O'REILLY®

Compliments of pentaho®

FIELD GUIDE TO Hadoop

An Introduction to Hadoop, Its Ecosystem, and Aligned Technologies

PREVIEW EDITION

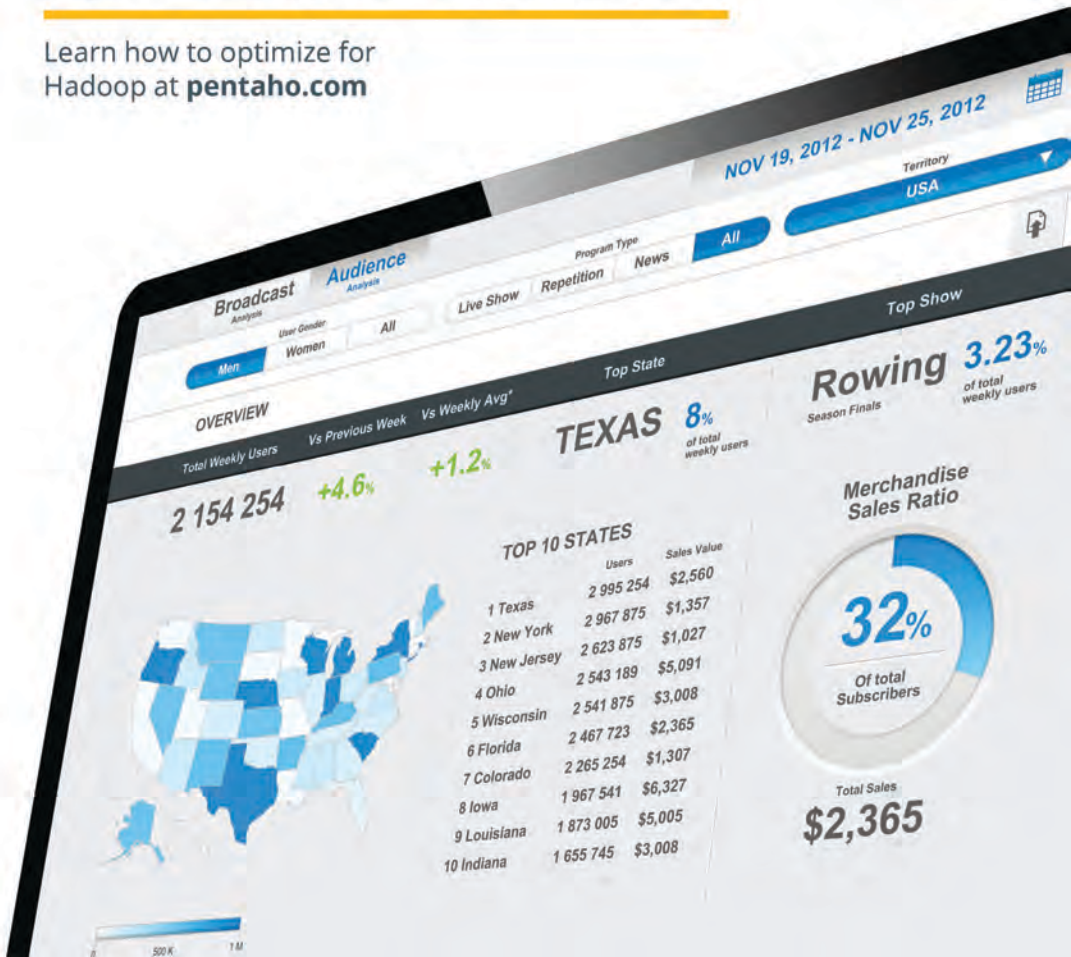
KEVIN SITTO & MARSHALL PRESSER

pentaho

Bring Your **Big Data to Life**

Big Data Integration and Analytics

Learn how to optimize for
Hadoop at **pentaho.com**



This Preview Edition of *Field Guide to Hadoop*, *Chapters 1 and 2*, is a work in progress. The final book is currently scheduled for release in March 2015 and will be available at *oreilly.com* and other retailers once it is published.

A Field Guide to Hadoop

Kevin Sitto and Marshall Presser

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

A Field Guide to Hadoop

by Kevin Sitto and Marshall Presser

Copyright © 2010 Kevin Sitto and Marshall Presser. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Ann Spencer
Production Editor: FIX ME
Copyeditor: FIX ME
Proofreader: FIX ME!
Indexer: FIX ME
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2014-12-02: First release

See *http://oreilly.com/catalog/errata.csp?isbn=0636920032830* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 063-6-920-03283-0

[?]

Table of Contents

1. Core Technologies. 1
Hadoop Distributed File System (HDFS) 3
Tutorial Links 4
Example Code 4
MapReduce 5
Tutorial Links 6
Example or Example Code 6
YARN 7
Tutorial Links 8
Example 8
Spark 9
Tutorial Links 10
Example 10

2. Database and Data Managment. 11
Cassandra 14
Tutorial Links 14
Example 15
HBase 16
Tutorial Links 17
Example code 17
Accumulo 19
Tutorial Links 20
Example 20
Memcached 21
Tutorial Links 22
Example 22
Blur 23

Tutorial Links 23
Example or Example Code 23
Solr 25
Tutorial Links 25
Example or Example Code 26
MongoDB 27
Tutorial Links 28
Example or Example Code 28
Hive 29
Tutorial Links 30
Example 30
Spark SQL Shark 31
Tutorial Links 32
Example 32
Giraph 33
Tutorial Links 35

CHAPTER 1
Core Technologies

In 2002, when the World Wide Web was relatively new and before you "googled" things, Doug Cutting and Mike Cafarella wanted to crawl the web and index the content so that they could produce an internet search enginge. They began a project called Nutch to do this but needed a scalable method to store the content of their indexing. The standard method to organize and store data in 2002 was by means of relational database management systems (RDBMS) which were accessed in a language called SQL. But almost all SQL and relational stores were not appropriate for internet search engine storage and retrieval because they were costly, not terribly scalable, not as tolerant to failure as required and possibly not as performant as desired.

In 2003 and 2004 Google released two important papers, one on the Google File System and the other on a programming model on clustered servers called MapReduce. Cutting and Cafarella incorporated these technologies into their project and eventually Hadoop was born. Hadoop is not an acronym. Cutting had a child that had a yellow stuffed elephant he named Hadoop and somehow that name stuck to the project and the icon is a cute little elephant. Yahoo! began using Hadoop as the basis of its search engine and soon its use spread to many other organizations. Now Hadoop is the predominant Big Data platform. There is a wealth of information that describes Hadoop in great detaill; here you will find a brief synopsis of many components and pointers on where to learn more.

Hadoop consists of three primary resources:

1. the Hadoop Distributed File system (HDFS),

2. the MapReduce programing platform
3. the Hadoop ecosystem, a collection of tools that use or sit beside MapReduce and HDFS to store and organize data, and manage the machines that run Hadoop.

These machines are called a *cluster*, a group of servers, almost always running some variant of the Linux operating system, that work to gether to perform a task.

The Hadoop ecosystem consists of modules that help program the system, manage and configure the cluster, manage data in the cluster, manage storage in the cluster, perform analytic tasks, and the like. The majority of the modules in this book will describe the components of the ecosystem and related technologies.

Hadoop Distributed File System (HDFS)

License	Apache License 2.0
Activity	High
Purpose	High capacity, fault tolerant, inexpensive storage of very large data sets
Official Page	*http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html*
Hadoop Integration	Fully Integrated

The Hadoop Distributed File System (HDFS) is the place in a Hadoop cluster that you store data. Built for data intensive applications, the HDFS is designed to run on clusters of inexpensive commodity servers. HDFS is optimized for high performance, read intensive operations and resiliant to failures in the cluster. It does not prevent failures, but is unlikely to lose data, since by default HDFS makes multiple copies of each of its data blocks. Moreover, HDFS is a WORM-ish (Write Once Read Many) file system. Once a file is created, the file system API only allows you to append to the file, not to overwrite it. As a result, HDFS is usually inappropriate for normal OLTP (On Line Transaction Processing) applications. Most uses of HDFS are for sequential reads of large files. These files are broken into large blocks, usually 64MB or larger in size, and these blocks are distributed amongst the nodes in the server.

HDFS is not a POSIX compliant file system as you would see on Linux, MacOS and on some Windows platforms (see *http://en.wikipedia.org/wiki/POSIX* for a brief explanation). It is not managed by the OS kernels on the nodes in the server. Blocks in HDFS are mapped to files in the host's underlying file system, often ext3 in Linux systems. HDFS does not assume that the underlying disks in the host are RAID protected, so by default, three copies of each block are made and are placed on different nodes in the cluster. This provides protection against lost data when nodes or disks fail, but also assists in Hadoop's notion of accessing data where it resides, rather than moving it through a network to access it.

Although an explanation is beyond the scope of this book, metadata about the files in the HDFS is managed through a Namenode, the Hadoop equivalent of the Unix/Linux superblock.

Tutorial Links

Often times you'll be interacting with HDFS through other tools like "Hive" on page 29 or (to come). That said there will be times when you want to work directly with HDFS, Yahoo has published an excellent guide for configuring and exploring a basic system.

Example Code

When you use the Command Line Interface (CLI) from a Hadoop client, you can copy a file from your local file system to the HDFS and then look at the first ten lines with the following code snippet:

```
[hadoop@client-host ~]$ hadoop fs -ls /data
Found 4 items
drwxr-xr-x - hadoop supergroup 0 2012-07-12 08:55 /data/faa
-rw-r--r-- 1 hadoop supergroup 100 2012-08-02 13:29 /data/sample.txt
drwxr-xr-x - hadoop supergroup 0 2012-08-09 19:19 /data/wc
drwxr-xr-x - hadoop supergroup 0 2012-09-11 11:14 /data/weblogs

[hadoop@client-host ~]$ hadoop fs -ls /data/weblogs/

[hadoop@client-host ~]$ hadoop fs -mkdir /data/weblogs/in

[hadoop@client-host ~]$ hadoop fs -copyFromLocal weblogs_Aug_2008.ORIG /data/webl

[hadoop@client-host ~]$ hadoop fs -ls /data/weblogs/in
Found 1 items
-rw-r--r-- 1 hadoop supergroup 9000 2012-09-11 11:15 /data/weblogs/in/weblogs_Aug

[hadoop@client-host ~]$ hadoop fs -cat /data/weblogs/in/weblogs_Aug_2008.ORIG \
| head
10.254.0.51 - - [29/Aug/2008:12:29:13 -0700] "GGGG / HTTP/1.1" 200 1456
10.254.0.52 - - [29/Aug/2008:12:29:13 -0700] "GET / HTTP/1.1" 200 1456
10.254.0.53 - - [29/Aug/2008:12:29:13 -0700] "GET /apache_pb.gif HTTP/1.1" 200 23
10.254.0.54 - - [29/Aug/2008:12:29:13 -0700] "GET /favicon.ico HTTP/1.1" 404 209
10.254.0.55 - - [29/Aug/2008:12:29:16 -0700] "GET /favicon.ico HTTP/1.1"
404 209
10.254.0.56 - - [29/Aug/2008:12:29:21 -0700] "GET /mapreduce HTTP/1.1" 301 236
10.254.0.57 - - [29/Aug/2008:12:29:21 -0700] "GET /develop/ HTTP/1.1" 200 2657
10.254.0.58 - - [29/Aug/2008:12:29:21 -0700] "GET /develop/images/gradient.jpg
HTTP/1.1" 200 16624
10.254.0.59 - - [29/Aug/2008:12:29:27 -0700] "GET /manual/ HTTP/1.1" 200 7559
10.254.0.62 - - [29/Aug/2008:12:29:27 -0700] "GET /manual/style/css/manual.css
HTTP/1.1" 200 18674
```

MapReduce

License	Apache License 2.0
Activity	High
Purpose	A programming paradigm for processing Big Data
Official Page	hadoop.apache.org
Hadoop Integration	Fully Integrated

MapReduce was the first and is the primary programming framework for developing application in Hadoop. You'll need to work in Java to use MapReduce in its orignal and pure form. You should study word-count, the "hello world" program of Hadoop. The code comes with all the standard Hadoop distributions. Here's your problem in word-count: you have a data set that consists of a large set of documents and the goal is to produce a list of all the words and the number of times they appear in the data set.

MapReduce jobs consist of Java programs called *mappers* and *reducers*. Orchestrated by the Hadoop software, each of the mappers is given chunks of data to analyze. Let's assume it gets a sentence: "The dog ate the food." It would emit 5 name/value pairs or maps: "the":1, "dog":1, "ate":1, "the":1, and "food":1. The name in the name/value pair is the word, and the value is a count of how many times it appears. Hadoop takes the result of your map job and sorts it and for each map a hash value is created to assign it to a reducer in a step called the shuffle. The reducer would sum all the maps for each word in its input stream and produce a sorted list of words in the document. You can think of mappers as programs that extract data from HDFS files into maps and reducers as programs that take the output from the mappers and aggregate results. The tutorials below explain this in greater detail.

You'll be pleased to know that much of the hard work of dividing up the input data sets, assigning the mappers and reducers to nodes, shuffling the data from the mappers to the reducers, and writing out the final results to the HDFS is managed by Hadoop itself. Programmers merely have to write the map and reduce functions. Mappers and reducers are usually written in Java as in the example cited below and writing MapReduce code is non-trivial for novices. To that end, higher level constructs have been developed to do this. Pig is one example and will be discussed in another section. Hadoop Streaming is another.

Tutorial Links

There's a number of excellent tutorials for working with MapReduce. A good place to start is the official Apache documentation but Yahoo has also put together a tutorial module. The folks at MapR, a commercial software company that makes a Hadoop distribution, has a great presentation on writing MapReduce.

Example or Example Code

Writing MapReduce can be fairly complicated and is beyond the scope of this book. A typical application folks write to get started is a simple word count. The official tutorial for building that application can be found with the official documentation.

YARN

License	Apache v2
Activity	Medium
Purpose	Processing
Official Page	hadoop.apache.org
Hadoop Integration	Integrated

When many folks think about Hadoop they are really thinking about two related technolgoies. These two technologies are the Hadoop Distributed File System (HDFS) which houses your data and MapReduce which allows you to actually do things with your data. While MapReduce is great for certain categories of tasks it falls short with others. This led to fracturing in the ecosystem and a variety of tools that live outside of your Hadoop cluster but attempt to communicate with HDFS.

In May of 2012 version 2.0 of Hadoop was released and with it came an exciting change to the way you can interact with your data. This change came with the introduction of YARN which stands for Yet Another Resource Negotiator.

YARN exists in the space between your data and where MapReduce now lives and it allows for many other tools which used to live outside your Hadoop system, such as Spark and Giraph, to now exist natively within a Hadoop cluster. It's important to understand that Yarn does not replace MapReduce; in fact Yarn doesn't do anything at all on its own. What Yarn does do is provide a convenient, uniform way for a variety of tools such as MapReduce, HBase, or any custom utilities you might build to run on your Hadoop cluster.

Tutorial Links

YARN is still an evolving technology and the official Apache guide is really the best place to get started.

Example

Often times we'd put an example here but the truth is that writing applications in Yarn is still very involved and too deep for this book. You can find a link to an excellent walkthrough for building your first Yarn application in the tutorial links.

Spark

License	Apache v2
Activity	High
Purpose	Processing/Storage
Official Page	spark.incubator.apache.org
Hadoop Integration	Integrated

MapReduce is the primary workhorse at the core of most Hadoop clusters. While highly effective for very large batch analytic jobs MapReduce has proven to be sub-optimal for applications like graph analysis that require iterative processing and data sharing.

Spark is designed to provide a more flexible model which supports many of the multi-pass application that falter in MapReduce. It accomplishes this goal by taking advantage of memory whenever possible in order to reduce the amount of data that is written to and read from disk. Spark is not a tool for making MapReduce easier to use like Pig or Hive. It is a complete replacement for MapReduce that includes it's own work execution engine.

Spark operates with three core ideas

1. Resilient Distributed Dataset (RDD):: RDDs contain data that you want to transform or analyze. They can either be be read from an external source such as a file or a database or they can be created by a transformation.
2. Transformation:: A transformation modifies an existing RDD to create a new RDD. For example, a filter that pulls ERROR messages out of a log file would be a transformation.
3. Action:: An action analyzes an RDD and returns a single result. For example, an action would count the number of results identified by our ERROR filter.

If you want to do any significant work in Spark, you would be wise to learn about Scala, a functional programming language. Scala combines object orientation with functional programming. Becasue Lisp is an older functional programming language, Scala might be called "Lisp joins the 21st century." You can read more about Scala at *http://www.scala-lang.org/*. This is not to say that Scala is the only way to work with Spark. The project also has strong support for Java and Python, but when new APIs or features are added, they appear first in Scala.

Tutorial Links

A quick start for Spark can be found on the project home page.

Example

We'll start with opening the Spark shell by running ./bin/spark-shell from the directory we installed Spark in.

In this example we're going to count the number of Dune reviews in our review file.

```
// Read the csv file containing our reviews
scala> val reviews = spark.textFile("hdfs://reviews.csv")
testFile: spark.RDD[String] = spark.MappedRDD@3d7e837f

// This is a two-part operation:
// first we'll filter down to the two lines that contain Dune reviews
// then we'll count those lines
scala> val dune_reviews = reviews.filter(line => line.contains("Dune")).count()
res0: Long = 2
```

CHAPTER 2
Database and Data Managment

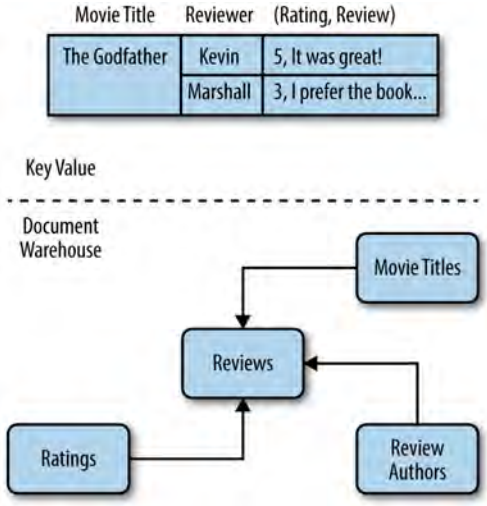
If you're planning to use Hadoop, it's likely that you'll be managing lots of data and in addition to MapReduce jobs, you may need some kind of database. Since the advent of Google's BigTable, Hadoop has an interest in the management of data. While there are some relational SQL databases or SQL interfaces to HDFS data, like Hive, much data management in Hadoop uses non SQL techniques to store and access data. The website *http://nosql-database.org/* lists more than 150 NoSql databases that are then classified as:

- Column Stores,
- Document Stores
- Key Value/Tuple Stores
- Graph Databases
- Multimodel Databases
- Object Databases
- Grid and Cloud Databaese
- Multivalue Databases
- Tabular Stores,
- Others.

NoSQL databases generally do not support relational join operations, complex transactions, or foreign key constraints common in relational systems, but generally scale better to large amounts of data. You'll have to decide what works best for your data sets and the information you

wish to extract from them. It's quite possible that you'll be using more than one.

This book will look at many of the leading examples in each section, but the focus will be on the two major categories: Key/Value Stores and Document Stores.



A Key/Value store can be thought of like a catalog. All the items in a catalog (the values) are organized around some sort of index (the keys). Just like a catalog, a key/value store is very quick and effective if you know the key you're looking for but isn't a whole lot of help if you don't.

For example, let's say I'm looking for Marshall's review of "The Godfather". I can quickly refer to my index, find all the reviews for that film and scroll down to Marshall's review:

I prefer the book…

A document warehouse, on the other hand, is a much more flexible type of database. Rather than forcing you to organize your data around a specific key it allows you to index and search for your data based on any number of parameters. Let's expand on the last example and say I'm in the mood to watch a movie based on a book. One naive way to find such a movie would be to search for reviews that contain the word "book".

In this case a key/value store wouldn't be a whole lot of help as my key is not very clearly defined. What I need is a document warehouse that

will let me quickly search all the text of all the reviews and find those that contain the word “book”

Cassandra

License	GPL v2
Activity	High
Purpose	Key/Value Store
Official Page	cassandra.apache.org
Hadoop Integration	Compatible

Often times you may need to simply organize some of your big data for easy retrieval. One common way to do this is to use a key/value datastore. This type of database looks like the white pages in a phone book. Your data is organized by a unique "key" and values are associated with that key. For example, if you want to store information about your customers you may use that customer's user name is their key and associate information such as their transaction history and address as values associated with that key.

Key/Value datastores are a common fixture in any big data system because they are easy to scale, quick and straightforward to work with. Cassandra is a distributed key/value database designed with simplicity and scalability in mind. While often compared to "HBase" on page 16, Cassandra differs in a few key ways:

- Cassandra is an all-inclusive system, this means you do not need a Hadoop environment or any other big data tools behind Cassandra.
- Cassandra is completely masterless, it operates as a peer-to-peer system. This makes it easier to configure and highly resilient.

Tutorial Links

DataStax, a company that provides commercial support for Cassandra, offers a set of freely available videos.

Example

The easiest way to interact with Cassandra is through its shell interface. You start the shell by running "bin/cqlsh" from your install directory.

Then you need to create a keyspace. Keyspaces are similar to schemas in traditional relational databases, they are a convenient way to organize your tables. A typical pattern is to use a single different keyspace for each application.

```
CREATE KEYSPACE field_guide
WITH REPLICATION = { 'class': 'SimpleStrategy', 'replication factor' : 3 };

USE field_guide;
```

Now that you have a keyspace you'll create a table within that keyspace to hold your reviews. This table will have three columns and a primary key that consists of both the reviewer and the title as that pair should be unique within the database.

```
CREATE TABLE reviews (
    reviewer varchar,
    title varchar,
    rating int,
    PRIMARY KEY (reviewer, title));
```

Once your table is created you can insert a few reviews.

```
INSERT INTO reviews (reviewer,title,rating)
    VALUES ('Kevin','Dune',10);
INSERT INTO reviews (reviewer,title,rating)
    VALUES ('Marshall','Dune',1);
INSERT INTO reviews (reviewer,title,rating)
    VALUES ('Kevin','Casablanca',5);
```

And now that you have some data you will create an index that will allow you to execute a simple sql query to retrieve Dune reviews.

```
CREATE INDEX ON reviews (title);

SELECT * FROM reviews WHERE title = 'Dune';

 reviewer |      title | rating
----------+------------+-------
    Kevin |       Dune |     10
 Marshall |       Dune |      1
    Kevin | Casablanca |      5
```

HBase

License	Apache, Version 2.0
Activity	High
Purpose	No-SQL database with random access
Official Page	hbase.apache.org
Hadoop Integration	Fully Integrated

There are many situations in which you might have sparse data. That is, there are many attributes of the data, but each observation only has a few of them. For example, you might want a table of various tickets in a help desk application. Tickets for email might have different information (and attributes or columns) than tickets for network problems or lost passwords, or issues with backup system. There are other situations in which you have data that has a large number of common values in an column or attribute, say Country or State. Each of these example might lead you to consider HBase

HBase is a No-SQL database system included in the standard Hadoop distributions. It is a key-value store logically. This means that rows are defined by a key, and have associated with them a number of bins (or columns) where the associated values are stored. The only data type is the byte string. Physically, groups of similar columns are stored together in column families. Most often, HBase is accessed via Java code, but APIs exist for using HBase with Pig, Thrift, jython (Python based), and others. HBase is not normally accessed in a MapReduce fashion. It does have a shell interface for interactive use.

HBase is often used for application which may require sparse rows. That is, each row may use only a few of the defined columns. It is fast (as Hadoop goes) when access to elements is done through the primary key, or defining key value. It's highly scalable and reasonable fast. Unlike traditional HDFS applications, it permits random access to rows, rather than sequential searches.

Though faster than MapReduce, you should not use HBase for any kind of transactional needs, nor any kind of relational analytics. It does not support any secondary indexes, so finding all rows where a given column has a specific value is tedious and must be done at the application level. HBase does not have a JOIN operation, this must be done by the individual application. You must provide security at the application level, other tools like "Accumulo" on page 19 are built with security in mind.

While Cassandra "Cassandra" on page 14 and MongoDB "MongoDB" on page 27 might still be the predominant No-SQL databases today, HBase is gaining in popularity and may well be the leader in the near future.

Tutorial Links

The folks at coreservlets.com have put together a handful of Hadoop tutorials including an excellent series on HBase. There's also a handful of videop recorded tutorials available on the internet, including this one the authors found particularly helpful.

Example code

In this example, your goal is to find the average review for the movie Dune. Each movie review has 3 elements, a reviewer name, a film title, and a rating (an iteger from zero to ten). The example is done in the hbase shell.

```
hbase(main):008:0> create 'reviews', 'cf1'
0 row(s) in 1.0710 seconds

hbase(main):013:0> put 'reviews', 'dune-marshall', 'cf1:score', 1
0 row(s) in 0.0370 seconds

hbase(main):015:0> put 'reviews', 'dune-kevin', 'cf1:score', 10
0 row(s) in 0.0090 seconds

hbase(main):016:0> put 'reviews', 'casablanca-kevin', 'cf1:score', 5
0 row(s) in 0.0130 seconds

hbase(main):017:0> put 'reviews', 'blazingsaddles-b0b', 'cf1:score', 9
0 row(s) in 0.0090 seconds

hbase(main):018:0> scan 'reviews'
ROW                          COLUMN+CELL
 blazingsaddles-b0b          column=cf1:score, timestamp=1390598651108, value=9
 casablanca-kevin            column=cf1:score, timestamp=1390598627889, value=5
```

```
 dune-kevin                  column=cf1:score, timestamp=1390598600034, value=10
 dune-marshall               column=cf1:score, timestamp=1390598579439, value=1
3 row(s) in 0.0290 seconds

hbase(main):023:0> scan 'reviews', {STARTROW => 'dune', ENDROW => 'dunf'}
ROW                          COLUMN+CELL
 dune-kevin                  column=cf1:score, timestamp=1390598791384, value=10
 dune-marshall               column=cf1:score, timestamp=1390598579439, value=1
2 row(s) in 0.0090 seconds
```

Now you've retrieved the two rows using an efficient range scan, but how do you compute the average? In the hbase shell, it's not possible, but using the hbase Java APIs, you can extract the values, but there is no builtin row aggregation function for average or sum, so you would need to do this in your Java code.

The choice of the row key is critical in hbase. If you want to find the average of all movie Kevin reviewed, you would need to do a full table scan, potentially a very tedious with a very large data set. You might want to have two versions of the table, one with the row key given by reviewer-film and another with film-reviewer. Then you would have the problem of ensuring they're in sync.

Accumulo

License	Apache, Version 2.0
Activity	High
Purpose	Name-value database with cell level security
Official Page	accumulo.apache.org
Hadoop Integration	Fully Integrated

You have an application that could use a good column/name-value store, like HBase "HBase" on page 16 but you have an additional security issue; you must carefully control which users can see which cells in your data. For example, you could have a multi-tenancy data store in which you are storing data from different organizations in your enterprise in a single table and want to ensure that users from one organization cannot see the data from another organization, but that senior management can see across the whole enterprise. For its internal security reasons, the U.S. National Security Agency (NSA) developed Accumulo and then donated the code to the Apache foundation.

You'll find that Accumulo is similar to, but not exactly the same as HBase with security. A major difference is the security mechanism. Each user has a set of security labels, simple text strings. Suppose yours were "admin", "audit", and "GroupW". When you want to define the access to a particular cell, you set the column visibility for that column in a given row to a boolean expression of the various labels. In this syntax, the & is logical AND and | is logical OR. If the cell's visibility rule were admin|audit, then any user with either admin or audit label could see that cell. If the column visibillity rule were admin&Group7, you would not be able to see it, as you lack the Group7 label and both are required.

But Accumulo is more than just security. It also can run at massive scale, with many petabytes of data with hundreds of thousands of ingest and retrieval operations per second.

Tutorial Links

An introduction from Aaron Cordova, one of the orgininators of Accumulo.

A video tutorial that focuses on performance and the Accumulo architecture.

This tutorial is more focused on security and encryption.

The 2014 Accumulo Summit has a wealth of information.

Example

Good example code is a bit long and complex to include here, but can be found on the "examples" section of the project's home page.

Memcached

License	Revised BSD License
Activity	Medium
Purpose	In-Memory Cache
Official Page	memcached.org
Hadoop Integration	None

It's entirely likely you will eventually encounter a situation where you need very fast access to a large amount of data for a short period of time. For example, let's say you want to send an email to your customers and prospects letting them know about new features you've added to your product but you also need to make certain you exclude folks you've already contacted this month.

The way you'd typically address this query in a big data system is by distributing your large contact list across many machines then loading the entirety of your list of folks contacted this month into memory on each machine and quickly checking each contact against your list of folks you've already emailed. In MapReduce this is often referred to as a "replicated join". However, let's assume you've got a large network of contacts consisting of many millions of email addresses you've collcted from trade shows, product demos and social media and you like to contact these folks fairly often. This means your list of folks you've already contacted this month could be fairly large and the entire list might not fit into the amount of memory you've got available on each machine.

What you really need is some way to pool memory across all your machines and let everyone refer back to that large pool. Memcached is a tool that lets you build such a distributed memory pool. To follow up on our previous example, you would store the entire list of folks who've already been emailed into your distributed memory pool and

instruct all the different machines processing your full contact list to refer back to that memory pool instead of local memory.

Tutorial Links

The spymemcached project has a handful of examples using their api available on their wiki.

Example

Let's say we need to keep track of which reviewers have already reviewed which movies so we don't ask a reviewer to review the same movie twice. Because there is no single, oficially supported Java client for memcached we'll use the popular spymemcached client, available at code.google.com/p/spymemcached.

We'll start by defining a client and pointing it at our memcached servers.

```
MemcachedClient client = new MemcachedClient(
    AddrUtil.getAddresses("server1:11211 server2:11211"));
```

Now we'll start loading data into our cache. We'll use the popular OpenCSV library (opencsv.sourceforge.net) to read our reviews file and write an entry to our cache for every reviewer and title pair we find.

```
CSVReader reader = new CSVReader(new FileReader("reviews.csv"));
String [] line;
while ((line = reader.readNext()) != null) {
    //Merge the reviewer name and the movie title
    // into a single value (ie: KevinDune) that we'll use as a key
    String reviewerAndTitle = line[0] + line[1];
    //Write the key to our cache and store it for 30 minutes (188 seconds)
    client.set(reviewerAndTitle, 1800, true);
}
```

Once we have our values loaded into the cache we can quickly check the cache from a MapReduce job or any other java code.

```
Object myObject=client.get(aKey);
```

Blur

License	Apache v2
Activity	Medium
Purpose	Document Warehouse
Official Page	incubator.apache.org/blur
Hadoop Integration	Integrated

Let's say you've bought in to the entire big data story using Hadoop. You've got Flume gathering data and pushing it into HDFS, your Map-Reduce jobs are transforming that data and building key/value pairs that are pushed into HBase and you even have a couple enterprising data scientists using Mahout to analyze your data. At this point your CTO walks up to you and asks how often one of your specific products is mentioned in a feedback form your are collecting from your users. Your heart drops as you realize the feedback is free-form text and you've got no way to search any of that data.

Blur is tool for indexing and searching text with Hadoop. Because it has Lucene, a very popular text indexing framework, at it's core it has many useful features such as fuzzy matching, wildcard searches and paged results. It allows you to search through unstructured data in a way that would be otherwise very dificult.

Tutorial Links

You can't go wrong with the official "getting started" guide on the project homepage. There is also an excellent, if not slightly out of date, presentation from a Hadoop User Group meeting in 2011.

Example or Example Code

There are a couple different ways to load data into Blur. When you have large amounts of data you want to index in bulk you will likely

use MapReduce whereas if you want to stream data in you are likely better off with the mutation interface. In this case we're going to use the mutation interface because we're just going to index a couple records.

```
import static org.apache.blur.thrift.util.BlurThriftHelper.*;

Iface aClient = BlurClient.getClient("controller1:40010,controller2:40010");

//Create a new Row in table 1
RowMutation mutation1 = newRowMutation("reviews", "Dune",
    newRecordMutation("review", "review_1.json",
        newColumn("Reviewer", "Kevin"),
        newColumn("Rating", "10")
        newColumn("Text", "I was taken away with the movie's greatness!")
    ),
    newRecordMutation("review", "review_2.json",
        newColumn("Reviewer", "Marshall"),
        newColumn("Rating", "1")
        newColumn("Text", "I thought the movie was pretty terrible :(")
    )
);

client.mutate(mutation);
```

Now let's say we want to search for all reviews where the review text mentions something being great. We're going to pull up the Blur shell by running "/bin/blur shell" from our installation directory and run a simple query. This query tells Blur to look in the Text column of the review column family in the reviews table for anything that looks like the word "great".

```
blur> query reviews review.Text:great
 - Results Summary -
    total : 1
    time : 41.372 ms

     hit : 0
   score : 0.9548232184568715
      id : Dune
recordId : review_1.json
  family : review
    Text : I was taken away with the movie's greatness!

 - Results Summary -
    total : 1
    time  : 41.372 ms
```

Solr

License	Apache v2
Activity	High
Purpose	Document Warehouse
Official Page	lucene.apache.org/solr
Hadoop Integration	Compatible

Sometimes you just want to search through a big stack of documents. Not all tasks require big, complex analysis jobs spanning petabytes of data. For many common use cases you may find that you have too much data for a simple unix grep or windows search but not quite enough to warrant a team of data scientists. Solr fits comfortably in that middle ground, providing an easy to use means to quickly index and search the contents of many documents.

Solr supports a distributed architecture that provides many of the benefits you expect from a big data systems such as linear scalability, data replication and failover. It is based on Lucene, a popular framework for indexing and searching documents and implements that framework by providing a set of tools for building indexes and querying data.

While Solr is able to use "Hadoop Distributed File System (HDFS)" on page 3 to store data it is not truly compatible with Hadoop and does not use "MapReduce" on page 5 or "YARN" on page 7 to build indexes or respond to queries. There is a similar effort named "Blur" on page 23 to build a tool on top of the Lucene framework that leverages the entire Hadoop stack.

Tutorial Links

Apart from the tutorial on the official Solr homepage there is a solr wiki with great information.

Example or Example Code

In this example we're going to assume we have a set of semi-structured data consisting of movie reviews with labels that clearly mark the title and the text of the review. These reviews will be stored in individual json files in the "reviews" directory.

We'll start by telling Solr to index our data, there's a handful of different ways to do this, all with unique tradeoffs. In this case we're going to use the simplest mechanism which is the post.sh script located in the exampledocs subdirectory of our Solr install.

```
./example/exampledocs/post.sh /reviews/*.json
```

Once our reviews have been indexed they are ready to search. Solr has its own web gui that can be used for simple searches, we'll pull up that gui and search of the titles of all movies with the word "great" in the review.

```
review_text:great&fl=title
```

This search tells Solr that we want to retrieve the title field (fl=title) for any review where the word "great" appears in the review_text field.

MongoDB

License	Free Software Foundation's GNU AGPL v3.0., commercial licenses available from MongoDB, Inc.,
Activity	High
Purpose	JSON document oriented database
Official Page	mongodb.org
Hadoop Integration	Compatible

If you have a large number of (to come) documents in your Hadoop cluster and need some data management tool to effectively use them, consider MongoDB, an open-source, big data, document oriented database whose documents are JSON objects. At the start of 2014 it is one the most popular NO-SQL database. Unlike some other database systems, MongoDB supports secondary indexes — meaning it is possible to quickly search on other than the primary key that uniquely identifies each document in the mongo database. The name derives from the slang word humongous, meaning very, very large. While MongoDB did not originally run on Hadoop and the HDFS, it can be used in conjunction with Hadoop.

Mongodb is a documented oriented database, the document being a JSON object. In relational databases, you have tables and rows. In Mongodb, the equivalent of a row is a JSON document and the analog to a table is a collection, a set of JSON documents. To understand Mongodb, you should read the JSON section of this book first.

Perhaps the best way to understand its use is by way of a code example. This time you'll want to compute the average ranking of the movie "Dune" in the standard data set. If you know Python, this will be clear. If you don't, the code is still pretty straight forward.

Tutorial Links

The tutorials section on the official project page is a great place to get started. There are also plenty of videos of talks available on the internet, including this informative series.

Example or Example Code

```
#!/usr/bin/python
# import required packages
import sys
import pymongo
# json movie reviews
movieReviews = [
    { "reviewer":"Kevin" , "movie":"Dune", "rating","10" },
    { "reviewer":"Marshall" , "movie":"Dune", "rating","1" },
    { "reviewer":"Kevin" , "movie":"Casablanca", "rating","5" },
    { "reviewer":"Bob" , "movie":"Blazing Saddles", "rating","9" }
]
# mongodb connection info
MONGODB_INFO = 'mongodb://juser:secretpassword@localhost:27018/db'
# connect to mongodb
client=pymongo.MongoClient(MONGODB_INFO)
db=client.get_defalut_database()
# create the movies collection
movies=db['movies']
#insert the movie reviews
movies.insert(movieReviews)
# find all the movies with title Dune, iterature through them finding all scores
# by using standard db cursor technology
mcur=movies.find({'movie': {'movie': 'Dune'})
count=0
sum=0
# for all reviews of dune, count them up and sum the rankings
for m in mcur:
    count += 1
    sum += m['rating']
client.close()
rank=float(sum)/float(count)
print ('Dune %s\n' % rank)
```

Hive

License	Apache
Cost	Free
Activity	High
Purpose	Data Interaction

At first, all access to data in your Hadoop cluster came through MapReduce jobs written in Java. This worked fine during Hadoop's infancy when all Hadoop users had a stable of Java savvy coders. But as Hadoop emerged into the broader world, many wanted to adopt Hadoop but had stables of SQL coders for whom writing MapReduce would be a steep learning curve. Enter Hive. The goal of Hive is to allow SQL access to data in the HDFS. The Apache Hive data warehouse software facilitates querying and managing large datasets residing in HDFS. Hive defines a simple SQL-like query language, called QL, that enables users familiar with SQL to query the data. Queries written in QL are converted into MapReduce code by Hive and executed by Hadoop. But beware! QL is not full ANSI standard SQL. While the basics are covered, some features are missing. Here's a partial list as of early 2014.

- there are no correlated sub-queries
- update and delete statements are not supported
- transactions are not supported
- outer joins are not possible

You may not need these, but if you run code generated by 3rd party solutions, they may generate non Hive compliant code.

Hive does not mandate read or written data be in the "Hive format"---there is no such thing. This means your data can be accessed directly by Hive without any of the extral, transform & load (ETL) preprocessing typically required by traditional relational databases.

Tutorial Links

A couple of great resources would be the official Hive tutorial and this video published by the folks at HortonWorks.

Example

Say we have a comma separated file containing movie reviews with information about the reviewer, the movie and the rating:

```
Kevin,Dune,10
Marshall,Dune,1
Kevin,Casablanca,5
Bob,Blazing Saddles,9
```

First we need to define the schema for our data:

```
CREATE TABLE movie_reviews ( reviewer STRING, title STRING, rating INT)
ROW FORMAT DELIMITED
FILEDS TERMINATED BY '\,'
STORED AS TEXTFILE
```

Next we need to load the data by pointing the table at our movie reviews file. Since Hive doesn't require that data be stored in any specific format loading a table consists simply of pointing Hive at a file in hdfs.

```
LOAD DATA LOCAL INPATH 'reviews.csv'
OVERWRITE INTO TABLE movie_reviews
```

Now we are ready to perform some sort of analysis. Let's say, in this case, we want to find the average rating for the movie Dune:

```
Select AVG(rating) FROM movie_reviews WHERE title = 'Dune';
```

Spark SQL Shark

License	Apache
Activity	High
Purpose	SQL access to Hadoop Data
Official Page	shark.cs.berkeley.edu
Hadoop Integration	Unknown

If you need SQL access to your data, and Hive "Hive" on page 29 is a bit underperforming, and you're willing to commit to a Spark "Spark" on page 9 environment, then you need to consider Spark SQL or Shark as your SQL engine. As of July 2014, the Shark project ceased development and its successor, Spark SQL is now the main line SQL project on Spark. You can find more information about the change at *http://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html* . Spark SQL, like Spark, has an in-memory computing model which helps to account for its speed. It's only in recent years that decreasing memory costs have made large memory Linux servers ubiquitious, thus leading to recent advances in in-memory computing for large data sets. Since memory access times are usually 100 times as fast as disk access times, it's quite appealing to keep as much in memory as possible, using the disks as infrequently as possible. But abandoning MapReduce has made Spark SQK much faster even if it requires disk access.

While Spark SQL is very similar to Hive, it has a few extra features that aren't in Hive. One is the ability to encache table data for the duration of a user session. This corresponds to temporary tables in many other databases but unlike other databases, these tables live in memory and are thus accessed much faster.

Spark SQL supports the Hive metastore, most of its query language, and data formats, so existing Hive users should have an easier time converting to Shark than many others. However, while the Spark SQL

documentation is currently not absolutely clear on this, not all the Hive features have yet been implemented in Spark SQL. APIs currently exist for Python, Java, and Scala. See the section on Hive for more details. Spark SQL also can run Spark's MLlib machine learning algorithms as SQL statements.

Spark SQL can use JSON (to come) and Parquet (to come) as data sources, so it's pretty useful in an HDFS environment.

Tutorial Links

There are a wealth of tutorials at on the project homepage.

Example

At the user level, Shark looks like Hive, so if you can code in Hive, you can almost code in Spark SQK. But you need to set up your Spark SQL environment. Here's how you would do it in Python using the movie review data we use in other examples. To understand the setup you'll need to read the section on Spark "Spark" on page 9 as well as have some knowledge of Python.

```
# Spark requires a Context object.  Let's assume it exists already.
# You need a SQL Context object as well
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# Load a the CSV text file and convert each line to a Python dictionary
# using lambda notation for anonymous functions.
lines = sc.textFile("reviews.csv")
movies = lines.map(lambda l: l.split(","))
reviews = movies.map(lambda p: {"name": p[0], "title": p[1], "rating": int(p[2])}

# Spark SQL needs to think of the RDD (Resilient Distributed Dataset)
# as a data schema and register the table name
schemaReviews = sqlContext.inferSchema(reviews)
schemaReviews.registerAsTable("reviews")

# once you've registered the RDD as a schema,
# you can run SQL statements over it.
dune_reviews = sqlContext.sql("SELECT * FROM reviews WHERE title = 'Dune'")
```

Giraph

License	Apache
Activity	High
Purpose	graph database
Official Page	giraph.apache.org
Hadoop Integration	Fully Integrated

You may know a parlor game called 6 Degrees of Separation from Kevin Bacon in which movie trivia experts try to find the closest relationship between a movie actor and Kevin Bacon. If an actor is in the same movie, that's a "path" of length 1. If an actor has never been in a movie with Kevin Bacon, but has been in a movie with an actor who has been, that's a path of length 2. It rests on the assumption that any individual involved in the Hollywood, California, film industry can be linked through his or her film roles to Kevin Bacon within six steps, 6 degrees of separation. For example, there is an arc between Kevin Bacon and Sean Penn because they were both in "Mystic River", so they have one degree of separation or a path of length 1. But Benicio Del Toro has a path of length 2 because he has never been in a movie with Kevin Bacon, but has been in one with Sean Penn.

You can show these relationship by means of a graph, a set of ordered pairs (N,M) which describe a connection from N to M.

You can think of a tree (such as a hierarchical file system) as a graph with a single source node or origin, and arcs leading down the tree branches. The set {(top, b1), (top, b2), (b1,c1), (b1,c2), (b2,c3)} is a tree rooted at top, with branches from top to b1 and b2, b1 to c1 and c2 and b2 to c3. The elements of the set {top, b1, b2, c1,c2,c3} are called the nodes.

You will find graphs useful in describing relationships between entities. For example, if you had a collection of emails sent between people in your organization, you could build a graph where each node represents a person in your organization and an arc would exist node a and node b if a sent an email to b. It could like like this:

Giraph is an Appache project to build and extract information from graphs. For example, you could use Giraph to calculate the shortest distance (number of arc hops) from one node in the graph to another or to calculate if there was a path between two nodes.

Apache Giraph is derived from a Google project called Pregel and has been used by Facebook to build and analyze a graph with a trillion nodes, admittedly on a very large Hadoop cluster. It is built using a technology call Bulk Synchronous Parallel (BSP).

The general notion is that there are a set of "supersteps" in the BSP model. In step zero, the vertices or nodes are distributed to worker processes. In each following superstep, the nodes each of the vertices iterates through a set of messages it received from the previous superset and sends messages to other nodes to which it is connected.

In the Kevin Bacon example, each node represents an actor, director, producer, screenwriter, etc. Each arc connects two people who are part of the same movie. And we want to test the hypothesis that everyone in the industry is connected to Kevin within 6 hops in the graph. Each node is given an initial value to the number of hops; for Kevin Bacon,

it is zero. For everyone else, the initial value is a very large integer. At the first superstep, each node sends its value to all those nodes connected to it. Then at each of the other supersteps, each node first reads all its messages and takes the minimum value. If it is less than its current value, the node adds one and then sends this to all its connected node at the end of the superstep. Why? Because if a connected node is N steps from Kevin, then this node is at mosts N+1 steps away. Once a node has established a new value, it opts out of sending more messages.

At the end of 6 supersteps, you'll have all the persons connected to Kevin Bacon by 6 or fewer hops.

Tutorial Links

The official product page has a quick start guide. In addition there's a handful of video taped talks, including one by PayPal and another by Facebook. Finally, there's this particularly informative blog post.