
 ■ Chapter 3: Apache Hive	209
Setting the Environment.....	210
Configuring Hadoop.....	212
Configuring Hive.....	213
Starting HDFS.....	214
Starting the Hive Server	214
Starting the Hive CLI.....	214
Creating a Database	216
Using a Database	216
Creating a Managed Table.....	217
Loading Data Into a Table	218
Creating a Table Using LIKE.....	220
Adding Data Into a Table from Queries.....	221
Adding Data Using INSERT INTO TABLE	221
Adding Data Using INSERT OVERWRITE.....	223
Creating a Table Using CREATE TABLE AS SELECT	225
Altering a Table.....	227
Truncating a Table	228
Dropping a Table.....	229
Creating an External Table.....	229
Summary	231

■ Chapter 4: Apache HBase	233
Setting the Environment.....	234
Configuring Hadoop.....	236
Configuring HBase.....	237
Configuring Hive.....	239
Starting HBase	240
Starting the HBase Shell	240
Creating an HBase Table	241
Adding Data to a Table.....	242
Listing All Tables.....	244
Getting a Row of Data	244
Scanning a Table	245
Counting the Number of Rows in a Table	250
Altering a Table.....	250
Deleting a Table Row.....	251
Deleting a Table Column.....	252
Disabling and Enabling a Table.....	253
Truncating a Table	254
Dropping a Table.....	254
Determining if a Table Exists.....	254
Creating a Hive External Table Stored By HBase	255
Summary.....	257
■ Part III: Bulk Transferring & Streaming	259
■ Chapter 5: Apache Sqoop	261
Installing MySQL Database.....	263
Creating MySQL Database Tables.....	264
Setting the Environment.....	268
Configuring Hadoop.....	269

Starting HDFS.....	271
Configuring Hive.....	271
Configuring HBase.....	272
Importing Into HDFS	273
Exporting from HDFS.....	277
Importing Into Hive	279
Importing Into HBase.....	282
Summary.....	286
■ Chapter 6: Apache Flume	287
Setting the Environment.....	288
Configuring Hadoop.....	289
Starting HDFS.....	290
Creating an HBase Table	291
Configuring Flume	292
Running a Flume Agent	294
Configuring Flume for HBase Sink	297
Streaming MySQL Logs to an HBase Sink.....	298
Summary.....	300
■ Part IV: Serializing.....	301
■ Chapter 7: Apache Avro	303
Setting the Environment.....	303
Creating an Avro Schema.....	305
Creating a Hive-Managed Table	306
Creating a Hive (Prior to Version 0.14.0) External Table Stored as Avro	307
Creating a Hive (Version 0.14.0 and Later) External Table Stored as Avro	311
Transferring the MySQL Table Data as an Avro Data File with Sqoop.....	315
Summary.....	323

■ Chapter 8: Apache Parquet	325
Setting the Environment.....	325
Configuring Hadoop.....	327
Configuring Hive.....	328
Starting HDFS.....	329
Creating a Hive-Managed Table	329
Creating a Hive External Table.....	330
Hive Version Prior to 0.13.0	331
Hive Version 0.13.0 and Later.....	331
Loading Data Into a Hive External Table	332
Listing the Parquet Data File	334
Summary.....	335
■ Part V: Messaging & Indexing	337
■ Chapter 9: Apache Kafka	339
Setting the Environment.....	340
Starting the Kafka Server.....	341
Creating a Kafka Topic.....	344
Starting a Kafka Producer	345
Starting a Kafka Consumer	345
Producing and Consuming Messages	346
Summary.....	347
■ Chapter 10: Apache Solr	349
Setting the Environment.....	349
Configuring the Solr Schema	351
Starting the Solr Server.....	354
Indexing a Document in Solr	356
Searching a Document in Solr.....	360
Deleting a Document from Solr	365

Indexing a Document in Solr with a Java Client.....	368
Creating a Hive-Managed Table	369
Creating a Hive External Table.....	371
Loading Hive External Table Data	373
Searching a Hive Table Data Indexed in Solr	374
Summary	376
■ Chapter 11: Apache Mahout	377
Setting the Environment.....	377
Configuring and Starting HDFS	379
Setting the Mahout Environment.....	380
Running a Mahout Classification Sample.....	382
Running a Mahout Clustering Sample	384
Developing a User-Based Recommender System with the Mahout Java API.....	385
The Sample Data	386
Creating a Maven Project in Eclipse.....	387
Creating a User-Based Recommender System.....	394
Creating a Recommender Evaluator	396
Running the Recommender	399
Choosing a Recommender Type	401
Choosing a User Similarity Measure.....	403
Choosing a Neighborhood Type	405
Choosing a Neighborhood Size for NearestUserNeighborhood	406
Choosing a Threshold for ThresholdUserNeighborhood.....	407
Running the Evaluator	409
Choosing the Split Between Training Percentage and Test Percentage	412
Summary	414
Index.....	415

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 Yearly 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. Yearly 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. Yearly 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.

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.

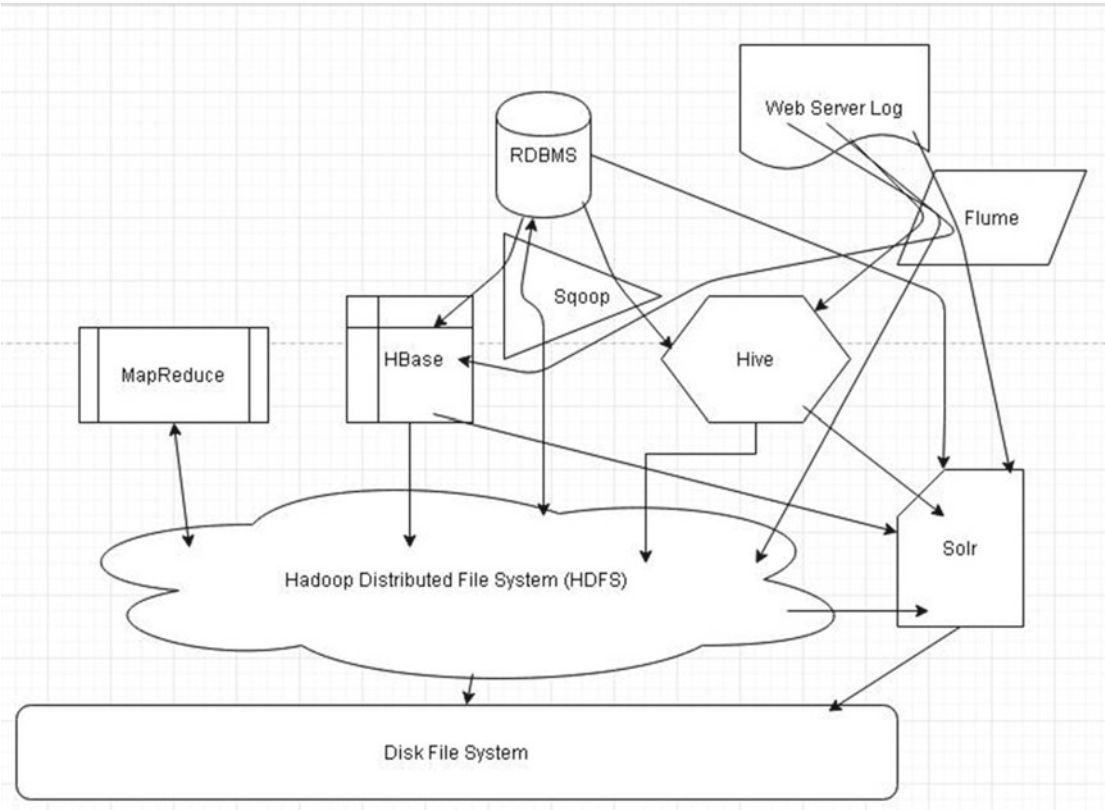


Figure 1-1. The Apache Hadoop ecosystem

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 are 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 compute 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 uncheckpointed 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/namespace`. 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.

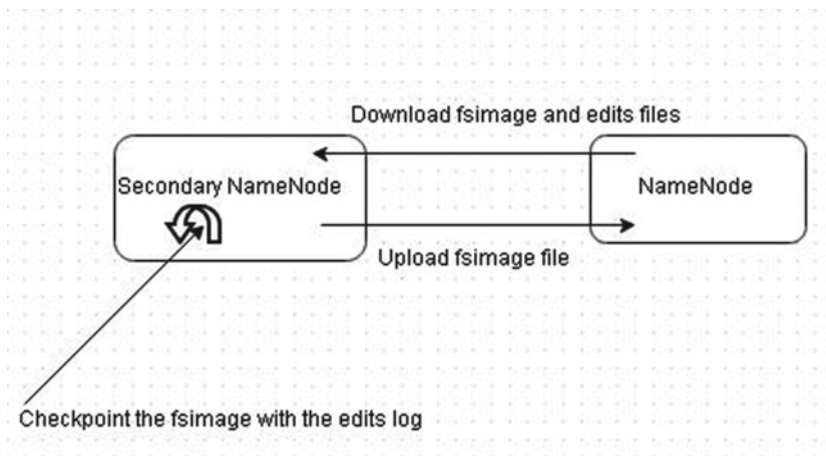


Figure 1-2. Check-pointing by secondary NameNode

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 uncheckpointed 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>
```