Community Experience Distilled

Scaling Big Data with Hadoop and Solr

Learn exciting new ways to build efficient, high performance enterprise search repositories for Big Data using Hadoop and Solr

Hrishikesh Karambelkar

[PACKT] open source*
PUBLISHING
community experience distilled

Scaling Big Data with Hadoop and Solr

Learn exciting new ways to build efficient, high performance enterprise search repositories for Big Data using Hadoop and Solr

Hrishikesh Karambelkar

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Scaling Big Data with Hadoop and Solr

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1190813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-137-4

www.packtpub.com

Cover Image by Prashant Timappa Shetty (sparkling.spectrum.123@gmail.com)

Credits

Author
Hrishikesh Karambelkar

Reviewer
Parvin Gasimzade

Acquisition Editor
Kartikey Pandey

Commisioning Editor
Shaon Basu

Technical Editors
Pratik More
Amit Ramadas
Shali Sasidharan

Project Coordinator
Akash Poojary

Proofreader
Lauren Harkins

Indexer
Tejal Soni

Graphics
Ronak Dhruv

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Author

Hrishikesh Karambelkar is a software architect with a blend of entrepreneurial and professional experience. His core expertise involves working with multiple technologies such as Apache Hadoop and Solr, and architecting new solutions for the next generation of a product line for his organization. He has published various research papers in the domains of graph searches in databases at various international conferences in the past. On a technical note, Hrishikesh has worked on many challenging problems in the industry involving Apache Hadoop and Solr.

> While writing the book, I spend my late nights and weekends bringing in the value for the readers. There were few who stood by me during good and bad times, my lovely wife Dhanashree, my younger brother Rupesh, and my parents. I dedicate this book to them. I would like to thank the Apache community users who added a lot of interesting content for this topic, without them, I would not have got an opportunity to add new interesting information to this book.

About the Reviewer

Parvin Gasimzade is a MSc student in the department of Computer Engineering at Ozyegin University. He is also a Research Assistant and a member of the Cloud Computing Research Group (CCRG) at Ozyegin University. He is currently working on the Social Media Analysis as a Service concept. His research interests include Cloud Computing, Big Data, Social and Data Mining, information retrieval, and NoSQL databases. He received his BSc degree in Computer Engineering from Bogazici University in 2009, where he mainly worked on web technologies and distributed systems. He is also a professional Software Engineer with more than five years of working experience. Currently, he works at the Inomera Research Company as a Software Engineer. He can be contacted at `parvin.gasimzade@gmail.com`.

Table of Contents

Preface

This book will provide users with a step-by-step guide to work with Big Data using Hadoop and Solr. It starts with a basic understanding of Hadoop and Solr, and gradually gets into building efficient, high performance enterprise search repository for Big Data.

You will learn various architectures and data workflows for distributed search system. In the later chapters, this book provides information about optimizing the Big Data search instance ensuring high availability and reliability.

This book later demonstrates two real world use cases about how Hadoop and Solr can be used together for distributer enterprise search.

What this book covers

Chapter 1, *Processing Big Data Using Hadoop and MapReduce*, introduces you with Apache Hadoop and its ecosystem, HDFS, and MapReduce. You will also learn how to write MapReduce programs, configure Hadoop cluster, the configuration files, and the administration of your cluster.

Chapter 2, *Understanding Solr*, introduces you to Apache Solr. It explains how you can configure the Solr instance, how to create indexes and load your data in the Solr repository, and how you can use Solr effectively for searching. It also discusses interesting features of Apache Solr.

Chapter 3, *Making Big Data Work for Hadoop and Solr*, brings the two worlds together; it drives you through different approaches for achieving Big Data work with architectures and their benefits and applicability.

Chapter 4, Using Big Data to Build Your Large Indexing, explains the NoSQL and concepts of distributed search. It then gets you into using different algorithms for Big Data search covering shards and indexing. It also talks about SolrCloud configuration and Lily.

Chapter 5, Improving Performance of Search while Scaling with Big Data, covers different levels of optimizations that you can perform on your Big Data search instance as the data keeps growing. It discusses different performance improvement techniques which can be implemented by the users for their deployment.

Appendix A, Use Cases for Big Data Search, describes some industry use cases and case studies for Big Data using Solr and Hadoop.

Appendix B, Creating Enterprise Search Using Apache Solr, shares a sample Solr schema which can be used by the users for experimenting with Apache Solr.

Appendix C, Sample MapReduce Programs to Build the Solr Indexes, provides a sample MapReduce program to build distributed Solr indexes for different approaches.

What you need for this book

This book discusses different approaches, each approach needs a different set of software. To run Apache Hadoop/Solr instance, you need:

- JDK 6
- Apache Hadoop
- Apache Solr 4.0 or above
- Patch sets, depending upon which setup you intend to run
- Katta (only if you are setting Katta)
- Lily (only if you are setting Lily)

Who this book is for

This book provides guidance for developers who wish to build high speed enterprise search platform using Hadoop and Solr. This book is primarily aimed at Java programmers, who wish to extend Hadoop platform to make it run as an enterprise search without prior knowledge of Apache Hadoop and Solr.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "You will typically find the `hadoop-example jar` in `/usr/share/hadoop`, or in `$HADOOP_HOME`."

A block of code is set as follows:

```
public static class IndexReducer {
  protected void setup(Context context) throws IOException,
    InterruptedException {
      super.setup(context);
      SolrRecordWriter.addReducerContext(context);
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

A programming task is divided into multiple identical subtasks, and when it is distributed among multiple machines for processing, it is called a **map task**. The results of these map tasks are combined together into one or many **reduce tasks**. Overall, this approach of computing tasks is called the **MapReduce approach**.

Any command-line input or output is written as follows:

```
java -Durl=http://node1:8983/solr/clusterCollection/update -jar
  post.jar ipod_video.xml
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The admin UI will start showing the **Cloud** tab."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Processing Big Data Using Hadoop and MapReduce

Traditionally computation has been processor driven. As the data grew, the industry was focused towards increasing processor speed and memory for getting better performances for computation. This gave birth to the distributed systems. In today's real world, different applications create hundreds and thousands of gigabytes of data every day. This data comes from disparate sources such as application software, sensors, social media, mobile devices, logs, and so on. Such huge data is difficult to operate upon using standard available software for data processing. This is mainly because the data size grows exponentially with time. Traditional distributed systems were not sufficient to manage the big data, and there was a need for modern systems that could handle heavy data load, with scalability and high availability. This is called **Big Data**.

Big data is usually associated with high volume and heavily growing data with unpredictable content. A video gaming industry needs to predict the performance of over 500 GB of data structure, and analyze over 4 TB of operational logs every day; many gaming companies use Big Data based technologies to do so. An IT advisory firm **Gartner** defines big data using 3Vs (high volume of data, high velocity of processing speed, and high variety of information). IBM added fourth V (high veracity) to its definition to make sure the data is accurate, and helps you make your business decisions.

While the potential benefits of big data are real and significant, there remain many challenges. So, organizations which deal with such high volumes of data face the following problems:

- **Data acquisition**: There is lot of raw data that gets generated out of various data sources. The challenge is to filter and compress the data, and extract the information out of it once it is cleaned.
- **Information storage and organization**: Once the information is captured out of raw data, the data model will be created and stored in a storage device. To store a huge dataset effectively, traditional relational system stops being effective at such a high scale. There has been a new breed of databases called **NOSQL** databases, which are mainly used to work with big data. NOSQL databases are non-relational databases.
- **Information search and analytics**: Storing data is only a part of building a warehouse. Data is useful only when it is computed. Big data is often noisy, dynamic, and heterogeneous. This information is searched, mined, and analyzed for behavioral modeling.
- **Data security and privacy**: While bringing in linked data from multiple sources, organizations need to worry about data security and privacy at the most.

Big data offers lot of technology challenges to the current technologies in use today. It requires large quantities of data processing within the finite timeframe, which brings in technologies such as massively parallel processing (**MPP**) technologies and distributed file systems.

Big data is catching more and more attention from various organizations. Many of them have already started exploring it. Recently Gartner (`http://www.gartner.com/newsroom/id/2304615`) published an executive program survey report, which reveals that Big Data and analytics are among the top 10 business priorities for CIOs. Similarly, analytics and BI stand as the top priority for CIO's technical priorities. We will try to understand Apache Hadoop in this chapter. We will cover the following:

- Understanding Apache Hadoop and its ecosystem
- Storing large data in HDFS
- Creating MapReduce to analyze the Hadoop data
- Installing and running Hadoop
- Managing and viewing a Hadoop cluster
- Administration tools

Understanding Apache Hadoop and its ecosystem

Google faced the problem of storing and processing big data, and they came up with the MapReduce approach, which is basically a divide-and-conquer strategy for distributed data processing.

> A programming task which is divided into multiple identical subtasks, and which is distributed among multiple machines for processing, is called a **map task.**. The results out of these map tasks are combined together into one or many **reduce tasks**. Overall this approach of computing tasks is called a **MapReduce** approach.

MapReduce is widely accepted by many organizations to run their Big Data computations. Apache Hadoop is the most popular open source Apache licensed implementation of MapReduce. Apache Hadoop is based on the work done by Google in the early 2000s, more specifically on papers describing the Google file system published in 2003, and MapReduce published in 2004. Apache Hadoop enables distributed processing of large datasets across a commodity of clustered servers. It is designed to scale up from single server to thousands of commodity hardware machines, each offering partial computational units and data storage.

Apache Hadoop mainly consists of two major components:

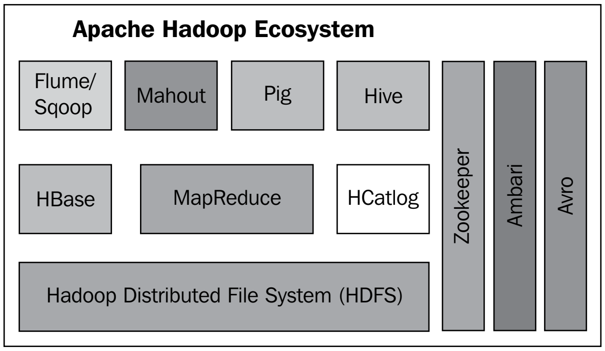
- The Hadoop Distributed File System (HDFS)
- The MapReduce software framework

HDFS is responsible for storing the data in a distributed manner across multiple Hadoop cluster nodes. The MapReduce framework provides rich computational APIs for developers to code, which eventually run as map and reduce tasks on the Hadoop cluster.

The ecosystem of Apache Hadoop

Understanding Apache Hadoop ecosystem enables us to effectively apply the concepts of the MapReduce paradigm at different requirements. It also provides end-to-end solutions to various problems that are faced by us every day.

Apache Hadoop ecosystem is vast in nature. It has grown drastically over the time due to different organizations contributing to this open source initiative. Due to the huge ecosystem, it meets the needs of different organizations for high performance analytics. To understand the ecosystem, let's look at the following diagram:



Apache Hadoop ecosystem consists of the following major components:

- Core Hadoop framework: HDFS and MapReduce
- Metadata management: HCatalog
- Data storage and querying: HBase, Hive, and Pig
- Data import/export: Flume, Sqoop
- Analytics and machine learning: Mahout
- Distributed coordination: Zookeeper
- Cluster management: Ambari
- Data storage and serialization: Avro

Apache HBase

HDFS is append-only file system; it does not allow data modification. Apache HBase is a distributed, random-access, and column-oriented database. HBase directly runs on top of HDFS, and it allows application developers to read/write the HDFS data directly. HBase does not support SQL; hence, it is also called as NOSQL database. However, it provides command-line-based interface, as well as a rich set of APIs to update the data. The data in HBase gets stored as key-value pairs in HDFS.

Apache Pig

Apache Pig provides another abstraction layer on top of MapReduce. It provides something called Pig Latin, which is a programming language that creates MapReduce programs using Pig. Pig Latin is a high-level language for developers to write high-level software for analyzing data. Pig code generates parallel execution tasks, therefore effectively uses the distributed Hadoop cluster. Pig was initially developed at Yahoo! Research to enable developers create ad-hoc MapReduce jobs for Hadoop. Since then, many big organizations such as eBay, LinkedIn, and Twitter have started using Apache Pig.

Apache Hive

Apache Hive provides data warehouse capabilities using Big Data. Hive runs on top of Apache Hadoop, and uses HDFS for storing its data. The Apache Hadoop framework is difficult to understand, and it requires a different approach from traditional programming to write MapReduce-based programs. With Hive, developers do not write MapReduce at all. Hive provides a SQL like query language called HiveQL to application developers, enabling them to quickly write ad-hoc queries similar to RDBMS SQL queries.

Apache ZooKeeper

Apache Hadoop nodes communicate with each other through Apache Zookeeper. It forms the mandatory part of Apache Hadoop ecosystem. Apache Zookeeper is responsible for maintaining coordination among various nodes. Besides coordinating among nodes, it also maintains configuration information, and group services to the distributed system. Apache ZooKeeper can be used independent of Hadoop, unlike other components of the ecosystem. Due to its in-memory management of information, it offers the distributed coordination at a high speed.

Apache Mahout

Apache Mahout is an open source machine learning software library that can effectively empower Hadoop users with analytical capabilities such as clustering, data mining, and so on, over distributed Hadoop cluster. Mahout is highly effective over large datasets, the algorithms provided by Mahout are highly optimized to run the MapReduce framework over HDFS.

Apache HCatalog

Apache HCatalog provides metadata management services on top of Apache Hadoop. It means all the software that runs on Hadoop can effectively use HCatalog to store their schemas in HDFS. HCatalog helps any third party software to create, edit, and expose (using rest APIs) the generated metadata or table definitions. So, any user or script can run Hadoop effectively without actually knowing where the data is physically stored on HDFS. HCatalog provides **DDL** (**Data Definition Language**) commands with which the requested MapReduce, Pig, and Hive jobs can be queued for execution, and later monitored for progress as and when required.

Apache Ambari

Apache Ambari provides a set of tools to monitor Apache Hadoop cluster hiding the complexities of the Hadoop framework. It offers features such as installation wizard, system alerts and metrics, provisioning and management of Hadoop cluster, job performances, and so on. Ambari exposes RESTful APIs for administrators to allow integration with any other software.

Apache Avro

Since Hadoop deals with large datasets, it becomes very important to optimally process and store the data effectively on the disks. This large data should be efficiently organized to enable different programming languages to read large datasets Apache Avro helps you to do that. Avro effectively provides data compression and storages at various nodes of Apache Hadoop. Avro-based stores can easily be read using scripting languages as well as Java. Avro provides dynamic access to data, which in turn allows software to access any arbitrary data dynamically. Avro can be effectively used in the Apache Hadoop MapReduce framework for data serialization.

Apache Sqoop

Apache Sqoop is a tool designed to do load large datasets in Hadoop efficiently. Apache Sqoop allows application developers to import/export easily from specific data sources such as relational databases, enterprise data warehouses, and custom applications. Apache Sqoop internally uses a map task to perform data import/export effectively on Hadoop cluster. Each mapper loads/unloads slice of data across HDFS and data source. Apache Sqoop establishes connectivity between non-Hadoop data sources and HDFS.

Apache Flume

Apache Flume provides a framework to populate Hadoop with data from nonconventional data sources. Typical use of Apache Flume could be for log aggregation. Apache Flume is a distributed data collection service that gets flow of data from their sources, aggregates them, and puts them in HDFS. Most of the time, Apache Flume is used as an **ETL** (**Extract-Transform-Load**) utility at various implementation of the Hadoop cluster.

We have gone through the complete ecosystem of Apache Hadoop. These components together make Hadoop one of the most powerful distributed computing software available today for use. Many companies offer commercial implementations and support for Hadoop. Among them is the **Cloudera** software, a company that provides Apache Hadoop's open source distribution, also called **CDH** (**Cloudera distribution including Apache Hadoop**), enables organizations to have commercial Hadoop setup with support. Similarly, companies such as IBM, Microsoft, MapR, and Talend provide implementation and support for the Hadoop framework.

Storing large data in HDFS

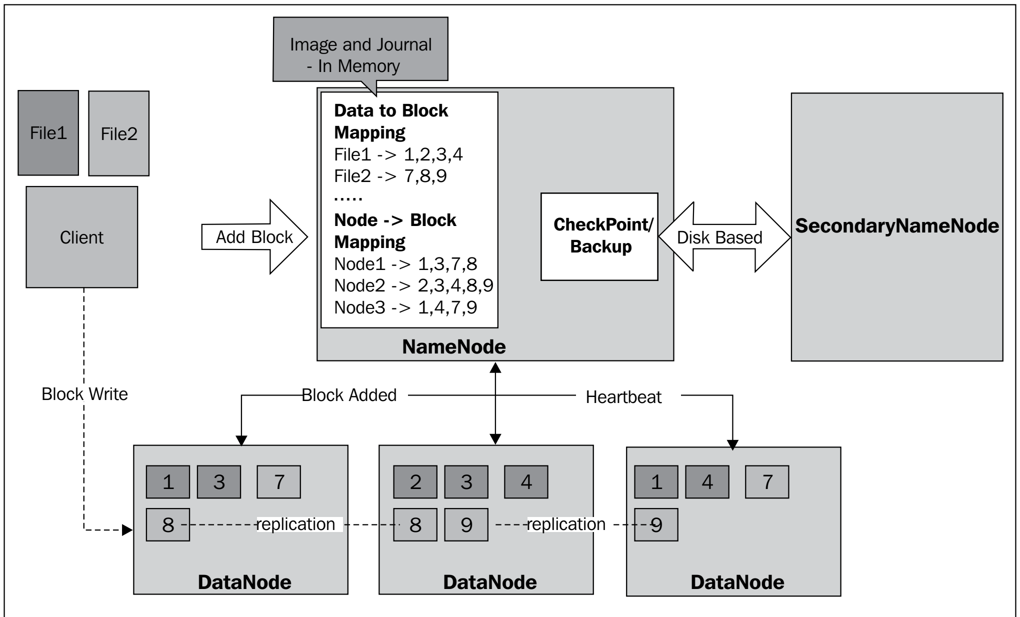
Hadoop distributed file system (HDFS) is a subproject of Apache foundation. It is designed to maintain large data/files in a distributed manner reliably. HDFS uses master-slave based architecture and is designed to run on low- cost hardware. It is a distributed file system which provides high speed data access across distributed network. It also provides APIs to manage its file system. To handle failures of nodes, HDFS effectively uses data replication of file blocks across multiple Hadoop cluster nodes, thereby avoiding any data loss during node failures. HDFS stores its metadata and application data separately. Let's understand its architecture.

HDFS architecture

HDFS, being a distributed file system, has the following major objectives to satisfy to be effective:

- Handling large chunks of data
- High availability, and handling hardware failures seamlessly
- Streaming access to its data
- Scalability to perform better with addition of hardware
- Durability with no loss of data in spite of failures
- Portability across different types of hardware/software
- Data partitioning across multiple nodes in a cluster

HDFS satisfies most of these goals effectively. The following diagram depicts the system architecture of HDFS. Let's understand each of the components in detail.



HDFS Architecture

NameNode

All the metadata related to HDFS is stored on NameNode. Besides storing metadata, NameNode is the master node which performs coordination activities among data nodes such as data replication across data nodes, naming system such as filenames, their disk locations, and so on. NameNode stores the mapping of blocks to the DataNodes. In a Hadoop cluster, there can only be one single active NameNode. NameNode regulates access to its file system with the use of HDFS based APIs to create, open, edit, and delete HDFS files. The data structure for storing file information is inspired from a UNIX-like filesystem. Each block is indexed, and its index node (inode) mapping is available in memory (RAM) for faster access. NameNode is a multithreaded process and can serve multiple clients at a time.

Any transaction first gets recorded in journal, and the journal file, after completion is flushed and response is sent back to the client. If there is any error while flushing journal to disk, NameNode simply excludes that storage, and moves on with another. NameNode shuts itself down in case no storage directory is available.

> **Safe mode**: When a cluster is started, NameNode starts its complete functionality only when configured minimum percentage of block satisfies the minimum replication. Otherwise, it goes into safe mode. When NameNode is in safe mode state, it does not allow any modification to its file systems. This can be turned off manually by running the following command:
>
> ```
> $  hadoop dfsadmin – safemode  leave
> ```

DataNode

DataNodes are nothing but slaves that are deployed on all the nodes in a Hadoop cluster. DataNode is responsible for storing the application's data. Each uploaded data file in HDFS is split into multiple blocks, and these data blocks are stored on different data nodes. Default file block size in HDFS is 64 MB. Each Hadoop file block is mapped to two files in data node, one file is the file block data, while the other one is checksum.

When Hadoop is started, each data node connects to NameNode informing its availability to serve the requests. When system is started, the namespace ID and software versions are verified by NameNode, and DataNode sends block report describing what all data blocks it holds to NameNode on startup. During runtime, each DataNode periodically sends NameNode a heartbeat signal, confirming its availability. The default duration between two heartbeats is 3 seconds. NameNode assumes unavailability of DataNode if it does not receive a heartbeat in 10 minutes by default; in that case NameNode does replication of data blocks of that DataNode to other DataNodes. Heartbeat carries information about disk space available, in-use space, data transfer load, and so on. Heartbeat provides primary handshaking across NameNode and DataNode; based on heartbeat information, NameNode chooses next block storage preference, thus balancing the load in the cluster. NameNode effectively uses heartbeat replies to communicate to DataNode regarding block replication to other DataNodes, removal of any blocks, requests for block reports, and so on.

Secondary NameNode

Hadoop runs with single NameNode, which in turn causes it to be a single point of failure for the cluster. To avoid this issue, and to create backup for primary NameNode, the concept of secondary NameNode was introduced recently in the Hadoop framework. While NameNode is busy serving request to various clients, secondary NameNode looks after maintaining a copy of up-to-date memory snapshot of NameNode. These are also called **checkpoints**.

Secondary NameNode usually runs on a different node other than NameNode, this ensures durability of NameNode. In addition to secondary NameNode, Hadoop also supports CheckpointNode, which creates period checkpoints instead of running a sync of memory with NameNode. In case of failure of NameNode, the recovery is possible up to the last checkpoint snapshot taken by CheckpointNode.

Organizing data

Hadoop distributed file system supports traditional hierarchy based file system (such as UNIX), where user can create their own home directories, subdirectories, and store files in these directories. It allows users to create, rename, move, and delete files as well as directories. There is a root directory denoted with slash (/), and all subdirectories can be created under this root directory, for example `/user/foo`.

> The default data replication factor on HDFS is three; however one can change this by modifying HDFS configuration files.

Data is organized in multiple data blocks, each comprising 64 MB size by default. Any new file created on HDFS first goes through a stage, where this file is cached on local storage until it reaches the size of one block, and then the client sends a request to NameNode. NameNode, looking at its load on DataNodes, sends information about destination block location and node ID to the client, then client flushes the data to the targeted DataNodes the from local file. In case of unflushed data, if the client flushes the file, the same is sent to DataNode for storage. The data is replicated at multiple nodes through a replication pipeline.

Accessing HDFS

HDFS can be accessed in the following different ways:

- Java APIs
- Hadoop command line APIs (FS shell)
- C/C++ language wrapper APIs
- WebDAV (work in progress)
- DFSAdmin (command set for administration)
- RESTful APIs for HDFS

Similarly, to expose HDFS APIs to rest of the language stacks, there is a separate project called **HDFS-APIs** (`http://wiki.apache.org/hadoop/HDFS-APIs`), based on the Thrift framework which allows scalable cross-language service APIs to Perl, Python, Ruby, and PHP. Let's look at the supported operations with HDFS.

Hadoop operations	**Syntax**	**Example**
Creating a directory	`hadoop dfs -mkdir  URI`	`hadoop dfs -mkdir /users/abc`
Importing file from local file store	`hadoop dfs -copyFromLocal <localsrc> URI`	`hadoop dfs -copyFromLocal /home/user1/info.txt /users/abc`
Exporting file to local file store	`hadoop dfs -copyToLocal [-ignorecrc] [-crc] URI <localdst>`	`hadoop dfs -copyToLocal /users/abc/info.txt /home/user1`
Opening and reading a file	`hadoop dfs -cat URI [URI …]`	`hadoop dfs -cat /users/abc/info.txt`
Copy files in Hadoop	`hadoop dfs -cp URI [URI …] <dest>`	`hadoop dfs -cp /users/abc/* /users/bcd/`
Moving or renaming a file or directory	`hadoop dfs -mv URI [URI …] <dest>`	`hadoop dfs -cp /users/abc/output /users/bcd/`
Delete a file or directory, recursive delete	`hadoop dfs -rm [-skipTrash] URI [URI …]`	`hadoop dfs -rm /users/abc/info.txt`
Get status of file or directory, size, other information	`hadoop dfs -du <args>`	`hadoop dfs -du /users/abc/info.txt`
List a file or directory	`hadoop dfs -ls <args>`	`hadoop dfs -ls /users/abc`
Get different attributes of file/directory	`hadoop dfs -stat URI`	`hadoop dfs -stat /users/abc`
Change permissions (single/recursive) of file or directory	`hadoop dfs -chmod [-R] MODE URI [URI …]`	`hadoop dfs -chmod 755 /users/abc`

Hadoop operations	Syntax	Example
Set owner for file/directory	`hadoop dfs -chown [-R] [OWNER][:[GROUP]] URI`	`hadoop dfs -chown -R hrishi /users/hrishi/home`
Setting replication factor	`hadoop dfs -setrep [-R] <path>`	`hadoop dfs -setrep -w 3 -R /user/hadoop/dir1`
Change group permissions with file	`hadoop dfs -chgrp [-R] GROUP URI [URI …]`	`hadoop dfs -chgrp -R abc /users/abc`
Getting the count of files and directories	`hadoop dfs -count [-q] <paths>`	`hadoop dfs -count /users/abc`

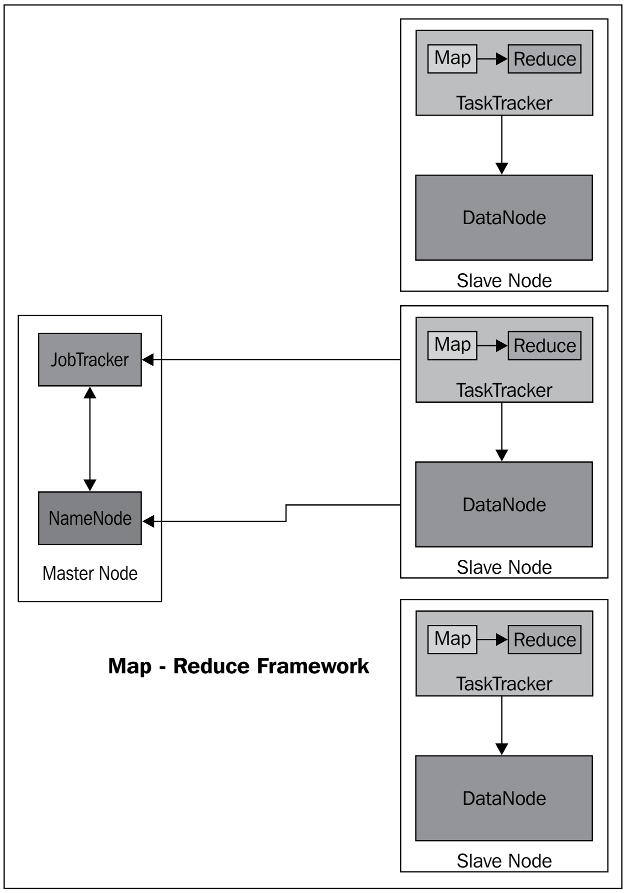
Creating MapReduce to analyze Hadoop data

The MapReduce framework was originally developed at Google, but it is now being adapted as the de facto standard for large scale data analysis.

MapReduce architecture

In the MapReduce programming model, the basic unit of information is a key-value pair. The MapReduce program reads sets of such key-value pairs as input, and outputs new key-value pairs. The overall operation occurs in three different stages, Map-Shuffle-Reduce. All the stages of MapReduce are stateless, enabling them to run independently in a distributed environment. Mapper acts upon one pair at a time, whereas shuffle and reduce can act on multiple pairs. In many cases, shuffle is an optional stage of execution. All of the map tasks should finish before the start of Reduce phase. Overall a program written in MapReduce can undergo many rounds of MapReduce stages one by one. Please take a look at an example of MapReduce in *Appendix C*.

The Hadoop-based MapReduce framework architecture is shown in the following diagram. It is a master-slave architecture consisting of two major components in MapReduce architecture of MapReduce: **JobTracker** and **TaskTracker**.



JobTracker

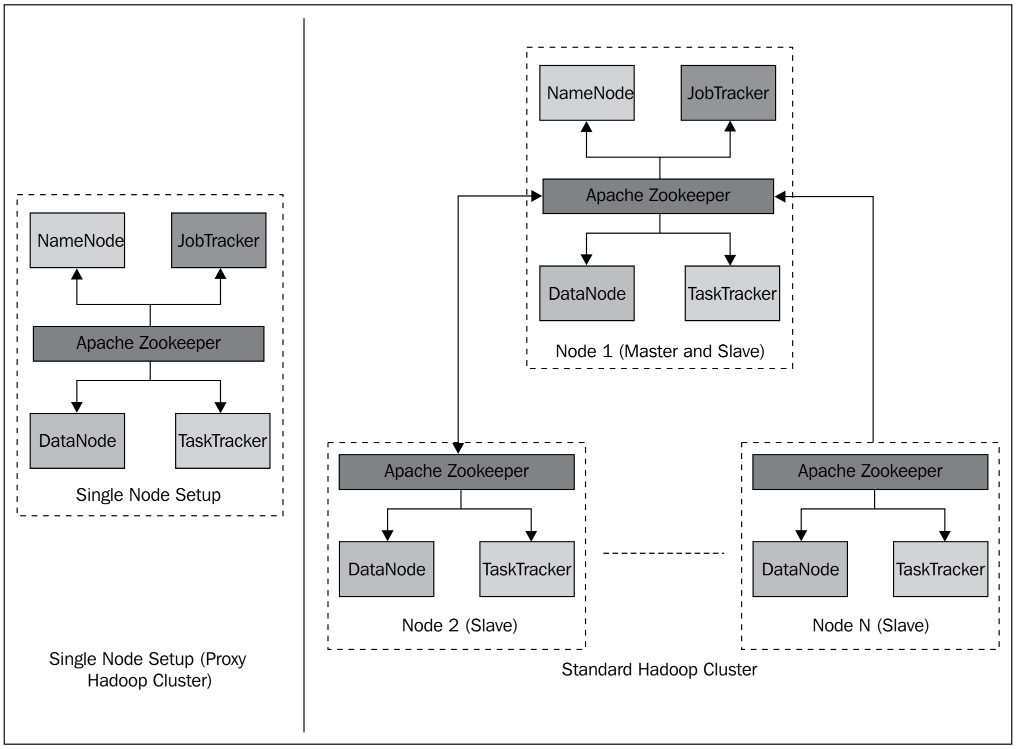
JobTracker is responsible for monitoring and coordinating execution of jobs across different TaskTrackers in Hadoop nodes. Each Hadoop program is submitted to JobTracker, which then requests location of data being referred by the program. Once NameNode returns the location of DataNodes, JobTracker assigns the execution of jobs to respective TaskTrackers on the same machine where data is located. The work is then transferred to TaskTracker for execution. JobTracker keeps track of progress on job execution through heartbeat mechanism. This is similar to the heartbeat mechanism we have seen in HDFS. Based on heartbeat signal, JobTracker keeps the progress status updated. If TaskTracker fails to respond within stipulated time, JobTracker schedules this work to another TaskTracker. In case, if a TaskTracker reports failure of task to JobTracker, JobTracker may assign it to a different TaskTracker, or it may report it back to the client, or it may even end up marking the TaskTracker as unreliable.

TaskTracker

TaskTracker are slaves deployed on Hadoop nodes. They are meant to serve requests from JobTracker. Each TaskTracker has an upper limit on number of tasks that can be executed on node, and they are called slots. Each task runs in its own JVM process, this minimizes impact on the TaskTracker parent process itself due to failure of tasks. The running tasks are then monitored by TaskTracker, and the status is maintained, which is later reported to JobTracker through heartbeat mechanism. To help us understand the concept, we have provided a MapReduce example in *Appendix A*, *Use Cases for Big Data Search*.

Installing and running Hadoop

Installing Hadoop is a straightforward job with a default setup, but as we go on customizing the cluster, it gets difficult. Apache Hadoop can be installed in three different setups: namely standalone mode, single node (pseudo-distributed) setup, and fully distributed setup. Local standalone setup is meant for single machine installation. Standalone mode is very useful for debugging purpose. The other two types of setup are shown in the following diagram:



In pseudo-distributed setup of Hadoop, Hadoop is installed on a single machine; this is mainly for development purpose. In this setup, each Hadoop daemon runs as a separate Java process. A real production based installation would be on multiple nodes or full cluster. Let's look at installing Hadoop and running a simple program on it.

Prerequisites

Hadoop runs on the following operating systems:

- All Linux flavors: It supports development as well as production
- Win32: It has limited support (only for development) through Cygwin

Hadoop requires the following software:

- Java 1.6 onwards
- ssh (Secure shell) to run start/stop/status and other such scripts across cluster
- Cygwin, which is applicable only in case of Windows

This software can be installed directly using apt-get for Ubuntu, dpkg for Debian, and rpm for Red Hat/Oracle Linux from respective sites. In case of cluster setup, this software should be installed on all the machines.

Setting up SSH without passphrases

Since Hadoop uses SSH to run its scripts on different nodes, it is important to make this SSH login happen without any prompt for password. This can simply be tested by running the `ssh` command as shown in the following code snippet:

```
$ssh localhost
  Welcome to Ubuntu (11.0.4)

hduser@node1:~/$
```

If you get a prompt for password, you should perform the following steps on your machine:

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ ssh localhost
```

This step will actually create authorization key with SSH, by passing passphrases check. Once this step is complete, you are good to go.

Installing Hadoop on machines

Hadoop can be first downloaded from the Apache Hadoop website (`http://hadoop.apache.org`). Make sure that you download and choose the correct release from different releases, which is stable release, latest beta/alpha release, and legacy stable version. You can choose to download the package or download the source, compile it on your OS, and then install it. Using operating system package installer, install the Hadoop package.

To setup a single pseudo node cluster, you can simply run the following script provided by Apache:

```
$ hadoop-setup-single-node.sh
```

Say Yes to all the options. This will setup a single node on your machine, you do not need to further change any configuration, and it will run by default. You can test it by running any of the Hadoop command discussed in HDFS section of this chapter.

For a cluster setup, the SSH passphrase should be set on all the nodes, to bypass prompt for password while starting and stopping TaskTracker/DataNodes on all the slaves from masters. You need to install Hadoop on all the machines which are going to participate in the Hadoop cluster. You also need to understand the Hadoop configuration file structure, and make modifications to it.

Hadoop configuration

Major Hadoop configuration is specified in the following configuration files, kept in the `$HADOOP_HOME/conf` folder of the installation:

File name	Description
`core-site.xml`	In this file, you can modify the default properties of Hadoop. This covers setting up different protocols for interaction, working directories, log management, security, buffer and blocks, temporary files, and so on.
`hdfs-site.xml`	This file stores the entire configuration related to HDFS. So properties such as DFS site address, data directory, replication factors, and so on, are covered in these files.
`mapred-site.xml`	This file is responsible for handling the entire configuration related to the MapReduce framework. This covers configuration for JobTracker and TaskTracker, properties for Job.
`common-logging.properties`	This file specifies the default logger used by Hadoop; you can override it to use your logger.

File name	Description
`capacity-scheduler.xml`	This file is mainly used by resource manager in Hadoop for setting up scheduling parameters of job queues.
`fair-scheduler.xml`	This file contains information about user allocations and pooling information for fair scheduler. It is currently under development.
`hadoop-env.sh`	All the environment variables are defined in this file; you can change any of the environments, that is, Java location, Hadoop configuration directory, and so on.
`hadoop-policy.xml`	This file is used to define various access control lists for Hadoop services. This can control who all can use Hadoop cluster for execution.
`Masters/slaves`	In this file, you can define the hostname for master and slaves. Master file lists all the masters, and Slave file lists the slave nodes. To run Hadoop in cluster mode, you need to modify these files to point to the respective master and slaves on all nodes.
`Log4j.properties`	You can define various log levels for your instance, helpful while developing or debugging the Hadoop programs. You can define levels for logging.

The files marked in bold letters are the files that you will definitely modify to set up your basic Hadoop cluster.

Running a program on Hadoop

You can start your cluster with the following command; once started, you will see the output shown as follows:

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


  Starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-
    hduser-namenode-ubuntu.out
  localhost: starting datanode, logging to
    /usr/local/hadoop/bin/../logs/hadoop-hduser-datanode-ubuntu.out
  localhost: starting secondarynamenode, logging to
    /usr/local/hadoop/bin/../logs/hadoop-hduser-secondarynamenode-
      ubuntu.out
starting jobtracker, logging to /usr/local/hadoop/bin/../logs/
    hadoop-hduser-jobtracker-ubuntu.out
  localhost: starting tasktracker, logging to /usr/local/hadoop/bin
    /../logs/hadoop-hduser-tasktracker-ubuntu.out
hduser@ubuntu:/usr/local/hadoop$
```

Now we can test the functioning of this cluster by running sample examples shipped with Hadoop installation. First, copy some files from your local directory on HDFS and you can run following command:

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -copyFromLocal
  /home/myuser/data /user/myuser/data
```

Run `hadoop dfs -ls` on your Hadoop instance to check whether the files are loaded in HDFS. Now, you can run the simple word count program to count the number of words in all these files.

```
bin/hadoop jar hadoop*examples*.jar wordcount /user/myuser/data
  /user/myuser/data-output
```

You will typically find hadoop-example jar in `/usr/share/hadoop`, or in `$HADOOP_HOME`. Once it runs, you can run `hadoop dfs cat` on `data-output` to list the output.

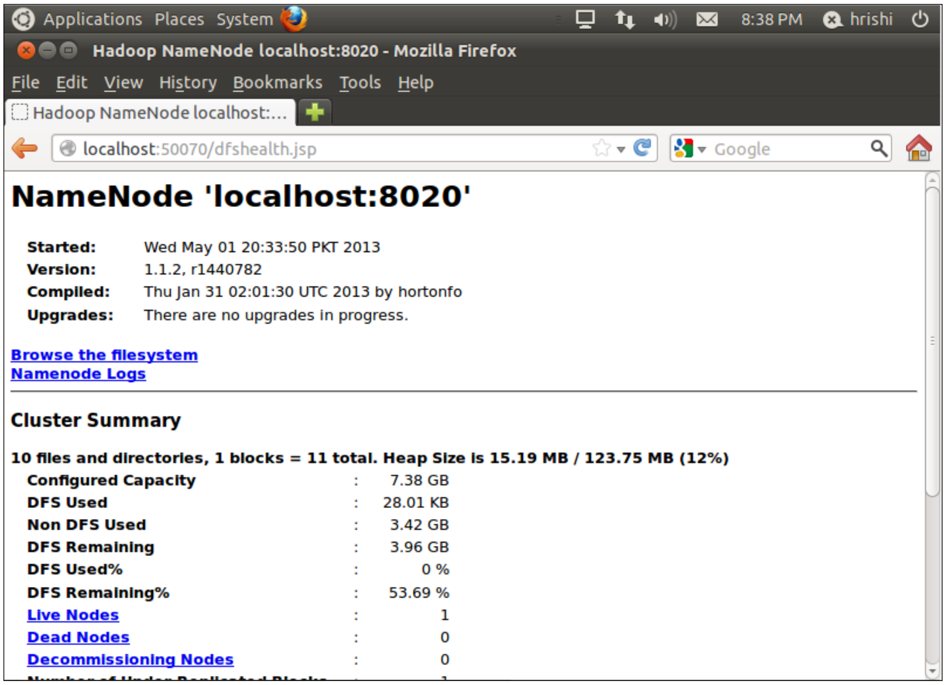
Managing a Hadoop cluster

Once a cluster is launched, administrators should start monitoring the Hadoop cluster. Apache Hadoop provides number of software to manage the cluster; in addition to that there are dedicated open sources as well as third party application tools to do the management of Hadoop cluster.

By default, Hadoop provides two web-based interfaces to monitor its activities. A JobTracker web interface and NameNode web interface. A JobTracker web interface by default runs on a master server (`http://localhost:50070`) and it provides information such as heap size, cluster usage, and completed jobs. It also provides administrators to drill down further into completed as well as failed jobs. The following screenshot describes the actual instance running in a pseudo distributed mode:

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.


Applications Places System
8:38 PM hrishi
Hadoop NameNode localhost:8020 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Hadoop NameNode localhost:...
localhost:50070/dfshealth.jsp
Google

NameNode 'localhost:8020'

Started:	Wed May 01 20:33:50 PKT 2013
Version:	1.1.2, r1440782
Compiled:	Thu Jan 31 02:01:30 UTC 2013 by hortonfo
Upgrades:	There are no upgrades in progress.

Browse the filesystem
Namenode Logs

Cluster Summary

10 files and directories, 1 blocks = 11 total. Heap Size is 15.19 MB / 123.75 MB (12%)

Configured Capacity	:	7.38 GB
DFS Used	:	28.01 KB
Non DFS Used	:	3.42 GB
DFS Remaining	:	3.96 GB
DFS Used%	:	0 %
DFS Remaining%	:	53.69 %
Live Nodes	:	1
Dead Nodes	:	0
Decommissioning Nodes	:	0
Number of Under-Replicated Blocks	:	1

Similarly, the NameNode interface runs on a master server (`http://localhost:50030`), and it provides you with information about HDFS. With it, you can browse the current file system in HDFS through the Web; you can see disk usage, its availability, and live data node related information.

Summary

In this chapter, we have learned about Apache Hadoop, its ecosystem, how to set up a cluster, and configure Hadoop for your requirements. We will look at Apache Solr which provides Big Data search capabilities in the next chapter.

2
Understanding Solr

The exponential growth of data coming from various applications over the past decade has created many challenges. Handling such massive data demanded focus on the development of scalable search engines. It also triggered development of data analytics. Apache Lucene along with Mahout and Solr were developed to address these needs. Out of these, Mahout was moved as a separate Apache top-level project, and Apache Solr was merged into the Lucene project itself.

Apache Solr is an open source enterprise search application which provides user abilities to search structured as well as unstructured data across the organization. It is based on the Apache Lucene libraries for information retrieval. Apache Lucene is an open source information retrieval library used widely by various organizations. Apache Solr is completely developed on Java stack of technologies. Apache Solr is a web application, and Apache Lucene is a library consumed by Apache Solr for performing search. We will try to understand Apache Solr in this chapter, while covering the following topics:

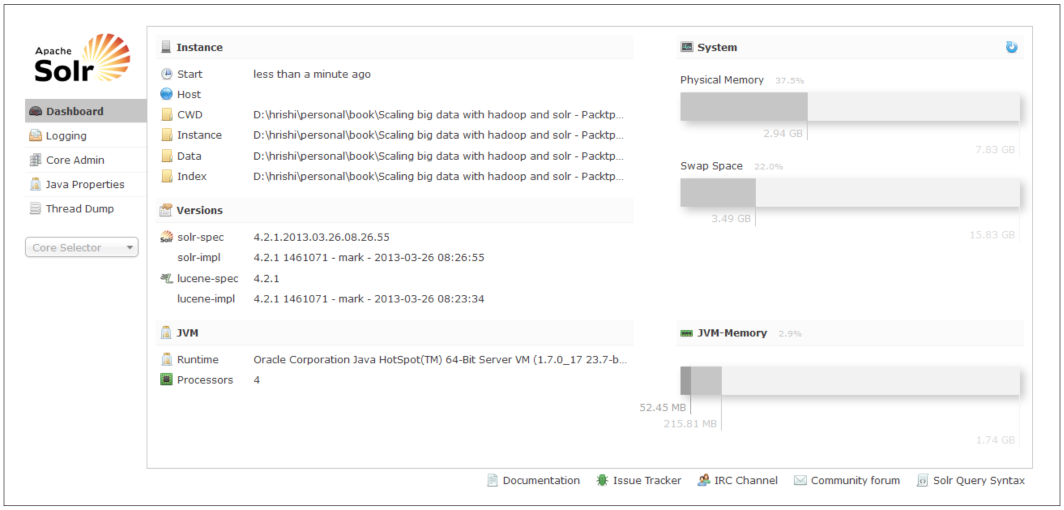
- Installation of an Apache Solr
- Understanding the Apache Solr architecture
- Configuring a Solr instance
- Understanding various components of Solr in detail
- Understanding data loading

Installing Solr

Apache Solr comes by default with a demo server based on Jetty, which can be downloaded and run. However, you can choose to customize it, and deploy it in your own environment. Before installation, you need to make sure that you have JDK 1.5 or above on your machines. You can download the stable installer from `http://lucene.apache.org/solr/` or from its nightly builds running on the same site. You may also need a utility called **curl** to run your samples. There are commercial versions of Apache Solr available from a company called **LucidWorks** (`http://www.lucidworks.com`). Solr being a web-based application can run on many operating systems such as *nix and Windows.

> Some of the older versions of Solr have failed to run properly due to locale differences on host systems. If your system's default locale, or character set is non-english (that is, en/en-US), for safety, you can override your system defaults for Solr by passing `-Duser.language` and `-Duser.country` in your Jetty to ensure smooth running of Solr.

If you are planning to run Solr in your own container, you need to deploy `solr.war` from the distribution to your container. You can simply check whether your instance is running or not by accessing its admin page available at `http://localhost:8983/solr/admin`. The following screenshot shows the Solr window:

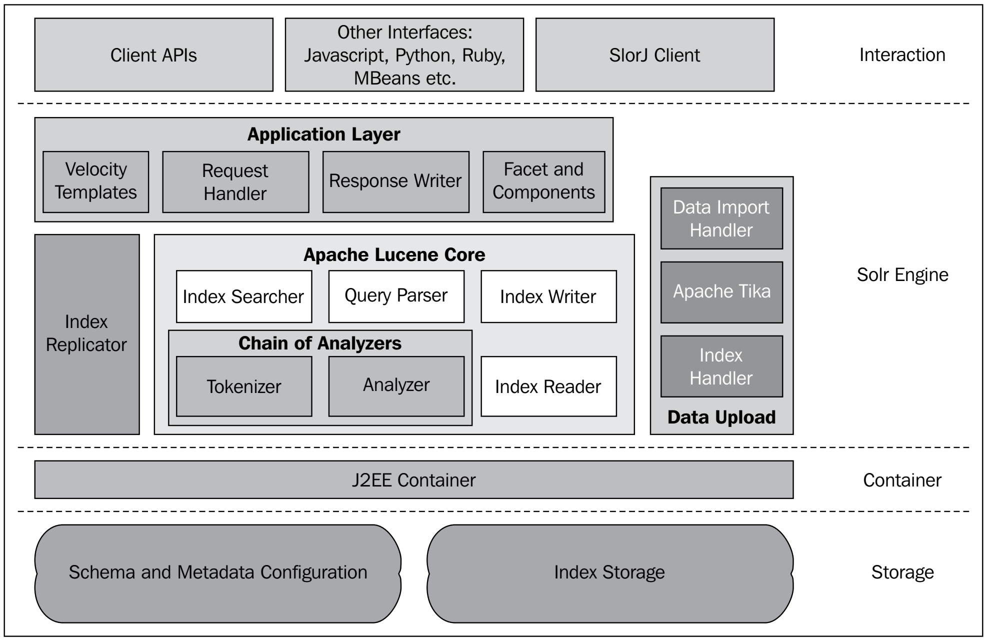


Instance	
Start	less than a minute ago
Host	
CWD	D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtp...
Instance	D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtp...
Data	D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtp...
Index	D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtp...
Versions	
solr-spec	4.2.1.2013.03.26.08.26.55
solr-impl	4.2.1 1461071 - mark - 2013-03-26 08:26:55
lucene-spec	4.2.1
lucene-impl	4.2.1 1461071 - mark - 2013-03-26 08:23:34
JVM	
Runtime	Oracle Corporation Java HotSpot(TM) 64-Bit Server VM (1.7.0_17 23.7-b...
Processors	4

If you are building Solr from source, then you need Java SE 6 JDK (Java Development Kit), Apache Ant distribution (1.8.2 or higher), and Apache Ivy (2.2.0 or higher). You can compile the source by simply navigating to Solr directory and running Ant from the directory.

Apache Solr architecture

Apache Solr is composed of multiple modules, some of them being separate projects in themselves. Let's understand the different components of Apache Solr architecture. The following diagram depicts the Apache Solr conceptual architecture:



Apache Solr can run as a single core or multicore. A Solr core is nothing but the running instance of a Solr index along with its configuration. Earlier, Apache Solr had a single core which in turn limited the consumers to run Solr on one application through a single schema and configuration file. Later support for creating multiple cores was added. With this support, now, one can run one Solr instance for multiple schemas and configurations with unified administrations. You can run Solr in multicore with the following command:

```
java -Dsolr.solr.home=multicore -jar start.jar
```

Storage

The storage of Apache Solr is mainly used for storing metadata and the actual index information. It is typically a file store locally, configured in the configuration of Apache Solr. The default Solr installation package comes with a Jetty server, the respective configuration can be found in the `solr.home/conf` folder of Solr install.

There are two major configuration files in Solr described as follows:

File name	Description
`solrconfig.xml`	This is the main configuration file of your Solr install. Using this you can control each and every thing possible; write from caching, specifying customer handlers, codes, and commit options.
`schema.xml`	This file is responsible for defining a Solr schema for your application. For example, Solr implementation for log management would have schema with log-related attributes such as log levels, severity, message type, container name, application name, and so on.
`solr.xml`	Using `solr.xml`, you can configure Solr cores (single or multiple) for your setup. It also provides additional parameters such as zookeeper timeout, transient cache size, and so on.

Apache Solr (underlying Lucene) indexing is a specially designed data structure, stored in the file system as a set of index files. The index is designed with specific format in such a way to maximize the query performance.

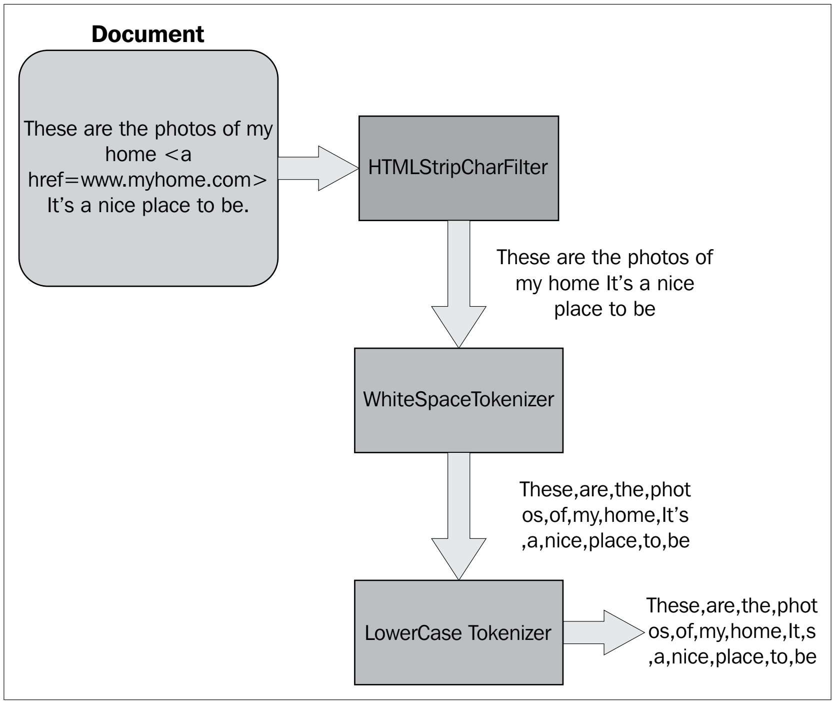
Solr engine

A Solr engine is nothing but the engine responsible for making Solr what it is today. A Solr engine with metadata configuration together forms the Solr core. When Solr runs in a replication mode, the index replicator is responsible for distributing indexes across multiple slaves. The master server maintains index updates, and slaves are responsible for talking with master to get them replicated. Apache Lucene core gets packages as library with Apache Solr application. It provides core functionality for Solr such as index, query processing, searching data, ranking matched results, and returning them back.

The query parser

Apache Lucene comes with variety of query implementations. Query parser is responsible for parsing the queries passed by the end search as a search string. Lucene provides `TermQuery`, `BooleanQuery`, `PhraseQuery`, `PrefixQuery`, `RangeQuery`, `MultiTermQuery`, `FilteredQuery`, `SpanQuery`, and so on as query implementations. `IndexSearcher` is a basic component of Solr searched with a default base searcher class. This class is responsible for returning ordered matched results of searched keywords ranked as per the computed score. `IndexReader` provides access to indexes stored in the file system. It can be used for searching for an index. Similar to `IndexReader`, `IndexWriter` allows you to create and maintain indexes in Apache Lucene.

Tokenizer breaks field data into lexical units or tokens. Filter examines field of tokens from Tokenizer and either it keeps them, transforms them, discards them, or creates new ones. Tokenizer and Filter together form chain or pipeline of analyzers. There can only be one Tokenizer per Analyzer. The output of one chain is fed to another. Analyzing process is used for indexing as well as querying by Solr. They play an important role in speeding up the query as well as index time; they also reduce the amount of data that gets generated out of these operations. You can define your own customer analyzers depending upon your use case. The following diagram shows the example of a filter:


Document
These are the photos of my home <a href=www.myhome.com> It's a nice place to be.
HTMLStripCharFilter
These are the photos of my home It's a nice place to be
WhiteSpaceTokenizer
These,are,the,phot os,of,my,home,It's ,a,nice,place,to,be
LowerCase Tokenizer
These,are,the,phot os,of,my,home,It,s ,a,nice,place,to,be

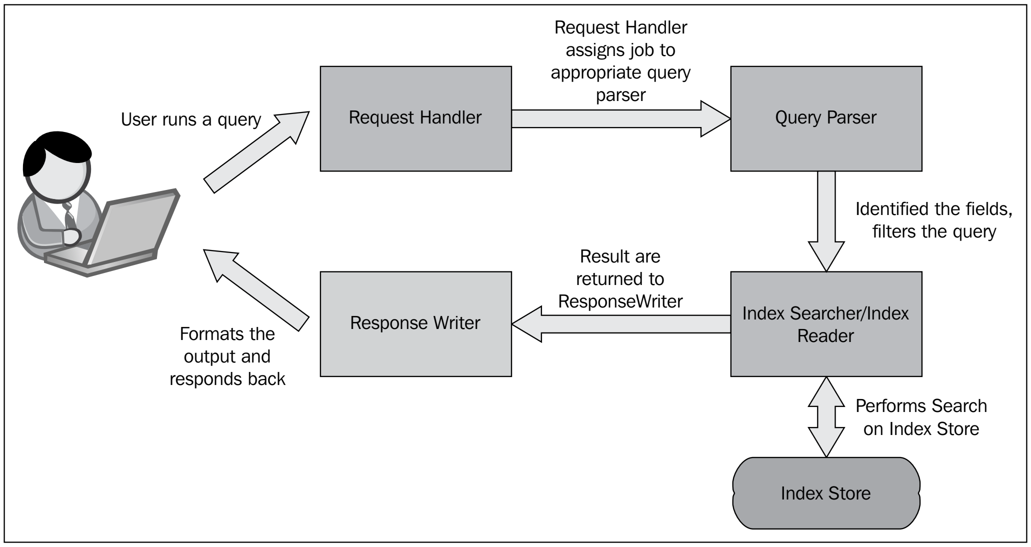
Application layer represents Apache Solr web application. It consists of different UI templates, request/response handlers, and different faceting provided by Solr.

Faceted browsing is one of the main features of Apache Solr; it helps users reach the right set of information they wanted to get. The facets and components deal with providing the faceted search capabilities on top of Lucene. When a user fires a search query on Solr, it actually gets passed on to a request handler. By default, Apache Solr provides DisMaxRequestHandler. This handler is designed to work for simple user queries. It can only search one field by default. You can visit here to find more details about this handler. Based on the request, request handler calls query parser.

Query parser is responsible for parsing the queries, and converting it into Lucene query objects. There are different types of parsers available (`Lucene`, `DisMax`, `eDisMax`, and so on). Each parser offers different functionalities and it can be used based on the requirements. Once a query is parsed, it hands it over to index searcher or reader. The job of **index reader** is to run the queries on index store, and gather the results to response writer.

Response Writer is responsible for responding back to the client; it formats the query response based on search outcomes from the Lucene engine.

The following diagram displays complete process flow when a search is fired from a client:



User runs a query
Request Handler
Request Handler assigns job to appropriate query parser
Query Parser
Identified the fields, filters the query
Index Searcher/Index Reader
Result are returned to ResponseWriter
Response Writer
Formats the output and responds back
Performs Search on Index Store
Index Store

Apache Solr ships with an example schema that runs using Apache velocity. Apache velocity is a fast, open source template engine which quickly generates an HTML-based frontend. Users can customize these templates as per their requirements.

Index handler is one type of update handler that handles the task of addition, updation, and deletion of document for indexing. Apache Solr supports updates through index handler through JSON, XML, and text format.

Data Import Handler (**DIH**) provides a mechanism for integrating different data sources with Apache Solr for indexing. The data sources could be relational databases or web-based sources (for example, RSS, ATOM feeds, and e-mails).

> Although DIH is part of Solr development, the default installation does not include it in the Solr application.

Apache Tika, a project in itself extends capabilities of Apache Solr to run on top of different types of files. When assigned a document to Tika, it automatically determines the type of file (that is, Word, Excel, or PDF) and extracts the content. Tika also extracts document metadata such as author, title, creation date, and so on, which if provided in schema go as text field in Apache Solr.

Interaction

Apache Solr, although a web-based application, can be integrated with different technologies. So, if a company has Drupal-based e-commerce site, they can integrate Apache Solr application and provide its rich faceted search to the user.

Client APIs and SolrJ client

Apache Solr client provides different ways of talking with Apache Solr web application. This enables Solr to easily get integrated with any application. Using client APIs, consumers can run search, and perform different operations on indexes. SolrJ or Solr Java client is an interface of Apache Solr with Java. SolrJ client enables any Java application to talk directly with Solr through its extensive library of APIs. Apache SolrJ is part of Apache Solr package.

Other interfaces

Apache Solr can be integrated with other various technologies using its API library and standards-based interfacing. JavaScript-based clients can straightaway talk with Solr using JSON-based messaging. Similarly, other technologies can simply connect to Apache Solr running instance through HTTP, and consume its services either through JSON, XML, and text formats.

Configuring Apache Solr search

Apache Solr allows extensive configuration to meet the needs of the consumer. Configuring the instance revolves around the following:

- Defining a schema
- Configuring Solr parameters

Let's look at all these steps to understand the configuration of Apache Solr.

Defining a Schema for your instance

Apache Solr lets you define the structure of your data to extend support for searching across the traditional keyword search. You can allow Solr to understand the structure of your data (coming from various sources) by defining fields in the schema definition file. These fields once defined, will be made available at the time of data import or data upload. The schema is stored in the `schema.xml` file in the `etc` folder of Apache Solr.

Apache Solr ships with a default `schema.xml file`, which you have to change to fit your needs. In schema configuration, you can define field types, (for example, `String`, `Integer`, `Date`), and map them to respective Java classes. Apache Solr ships with default data types for text, integer, date, and so on.

```
<field name="id" type="string" indexed="true" stored="true"
  required="true"/>
```

This enables users to define the custom type in case if they wish to. Then you can define the fields with name and type pointing to one of the defined types. A field in Solr will have the following major attributes:

Name	Description
`default`	Sets default value, if not read while importing a document
`indexed`	True, when it has to be indexed (that is, can be searched, sorted, and facet creation)
`stored`	When true, a field is stored in the index store, and it will be accessible while displaying results
`compressed`	When true, the field will be zipped (using gzip). It is applicable for text-based fields
`multiValued`	If a field contains multiple values in the same import cycle of the document/row
`omitNorms`	When true, it omits the norms associated with field (such as length normalization and index boosting)
`termVectors`	When true, it also persists metadata related to document, and returns that when queried.

Each Solr instance should have a unique identifier field (ID) although it's not mandatory condition. In addition to static fields, you can also use Solr dynamic fields for getting the flexibility in case if you do not know the schema affront. Use the `<dynamicField>` declaration for creating a field rule to allow Solr understands which data type to be used. In the following sample code, any field imported, and identified as `*_no` (for example, id_no, book_no) will in turn be read as `integer` by Solr.

```
<dynamicField name="*_no" type="integer" indexed="true"
  stored="true"/>
```

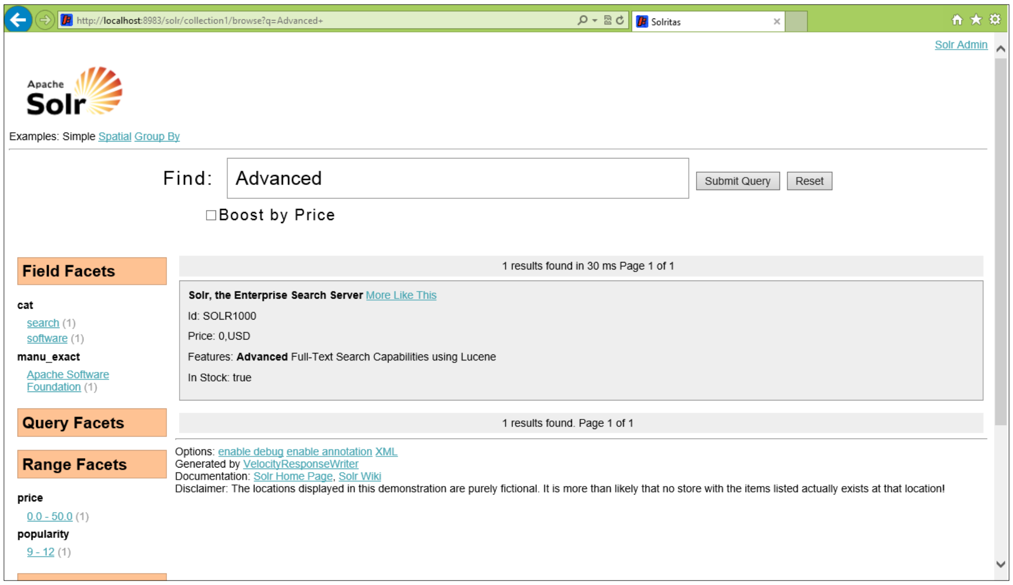
You can also index same data into multiple fields by using the `<copyField>` directive. This is typically needed when you want to have multi-indexing for same data type, for example, if you have data for refrigerator with company followed by model number (WHIRLPOOL-1000LTR, SAMSUNG-980LTR), you can have these indexed separately by applying your own tokenizers to different field. You might generate indexes for two different fields, company name and model number. You can define Tokenizers specific to your field types. Similarly, a Lucene class responsible for scoring the matched results. Solr allows you to override default similarity behavior through the `<similarity>` declaration. Similarity can be configured at the global level; however with Solr 4.0, it extends similarity to be configured at the field level.

Configuring a Solr instance

Once a schema is configured, next step would be to configure the instance itself. To configure the instance, you need to touch upon many files, some of them are configuration files, and some of them are metadata files. The entire configuration is part of `/conf` directory where Solr instance is setup. You can simply run examples by going to the `examples/example-docs` directory and running the following code:

```
java -jar post.jar solr.xml monitor.xml
```

Now, try accessing your instance by typing `http://localhost:8983/solr/collection1/browse`, and you will be able to see the following screenshot when you search on **Advanced**:



Examples: Simple Spatial Group By

Find: Advanced

□Boost by Price

Field Facets

cat

search (1)
software (1)

manu_exact

Apache Software Foundation (1)

Query Facets

Range Facets

price

0.0 - 50.0 (1)

popularity

9 - 12 (1)

1 results found in 30 ms Page 1 of 1

Solr, the Enterprise Search Server More Like This

Id: SOLR1000

Price: 0,USD

Features: **Advanced** Full-Text Search Capabilities using Lucene

In Stock: true

1 results found. Page 1 of 1

Options: enable debug enable annotation XML
Generated by VelocityResponseWriter
Documentation: Solr Home Page, Solr Wiki
Disclaimer: The locations displayed in this demonstration are purely fictional. It is more than likely that no store with the items listed actually exists at that location!

Configuration files

There are two major configurations that go in the Solr configuration: namely, `solrconfig.xml` and `solr.xml`. Among these, `solr.xml` is responsible for maintaining configuration for logging, cloud setup, and Solr core primarily, whereas `solrconfig.xml` focuses more on the Solr application front. Let's look at the `solrconfig.xml` file, and understand all the important declarations you'd be using frequently.

Directive	Description
`luceneMatchVersion`	It tells which version of Lucene/Solr the `solrconfig.xml` configuration file is set to. When upgrading your Solr instances, you need to modify this attribute.
`lib`	In case if you create any plugins for Solr, you need to put a library reference here, so that it gets picked up. The libraries are loaded in the same sequence that of the configuration order. The paths are relative; you can also specify regular expressions. For example, `<lib dir="../../../contrib/velocity/lib" regex=".*\.jar" />.`

Directive	Description
`dataDir`	By default Solr uses `./data` directory for storing indexes, however this can be overridden by changing the directory for data using this directive.
`indexConfig`	This directive is of `xsd complexType`, and it allows you to change the settings of some of the internal indexing configuration of Solr.
`filter`	You can specify different filters to be run at the time of index creation.
`writeLockTimeout`	This directive denotes maximum time to wait for the write Lock for `IndexWriter`.
`maxIndexingThreads`	It denotes maximum number of index threads that can run in the `IndexWriter` class; if more threads arrive, they have to wait. The default value is 8.
`ramBufferSizeMB`	It specifies the maximum RAM you need in the buffer while index creation, before the files are flushed to filesystem.
`maxBufferedDocs`	It specifies the limited number of documents buffered.
`lockType`	When indexes are generated and stored in the file, this mechanism decides which file locking mechanism to be used to manage concurrent read/writes. There are three types: single (one process at a time), native (native operating system driven), and simple (based on locking using plain files).
`unlockOnStartup`	When true, it will release all the write locks held in the past.
`Jmx`	Solr can expose statistics of runtime through MBeans. It can be enabled or disabled through this directive.
`updateHandler`	This directive is responsible for managing the updates to Solr. The entire configuration for `updateHandler` goes as a part of this directive.
`updateLog`	You can specify the directory and other configuration for transaction logs during index updates.
`autoCommit`	It enables automatic commit, when updates are done. This could be based on the documents or time.
`Listener`	Using this directive, you can subscribe to update events when `IndexWriter` is updating the index. The listeners can be run either at the time of `postCommit` or `postOptimize`.
`Query`	This directive is mainly responsible for controlling different parameters at the query time.

Directive	Description
`requestDispatcher`	By setting parameters in this directive, you can control how a request will be processed by `SolrDispatchFilter`.
`requestHandler`	It is described separately in the next section.
`searchComponent`	It is described separately in the next section.
`updateRequest Processor` chain	The `updateRequestProcessor` chain defines how update requests are processed; you can define your own `updateRequestProcessor` to perform things such as cleaning up data, optimizing text fields, and so on.
`queryResponseWriter`	Each request for query is formatted and written back to user through `queryResponseWriter`. You can extend your Solr instance to have responses for XML, JSON, PHP, Ruby, Python, and CSVS by enabling respective predefined writers. If you have a custom requirement for a certain type of response, it can easily be extended.
`queryParser`	The `queryParser` directive tells Apache Solr which query parser to be used for parsing the query and creating the Lucene query objects. Apache Solr contains predefined query parsers such as `Lucene` (default), `DisMax` (based on weights of fields), and `eDisMax` (similar to `DisMax`, with some additional features).

Request handlers and search components

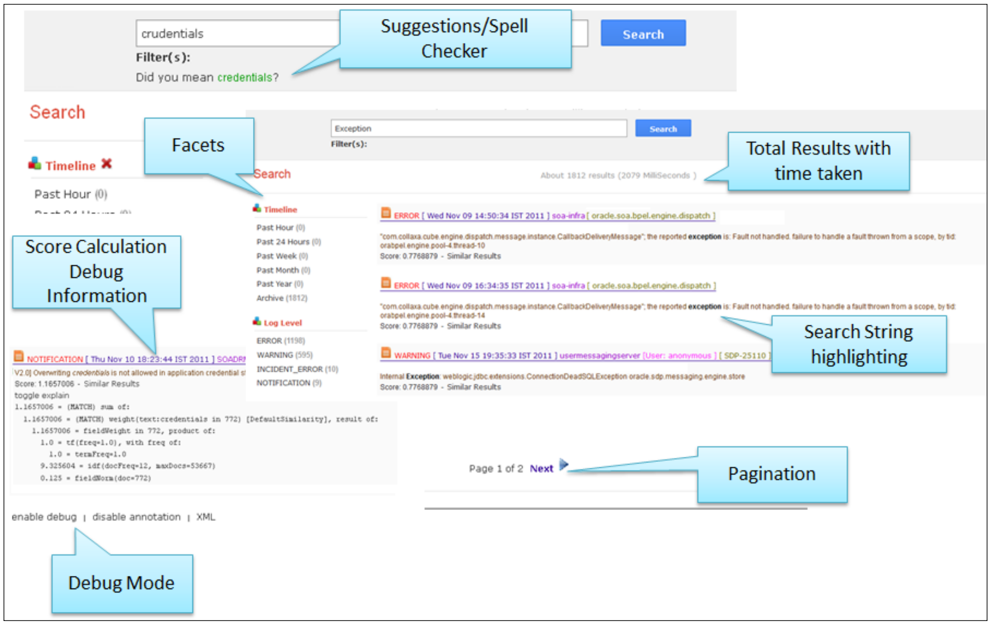
Apache Solr gets requests for searching on data or index generation. In such cases, `RequestHandler` is the directive through which you can define different ways of tackling these requests. One request handler is assigned with one relative URL where it would serve the request. A request handler may or may not provide search facility. In case if it provides, it is also called `searchHandler`. `RealTimeGetHandler` provides latest stored fields of any document. `UpdateRequestHandler` is responsible for updating the process of the index. Similarly, `CSVRequestHandler` and `JsonUpdateRequestHandler` takes the responsibility of updating the indexes with CSV and JSON formats, respectively. `ExtractingRequestHandler` uses Apache Tika to extract the text out of different file formats. By default, there are some important URLs configured with Apache Solr which are listed as follows:

URL	Purpose
/select	SearchHandler in text
/query	SearchHandler for JSON-based requests
/get	RealTimeGetHandler in JSON format
/browse	SearchHandler faceted web-based search, primary interface
/update/extract	ExtractingRequestHandler
/update/csv	CSVRequestHandler
/update/json	JsonUpdateRequestHandler
/analysis/*	For analyzing the field, documents. It makes use of FieldAnalysisRequestHandler
/admin	AdminHandler for providing administration of Solr. AdminHandler has multiple subhandlers defined; /admin/ping is used for health checkup
/debug/dump	DumpRequestHandler echoes the request content back to the client
/replication	Supports replicating indexes across different Solr servers, used by masters and slaves for data sharing. It makes use of ReplicationHandler

A searchComponent is a one of the main feature of Apache Solr. It brings the capability of enhancing new features to Apache Solr. You can use searchComponent in your searchHandler. It has to be defined separately from requestHandler. These components can be defined, and then they can be used in any of the requestHandler directives. Some components also allow access through either searchComponent, or directly as a separate request handler. You can alternatively specify your query parser in the context of your requestHandler. Different parsers can be used for this. The default parser is the Lucene-based standard parser.

Facet

Facets are one of the primary features of Apache Solr. Your search results can be organized in different formats through facets. This is an effective way of helping users to drill down to right set of information. The following screenshot shows one of a customized instance of Apache Solr with facets on the left-hand side:



```
ERROR [ Wed Nov 09 14:50:34 IST 2011 ] soa-infra [ oracle.soa.bpel.engine.dispatch ]
"com.collaxa.cube.engine.dispatch.message.instance.CallbackDeliveryMessage"; the reported exception is: Fault not handled. failure to handle a fault thrown from a scope, by tid:
orabpel.engine.pool-4.thread-10
Score: 0.7768879 - Similar Results

ERROR [ Wed Nov 09 16:34:35 IST 2011 ] soa-infra [ oracle.soa.bpel.engine.dispatch ]
"com.collaxa.cube.engine.dispatch.message.instance.CallbackDeliveryMessage"; the reported exception is: Fault not handled. failure to handle a fault thrown from a scope, by tid:
orabpel.engine.pool-4.thread-14
Score: 0.7768879 - Similar Results

WARNING [ Tue Nov 15 19:35:33 IST 2011 ] usermessagingserver [User: anonymous ] [ SDP-25110 ]
Internal Exception: weblogic.jdbc.extensions.ConnectionDeadSQLException oracle.sdp.messaging.engine.store
Score: 0.7768879 - Similar Results

NOTIFICATION [ Thu Nov 10 18:23:44 IST 2011 ] SOADRI
V2.0] Overwriting credentials is not allowed in application credential s
Score: 1.1657006 - Similar Results
toggle explain
1.1657006 = (MATCH) sum of:
  1.1657006 = (MATCH) weight(text:credentials in 772) [DefaultSimilarity], result of:
    1.1657006 = fieldWeight in 772, product of:
      1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
      9.325604 = idf(docFreq=12, maxDocs=53667)
      0.125 = fieldNorm(doc=772)
```

Using facets, you can filter down your query. Facets can be created on your schema-based fields. So, considering the log-based search, you can create facets based on the log severity. There are different types of facets:

Facet	Description
Field-value	You can have your schema fields as facet component here. It shows the count of top fields.
Range	Range faceting is mostly used on date/numeric fields, and it supports range queries. You can specify start and end dates, gap in the range, and so on.
Date	This is a deprecated faceting, and it is now being handled in the range faceting itself.
Pivot	Pivot gives you the ability to perform simple math on your data. With this facet, you can summarize your results, and then you can get them sorted, and take average. This gives you hierarchical results (also sometimes called hierarchical faceting).

MoreLikeThis

The Solr-based search results are enhanced with the `MoreLikeThis` component, because they provide a better user-browsing experience by allowing the user to choose similar results. This component can be accessed either through `requestHandler`, or through `searchcomponent`.

Highlight

The matched search string can be highlighted in the search results when a user fires a query to Apache Solr through the highlight search component.

SpellCheck

Searching in Solr can be extended further with the support for spell checks using the `spellcheck` component. You can get support for multiple dictionaries together per field. This is very useful in case of multilingual data. It also has a Suggestor that responds to user with **Did you mean** type of suggestions. Additionally, Suggestor with autocomplete feature starts providing users options right at the time when user is typing search query enhancing the overall experience.

Metadata management

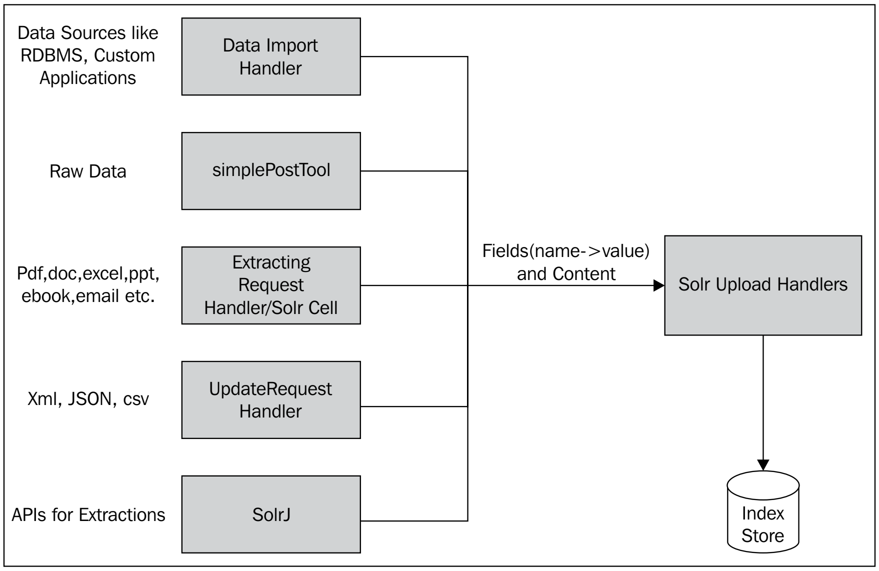
We have already seen the `solr.xml`, `solrconfig.xml`, and `schema.xml` configuration files. Besides these, there are other files where a metadata can be specified. These files again appear in the `conf` directory of Apache Solr.

File name	Description
`protwords.txt`	In this file, you can specify protected words that you do not wish to get stemmed. For example, a stemmer might stem the word catfish to cat or fish.
`currency.txt`	Current stores mapping between exchange rates across different countries; this file is helpful when you have your application accessed by people from different country.
`elevate.txt`	With this file, you can influence the search results and make your own results among the top ranked results. This overrides Lucene's standard ranking scheme taking into account the elevations from this file.
`spellings.txt`	In this file, you can provide spelling suggestions to the end user.

File name	Description
`synonyms.txt`	Using this file, you can specify your own synonyms. For example, `Cost => money`, `Money => dollars`.
`stopwords.txt`	Stopwords are those words which will not be indexed and used by Solr in the applications; this is particularly helpful when you wish to get rid of certain words, for example, in the string, **Jamie and Joseph**, the word and can be marked as a stopword.

Loading your data for search

Once a Solr instance is configured, next step is to index your data, and then simply use the instance for querying and analyzing. Apache Solr/Lucene is designed in such a way that it allows you to plugin any type of data from any data source in the world. If you have structured data, it makes sense to extract the structured information, create exhaustive Solr schema ,and feed in the data to Solr, effectively adding different data dimensions to your search. Data Import Handler (DIH) is used mainly for indexing structured data. It is mainly associated with data sources such as relational databases, XML databases, RSS feeds, and ATOM feeds. DIH uses multiple entity processors to extract the data from various data sources, transform them, and finally generate indexes out of it. For example, in a relational database, a table or a view can be viewed as an entity. DIH allows you to write your own custom entity processors. There are different ways to load the data in Apache Solr as shown in the following diagram:


Data Sources like RDBMS, Custom Applications
Data Import Handler
Raw Data
simplePostTool
Pdf,doc,excel,ppt, ebook,email etc.
Extracting Request Handler/Solr Cell
Xml, JSON, csv
UpdateRequest Handler
APIs for Extractions
SolrJ
Fields(name->value) and Content
Solr Upload Handlers
Index Store

ExtractingRequestHandler/Solr Cell

Solr Cell is one of the most powerful handlers for uploading any type of data. If you wish you can run Solr on a set of files/unstructured data containing different formats such as MS Office, PDF, e-book, e-mail, text, and so on. In Apache Tika, text extraction is based purely on how exhaustive any file is. Therefore, if you have a PDF of scanned images containing text, Apache Tika won't be able to extract any of the text out of it. In such cases, you need to use **Optical Character Recognition (OCR)-**based software to bring in such functionality for Solr. You can simply try this on your downloaded curl utility, and then running it on your document:

```
curl 'http://localhost:8983/solr/update/extract?
  literal.id=doc1&commit=true' -F "myfile=@<your document name
    with extension>"
```

Index handlers such as `SimplePostTool`, `UpdateRequestHandler`, and `SolrJ` provide addition, updation, and deletion of documents to index them for XML, JSON, and CSV format. `UpdateRequestHandler` provide web-based URL for uploading the document. This can be done through curl utility.

> Curl/wget utilities can be used for uploading data to Solr in your environment. They are command line based; you can also use the FireCURL plugin to upload data through your Firefox browser.

Simple post tool is a command-line tool for uploading the raw data to Apache Solr. You can simply run it on any file or type in your input through `STDIN` to load it in Apache Solr.

SolrJ

SolrJ or (SolrJava) is a tool that can be used by your Java-based application to connect to Apache Solr for indexing. It provides a user-friendly interface hiding connection details from consumer application. Using SolrJ, you can index your documents and perform your queries. There are two major ways to do so; one is using the `EmbeddedSolrServer` interface. If you are using Solr in an embedded application, this is the recommended interface suited for you. It does not use HTTP-based connection. The other way is to use the `HTTPSolrServer` interface, which talks with Solr server through HTTP protocol. This is suited if you have a remote client-server based application. You can use `ConcurrentUpdateSolrServer` for bulk uploads whereas `CloudSolrServer` for communicating with Solr running in a cloud setup.

In analyzing and querying your data, we have already seen how Apache Solr effectively uses different request handlers to provide consumers with extensive ways of getting search results. Each request handler uses its own query parser, which extracts the parameters and their values from the query string, and forms the Lucene query objects. Standard query parser allows greater precision over search data; `DisMaxQueryParser` and `ExtendedDisMaxQueryParser` provide a Google-like searching syntax while searching. Depending upon which request handler is called, the query syntax is changed. Let's look at some of the important terms:

Term	**Meaning**
`q?<string>`	Can support wildcard (*:*), for example, `title:Scaling*`
`fl=id,book-name`	Field list that a search response will return
`sort=author asc`	Results/facets to be sorted on authors in ascending order
`price[* TO 100]&rows=10&start=5`	Limits the result to 10 rows at a time, starting at fifth matched result
`hl=true&hl.fl=name,features`	Enables highlighting on field list name and features
`&q=*:*&facet=true&facet.field=year`	Enables faceted search on field `year`
`Publish-date:[NOW-1YEAR/DAY TO NOW/DAY]`	Published date between last year (same day) until today
`description:"Java sql"~10`	Called proximity search. Searches for the descriptions containing Java and sql in a single document with a proximity of 10 words maximum
`"open jdk" NOT "Sun JDK"`	Searches for the `open jdk` term in the document
`&q=id:938099893&mlt=true`	Searches for a specific ID, and also searches for similar results (more like this)

Summary

We have gone through various details of Apache Solr in this chapter. We reviewed the architecture, the configuration, the data loading, and its features. In the next chapter, we will look into how you can bring the two worlds of Apache Solr and Apache Hadoop together to work with Big Data.

3

Making Big Data Work for Hadoop and Solr

The Hadoop platform is widely used for processing large data sets due to its dynamic scaling and reliable data processing. With Hadoop, many organizations have created a massive cluster of commodity machines to process petabytes of data. While the Hadoop platform offers many advantages, the modern businesses demand an enterprise ready search platform together with robust management tools to assist organizations in analyzing Big Data. We are going to look at the problem and different approaches for making Big Data work for Hadoop and Solr.

The problem

Apache Solr is an open source, extendible, and enterprise search having effective community development focused on enhancing it every day. Searching has evolved over time, from basic web-crawling documents search to more sophisticated structured/unstructured content search that provides a lot of user interactions. As the data grows, there is a paradigm shift and more focus is towards the effective use of MapReduce or similar distributed technology for handling such a high volume of data. At the same time, the cost of enterprise storage also needs to be controlled.

By design, Apache Lucene and Solr are designed to support large scale implementation. Apache Solr based distributed environment is useful when:

- Speeding up the search: If Apache Solr is taking longer time for creation of indexes from raw data or for searching on a keyword across the index store, it is possibly the best candidate to run in a distributed environment.
- Index generation time: Incremental generation of indexes at faster speeds is an important aspect during the lifecycle of enterprise search. Distributed Solr can add faster performance.

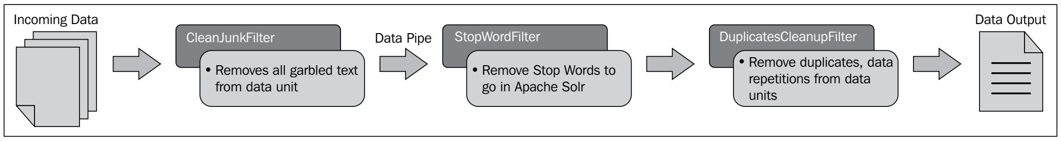
- Large indexes: In cases when you have large indexes, a distribution of search index by means of partitioning adds a lot of value in terms of performance.
- Increase in index creation complexity

At the same time, having your search distributed can address the following problems:

- No single point of failure for your search engine. With effective replication of indexes, this can be achieved.
- High availability of the system in spite of multiple nodes failing due to high replication factor.

Understanding data-processing workflows

Based on the data, configuration, and the requirements, data can be processed at multiple levels while it is getting ready for search. Cascading and LucidWorks Big Data are few such application platforms with which a complex data processing workflow can be rapidly developed on the Hadoop framework. In Cascading, the data is processed in different phases, with each phase containing a pipe responsible for carrying data units and applying a filter. The following diagram shows how incoming data can be processed in the pipeline-based workflow:


Incoming Data
CleanJunkFilter
• Removes all garbled text from data unit
Data Pipe
StopWordFilter
• Remove Stop Words to go in Apache Solr
DuplicatesCleanupFilter
• Remove duplicates, data repetitions from data units
Data Output

Once a data is passed through the workflow, it can be persisted at the end with repository, and later synced with various nodes running in a distributed environment. The pipelining technique offers the following advantages:

- Apache Solr engine has minimum work to handle while index creation
- Incremental indexing can be supported
- By introducing intermediate store, you can have regular data backups at required stages
- The data can be transferred to a different type of storage such as HDFS directly through multiple processing units
- The data can be merged, joined, and processed as per the needs for different data sources

LucidWorks Big Data is a more powerful product which helps the user to generate bulk indexes on Hadoop, allowing them to classify and analyze the data, and provide distributed searching capabilities.

> Sharding is a process of breaking one index into multiple logical units called **shards** across multiple records. In case of Solr, the results will be aggregated and returned.

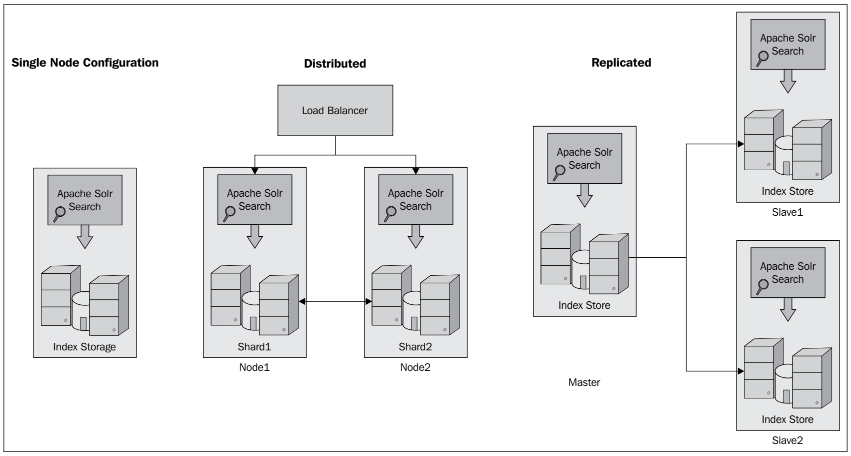
Big Data based technologies can be used with Apache Solr for various operations. Index creation itself can be made to run on distributed system in order to speed up the overall index generation activity. Once that is done, it can be distributed on different nodes participating in Big Data, and Solr can be made to run in a distributed manner for searching the data. You can set up your Solr instance in the following different configurations:

The standalone machine

This configuration uses single high end server containing indexes and Solr search; it is suitable for development, and in some cases, production.

Distributed setup

A distributed setup is suitable for large scale indexes where the index is difficult to store on one system. In this case index has to be distributed across multiple machines. Although distributed configuration of Solr offers ample flexibility in terms of processing, it has its own limitations. A lot of features of Apache Solr such as MoreLikeThis and Joins are not supported. The following diagram depicts the distributed setup:



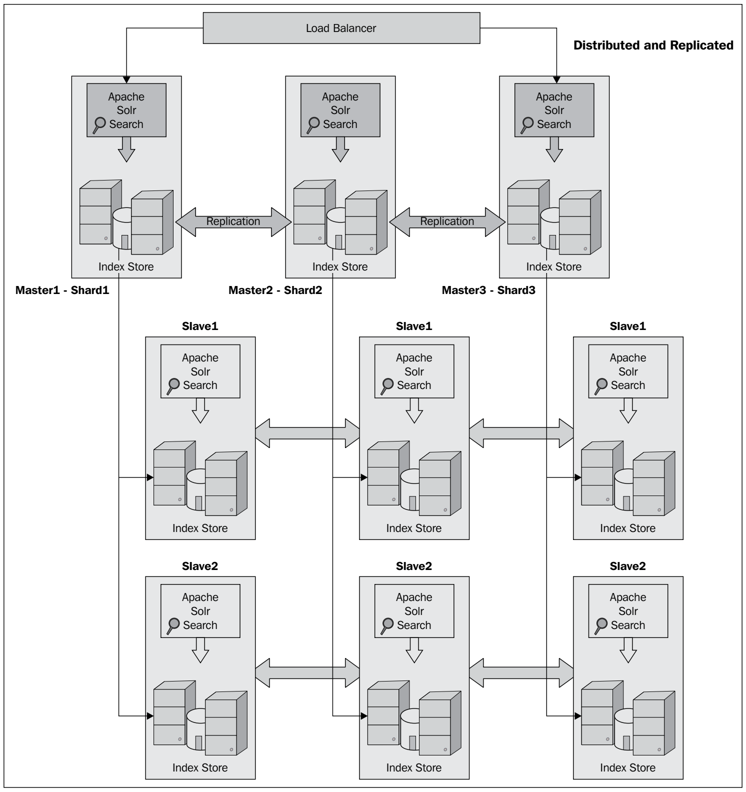
The replicated mode

In this mode, more than one Solr instance exists; among them the master instance provides shared access to its slaves for replicating the indexes across multiple systems. Master continues to participate in index creation, search, and so on. Slaves sync up the storage through various replication techniques such as rsync utility. By default, Solr includes Java-based replication that uses HTTP protocol for communication. This replication is recommended due to its benefits over other external replication techniques. This mode is not used anymore with the release of Solr 4.x versions.

The sharded mode

This mode combines the best of both the worlds and brings in the real value of distributed system with high availability. In this configuration, the system has multiple masters, and each master holds multiple slaves where the replication has gone through. Load balancer is used to handle the load on multiple nodes equally.

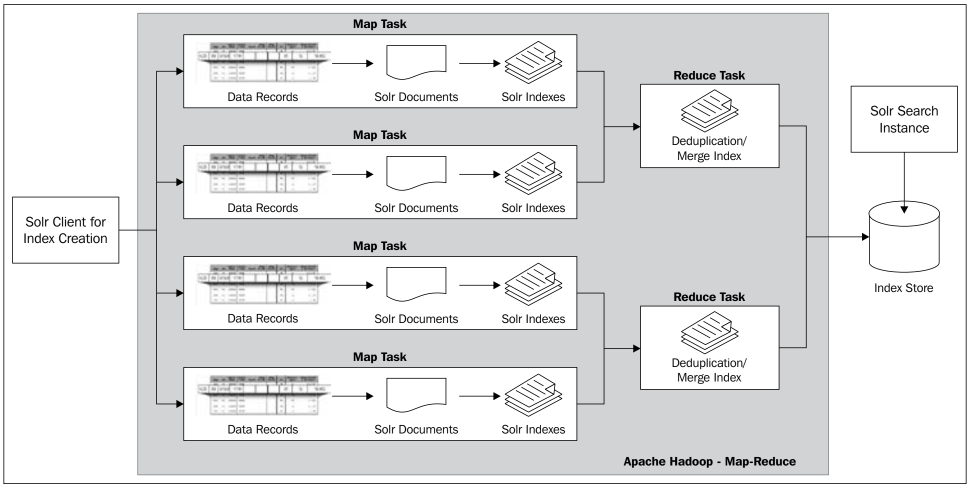
The following diagram depicts the distributed and replicated setup:



If Apache Solr is deployed on a Hadoop-like framework, it falls into this category. Solr also provides SolrCloud for distributed Solr. We are going to look at different approaches in the next section.

Using Solr 1045 patch – map-side indexing

The work for Solr-1045 patch started with a goal to achieve index generation/building using the Apache MapReduce task. Solr-1045 patch converts all the input records to a set of <key, value> pairs in each map task that runs on Hadoop. Further it goes on creating `SolrInputDocument` from the <key, value>, and later creating the Solr indexes. The following diagram depicts this process:



Reduce tasks can be used to perform deduplication of indexes, and merge them together if required. Although merge index seems to be an interesting feature, it is actually a costly affair in terms of processing, and you will not find many implementations with merge index functionality. Once the indexes are created, you can load them on your Solr instance and use them for searching.

You can download this particular patch from `https://issues.apache.org/jira/browse/SOLR-1045`, and patch your Solr instance. To apply a patch to your Solr instance, you need to first build your Solr instance using source. You can download the patch from Apache JIRA. Before running the patch, first do a dry run which does not actually apply patch. You can do it with following command:

```
cd <solr-trunk-dir>
svn patch <name-of-patch> --dry-run
```

If it is successful, you can run the patch without the `-dry-run` option to apply the patch. Let's look at some of the important classes in the patch.

Important class	**Description**
`SolrIndexUpdateMapper`	This class is a Hadoop mapper responsible for creating indexes out of <key, value> pairs of input.
`SolrXMLDocRecordReader`	This class is responsible for reading Solr input XML files.
`SolrIndexUpdater`	This class creates a MapReduce job configuration, runs the job to read the document, and updates the Solr instance. Right now it is built using the Lucene index updater.

Benefits and drawbacks

The following are the benefits and drawbacks of using the Solr-1045 patch:

Benefits

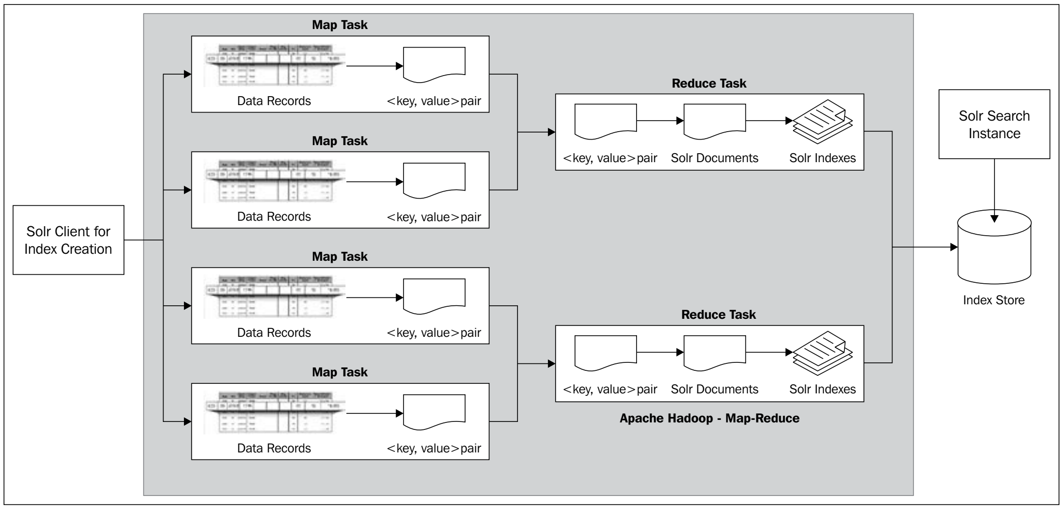
- It achieves complete parallelism by index creation right at the map task.
- Merging of indexes is possible in the reduce phase of MapReduce.

Drawbacks

- When the indexing is done at map-side, all the <key, value> pairs received by reducer gain equal weight/importance. So, it is difficult to use this patch with data that carries ranking/weight information.

Using Solr 1301 patch – reduce-side indexing

This patch focuses on using the Apache MapReduce framework for index creation. Keyword search can happen over Apache Solr or Apache SolrCloud. Unlike Solr-1045, in this patch, the indexes are created in the reduce phase of MapReduce. In this patch, a map task is responsible for converting input records to a <key, value> pair; later, they are passed to the reducer, which in turn converts them into `SolrInputDocument`, and then creates indexes out of it. This index is then passed as outputs of Hadoop MapReduce process. The following diagram depicts this process:



To use Solr-1301 patch, you need to set up a Hadoop cluster. Once the index is created through Hadoop patch, it should then be provisioned to Solr server. The patch contains default converter for CSV files. Let's look at some of the important classes which are part of this patch.

Important class	Description
`CSVDocumentConverter`	This class is responsible for converting output of the map task, that is, key-value pair to `SolrInputDocument`; you can have multiple document converters.
`CSVReducer`	This is a reducer code implemented for Hadoop reducers.
`CSVIndexer`	This is the main class to be called from your command line for creating indexes using MapReduce. You need to provide input path for your data and output path for storing shards.
`SolrDocumentConverter`	This class is used in your map task for converting your objects in Solr document.
`SolrRecordWriter`	This class is an extension of `mapreduce.RecordWriter`; it breaks the data into multiple (key, value) pairs which are then converted into collection of `SolrInputDocument`(s), and then this data is submitted to `SolrEmbeddedServer` in batches. Once completed, it will commit the changes and run the optimizer on the embedded server.
`CSVMapper`	This class parses CSV file and gets key-value pair out of it. This is a mapper class.
`SolrOutputFormat`	This class is responsible for converting key-value pairs to write the data on file/HDFS as zip/raw format.

Perform the following steps to run this patch:

1. Create a local folder with configuration and library folder, `conf` containing Solr configuration (`solr-config.xml`, `schema.xml`), and `lib` containing library.
2. Create your own converter class implementing `SolrDocumentConverter`; this will be used by `SolrOutputFormat` to convert output records to Solr document. You may also override the `OutputFormat` class provided by Solr.
3. Write the Hadoop MapReduce job in the configuration writer:

   ```
   SolrOutputFormat.setupSolrHomeCache(new
     File(solrConfigDir), conf);
   conf.setOutputFormat(SolrOutputFormat.class);
   SolrDocumentConverter.setSolrDocumentConverter(<your
     classname>.class, conf);
   ```

4. Zip your configuration, and load it in HDFS. The ZIP file name should be `solr.zip` (unless you change the patch code).
5. Now run the patch, each of the jobs will instantiate `EmbeddedSolrInstance` which will in turn do the conversion, and finally the `SolrOutputDocument`(s) get stored in the output format.

Benefits and drawbacks

The following are the benefits and drawbacks of using Solr-1301 patch:

Benefits

- With reduced size index generation, it is possible to preserve the weights of documents, which can contribute while performing a prioritization during a search query.

Drawbacks

- Merging of indexes is not possible like in Solr-1045, as the indexes are created in the reduce phase.
- Reducer becomes the crucial component of the system due to major tasks being performed.

Using SolrCloud for distributed search

SolrCloud provides fault-tolerant distributed search capabilities using Apache Solr. SolrCloud supports all types of distributed search configurations. It also has in-built load balancing capabilities that optimize effective load on all the nodes participating in SolrCloud. With the simplest configuration of all, the cloud can be set up.

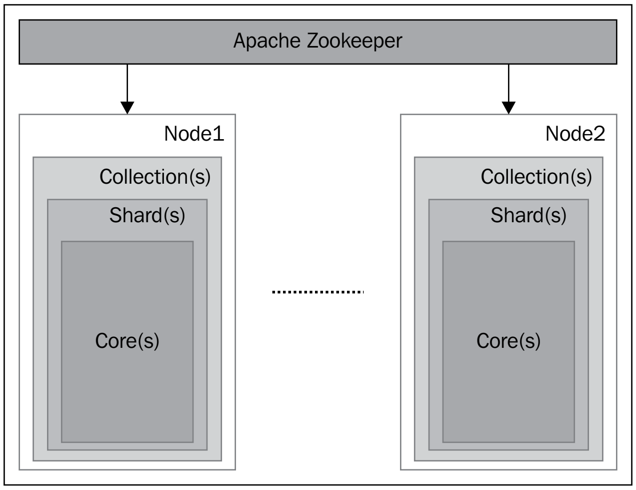
> A collection in Solr is a combination of one or more indexes spanning one or more cores of Apache Solr.

SolrCloud architecture

SolrCloud lets you create a cluster of Solr nodes, each of them running one or more collections. A collection holds one or more shards which are hosted on one or more (in case of replication) nodes.

Any updates to any nodes participating in SolrCloud can in turn sync of rest of the nodes. It uses Apache Zookeeper to bring in distributed coordination and configuration among multiple nodes. This in turn enables near real-time searching on SolrCloud due to active sync of indexes. Apache Zookeeper loads all the configuration files of Apache Solr in its own repository from filesystem, and allows nodes to get access to it in a distributed manner. With this, even if the instance goes away, the configuration will still be accessible to all other nodes. When a new core is introduced in SolrCloud, it registers with a ZooKeeper server by sharing information regarding core and how to contact. In production setup, it is recommended to use external ZooKeeper instead of embedded instance that gets shipped with Apache Solr to run it independently.

The following diagram depicts the architecture:



A replication of shard can simply be created by copying the master Solr instance, and then starting the instance. Once replica starts participating in SolrCloud, you will find the changes in the administration user interface; SolrCloud automatically detects the replicated shards and shows it in the graphical view. With the newer release of Solr post 3.x, there are no masters or slaves. Each node with its replica works in a leader replica mode. One of the nodes containing the shard becomes leader through election. Once the data/document is sent to any of the nodes, the request is forwarded to the respective master node, and then it is processed. The slave node is also requested to process the same. Once a Solr instance runs in a replication mode, it takes care of replicating all the new uploaded indexes across multiple nodes. Solr administration UI allows creation of new cores. This can also be done offline through any HTTP tool such as curl by passing the correct URL.

Configuring SolrCloud

Run the following command from one of the nodes that you wish to make the central node for coordination. This is for running it on Jetty with Tomcat, and you can set similar parameters.

```
java -DzkRun -DnumShards=2 -Dbootstrap_confdir=solr/cloudCore/conf -
  Dcollection.configName=config1 -jar start.jar
```

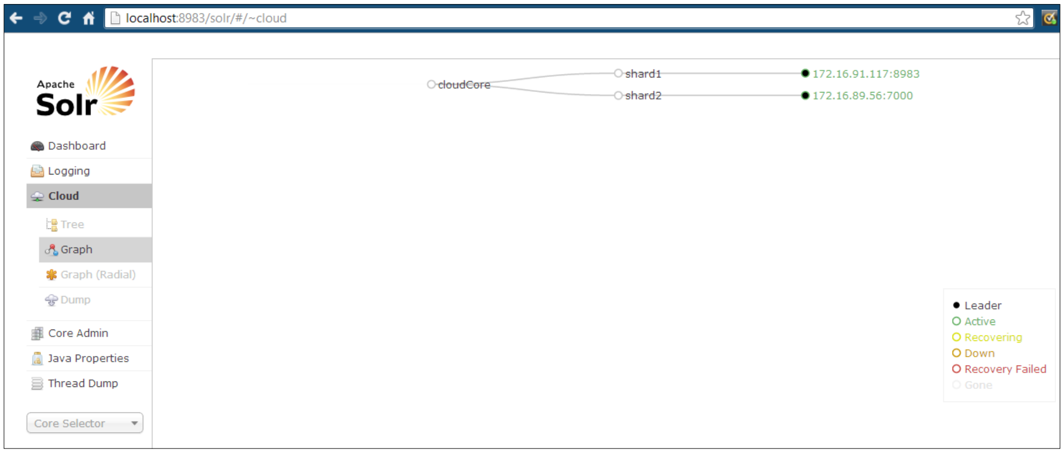
For all other servers participating in cloud, run the following command:

```
java -Djetty.port=7000 -DzkHost=<server-ip>:9983 -jar start.jar
```

Let's look at some of the important parameters that are passed while running Solr in the cloud mode:

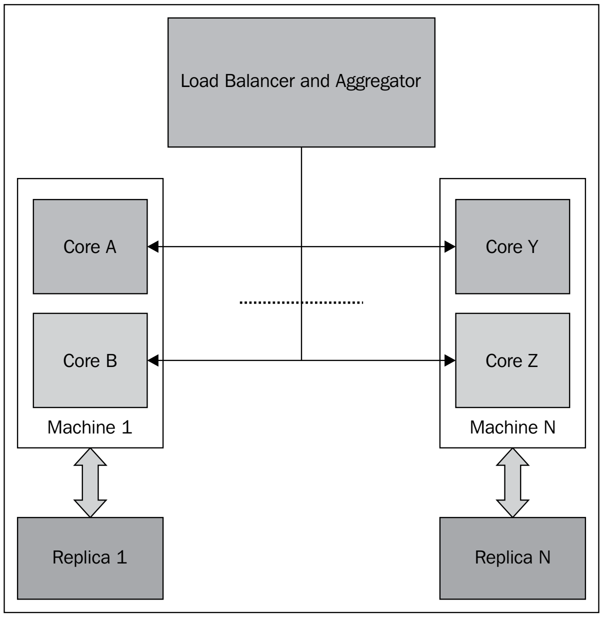
Parameter	Description
`zkRun`	Runs an instance of embedded ZooKeeper as a part of Solr server. Run this on one of the nodes which will serve as central node for all the coordination
`collection.configName`	Sets the configuration to be used for collection (optional)
`bootstrap_confdir=<dir-name>`	The given directory name should contain the complete configuration for SolrCloud, which will include all the configuration files such as `solrconfig.xml`, `schema.xml`, and so on. When Solr runs, the configuration is loaded in ZooKeeper as the name given in `collection.configName`
`zkHost=<host>:<port>`	This parameter points to the instance of ZooKeeper (ZooKeeper ensemble) containing cluster state and configuration
`numShards=<number>`	SolrCloud can be run on one or multiple indexes, the number of shards denote the number of partitions to be carried out on these indexes

The admin UI will start showing the **Cloud** tab. It also shows the distribution of shards in the cloud, and the storage of shards in the context of each system in a graphical view.



Using multicore Solr search on SolrCloud

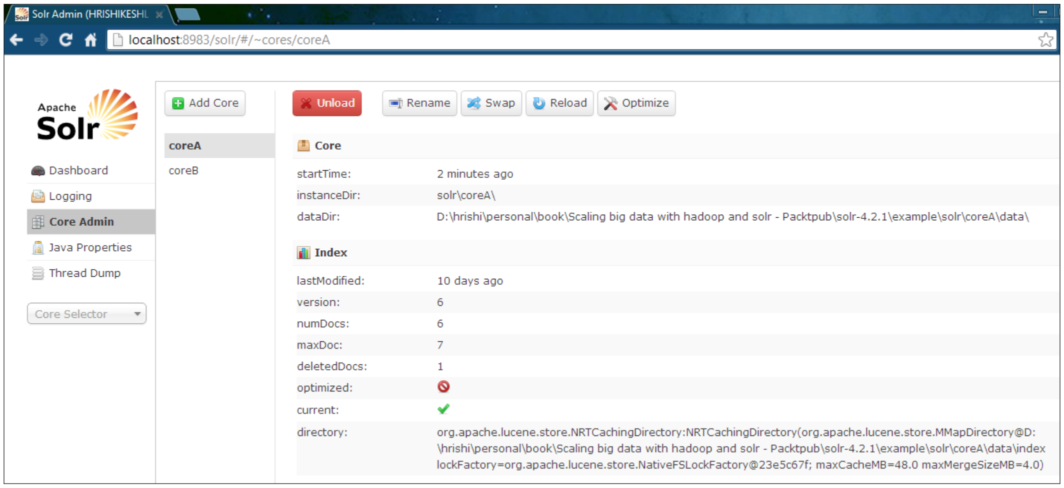
With this approach you can use multiple cores of Apache Solr distributed over one or more than one machine for searching as well as index storage. With multiple cores, you can have one single Solr administration, with multiple instances of Apache Solr running on different configurations on SolrCloud. With Apache Solr multicore architecture, you can achieve replication as well as distributed indexes. The following architecture shows an example multicore Solr setup:



To enable Apache Solr run in multicore mode, you simply need to open the `solr.xml` file, and make the following changes in the file on each of your machine:

```
<solr persistent="true">
  <cores adminPath="/admin/cores" host="${host:}"
    hostPort="${jetty.port:}">
    <core name="coreA" instanceDir="coreA" />
    <core name="coreB" instanceDir="coreB" />
  </cores>
</solr>
```

Once you add a core, you need to copy the Solr configuration directories to multiple instance directories that you have specified in the configuration. Now you can start the Solr instance, and you will find multiple cores in the admin interface, as shown in the following screenshot:



Core

startTime: 2 minutes ago
instanceDir: solr\coreA\
dataDir: D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtpub\solr-4.2.1\example\solr\coreA\data\

Index

lastModified: 10 days ago
version: 6
numDocs: 6
maxDoc: 7
deletedDocs: 1
optimized: ⊘
current: ✓
directory: org.apache.lucene.store.NRTCachingDirectory:NRTCachingDirectory(org.apache.lucene.store.MMapDirectory@D:\hrishi\personal\book\Scaling big data with hadoop and solr - Packtpub\solr-4.2.1\example\solr\coreA\data\index lockFactory=org.apache.lucene.store.NativeFSLockFactory@23e5c67f; maxCacheMB=48.0 maxMergeSizeMB=4.0)

You can specify the default core to run from user interface (browse). When you run a query, you can see that it is running on the default core setup in the configuration. To run a query on a multicore setup, you need to first set up the SolrCloud. Your indexes are loaded as shards, on different cores of your Solr. You can use your own client or SolrJ to run your query on multiple systems in a distributed manner. This can be achieved by passing the `shards` parameter in your query (that is, `shards=host: port/location, host:port/location…`).

> With multicore on SolrCloud, there is a possibility of deadline due to shards requesting each other for running a query. In such cases users need to make sure that number of HTTP request threads for J2EE container is greater than those received from clients.

All the components except `MoreLikeThis` are supported on distributed search. Setting Apache SolrCloud in multicore mode will enable the query request to perform search across multiple machines. Please look at some of the following examples:

Functionality	URL
Run your search on multiple nodes	`http://instance1:8983/solr/core1/select?shards=instance1:8983/solr/core1,instance2:8983/solr/core2&q=scaling big data`
Get the status of core	`http://localhost:8983/solr/admin/cores?action=STATUS&core=core0`

Functionality	URL
Splitting the indexes into two cores	`http://localhost:8983/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1&targetCore=core2`
Merging the indexes from different cores	`http://localhost:8983/solr/admin/cores?action=mergeindexes&core=core0&srcCore=core1&srcCore=core2`

Solr core offers you an easy way of sharding your indexes across multiple nodes. It is up to the developer to decide upon how the indexes will be distributed. For example, a simple formula of *uniqueId.hash() % no_of_servers* can get uniformed distribution across multiple machines.

Benefits and drawbacks

Let's look at the benefits and drawbacks of using SolrCloud for distributed search.

Benefits

- High availability and fault tolerance for data
- Easy to configure and manage through common administration UI
- Allows scaling of Solr to multi-server environment
- Provides massive horizontal scaling of data

Drawbacks

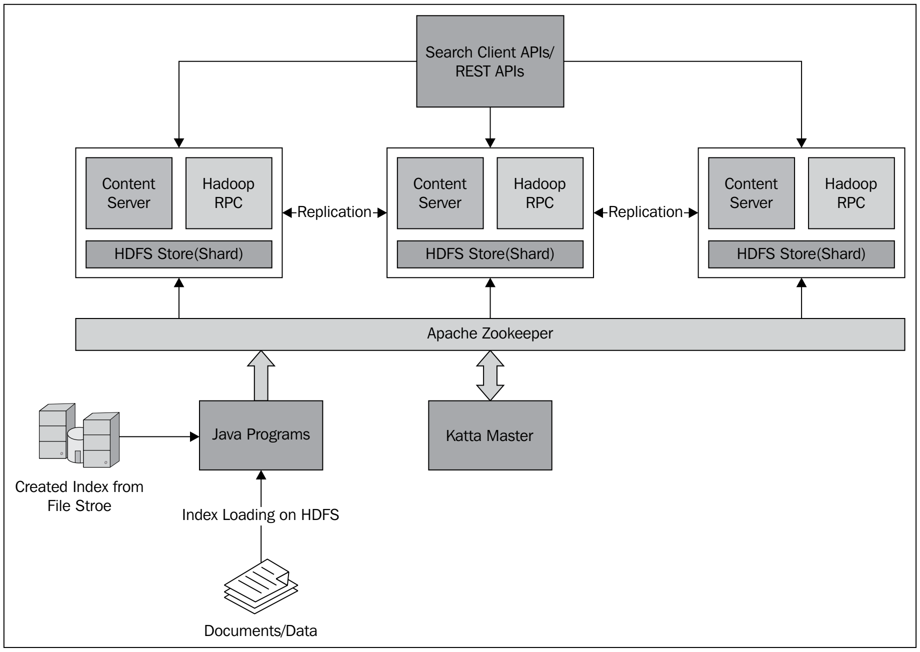
- As the scale increases, it becomes difficult to manage multiple cores
- Compared to single core, performance is low; however, it can be tuned
- Configuration of load balancer, and sync of data replication to be done by an external tool

Using Katta for Big Data search (Solr-1395 patch)

Katta is an open source project that enables you to store your data in a distributed manner without any failures. Although we do not see a lot of active development happening in the project, a lot of organizations have taken Katta and customized it to address their needs for distributed search. With Katta together with Hadoop and Solr, one can achieve distributed and replicated configuration of Apache Solr. There are two important tasks that can be deployed in the Hadoop framework with the help of Katta; they are indexing and searching.

Katta architecture

The following diagram depicts the Katta architecture:



Each Katta Hadoop cluster has a master node and the rest of the other nodes participate actively in the storage of data. A master node is responsible for managing the nodes as well as determining the assignment of index shards to them. Each node is responsible for sharing a shard. A content server on each node determines the type of shard supported by the given node.

Katta master communicates with all the nodes by means of Apache ZooKeeper. A virtual directory base is created among all the nodes including master, where each node updates their status instead of using heartbeat mechanism of ZooKeeper. The work distribution among them is done through a blocking queue. Each operation such as index deployment, shard undeployment is pushed to individual's queue, and then each node processes the queue for next task sequentially.

When search query is fired, client multicasts the query on Hadoop cluster and requests results from each node containing shard. It uses the Hadoop RPC-based mechanism for faster communication. Each node returns matched results with the scores. Katta supports distributed scoring; so once these results are retrieved, they are merged together based on scoring and returned back to the client.

Configuring Katta cluster

You can either download the distribution from `http://katta.sourceforge.net`, or build the executable by compiling the source using the following command:

```
ant compile
```

Once it is compiled, you need to copy the folder to all your nodes including master. Before starting master, verify the settings in `katta.master.properties`, and in the nodes file, add all the nodes. Similarly, for ZooKeeper, if you intend to run an embedded ZooKeeper, then you will need to modify the `zookeeper.servers` attribute in the `katta.zk.properties` file for all nodes. You need to point to the master node. Now, start the master using the following command:

```
bin/katta startMaster
```

This will start the master at first. You should start the individual nodes on all machines using the following command:

```
bin/katta startNode
```

Once all the nodes are started, you can start adding indexes to Katta.

Creating Katta indexes

Katta defines two types of shards: namely, Hadoop map files and Lucene index. It also allows you to create your own type of shard. The simplest implementation is using the Lucene indexes. You can simply transform your Lucene index into Katta index by combining them into one folder and loading them on Hadoop cluster. The index creation itself can be run on Hadoop cluster. First, you can start with setting up Katta for your machine.

Now set up a Hadoop cluster and format the NameNode. The next task would be to load your data in a Hadoop cluster directly, or through a Hadoop sequence file. Katta provides in-built tool to do that. You can simply create the Lucene indexes, and then convert them to the Katta indexes. Once the index is created, it has to be deployed on Hadoop cluster to be searchable. You can deploy it using the following sequence:

```
bin/katta addIndex <index-name> hdfs://<location-of-index>
```

You can check the addition of the index by searching for some text which is indexed.

```
bin/katta search <index-name> <field:search-string>
```

Katta also provides a web-based interface for monitoring and administration purposes. It can simply be started by running the following command:

```
bin/katta startGui
```

It provides masters and nodes information shards and indexes on administration UI. This application is developed using the Grails technology.

Benefits and drawbacks

Let's look at the benefits and drawbacks of using Katta for distributed search.

Benefits

- Completely customizable framework based on Hadoop/HDFS
- Provides failover for master as well as replication of nodes
- Can be used as production instance

Drawbacks

- Difficult to get real-time updates and requires lot of tweaking
- Not actively developed by open source any more
- Needs a lot of customization, provides basic vanilla setup

Summary

In this chapter, we have understood different possible approaches of how Big Data can be made to work with Apache Hadoop and Solr. We also looked at the benefits and drawbacks of these approaches. In the next chapter, we will get into more details about how you can effectively use these technologies for building large indexes out of Big Data.

4

Using Big Data to Build Your Large Indexing

This chapter talks about how you can use Big Data technologies to effectively build your indexes. It starts with explaining the concept of NOSQL; it takes you through the deep dive of sharding your indexes. It primarily covers the following topics:

- Understanding the concept of NOSQL
- Understanding the concepts of distributed search
- Lily bringing Hadoop, Solr, and Hbase together
- Deep dive in sharding and indexing of Big Data
- Configuring your cloud for large indexes

Understanding the concept of NOSQL

Traditional relational databases allow users to define a strict data structure and a SQL-based querying mechanism. NOSQL databases rather than confining users to define the data structures, allow an open database with which they can store any kind of data and retrieve it by running queries that are not SQL based.

The CAP theorem

Before we get into the role of NOSQL, we must first understand the CAP theorem. In the theory of computer science, the **CAP theorem** or **Brewer's theorem** talks about distributed consistency. It states that it is impossible to achieve all of the following in a distributed system:

- Consistency: Every client sees the most recently updated data state
- Availability: The distributed system functions as expected, even if there are node failures
- Partition tolerance: Intermediate network failure among nodes does not impact system functioning

Although all three are impossible to achieve, any two can be achieved by the systems. That means in order to get high availability and partition tolerance, you need to sacrifice consistency. There are three types of systems:

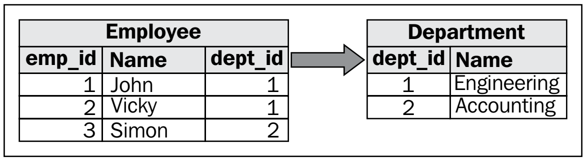
- CA: Data is consistent between all nodes, and you can read/write from any node, while you cannot afford to let your network go down. (For example: relational databases, columnar relational stores)
- CP: Data is consistent and maintains tolerance for partitioning and preventing data going out of sync. (For example: Berkeley DB (key-value), MongoDB (document oriented), and HBase (columnar))
- AP: Nodes are online always, but they may not get you the latest data; however, they sync whenever the lines are up. (For example: Dynamo (key-value), CouchDB (document oriented), and Cassandra (columnar))

High availability can achieved through data replication; consistency is achieved by updating multiple nodes for changes in data. Relational databases are designed to achieve CA capabilities. NOSQL databases can either achieve CP or AP.

What is a NOSQL database?

In an enterprise, the data is generated from all the software that is participating in the day-to-day operation. This data has different formats, and bringing in this data for Big Data processing requires a need for a storage system that is flexible enough to accommodate data with varying data models. NOSQL database by its design is the best suited for this storage. One of the primary objectives of NOSQL is horizontal scaling, that is, P in the CAP theorem at the cost of sacrificing Consistency or Availability.

NOSQL databases are highly optimized for retrieval, and intended to work with huge datasets where the nature of the data is not known. Due to the design of these databases, they are extremely flexible in terms of creating data models. NOSQL databases are categorized under three major categories described as follows, although there is an overlap in terms of the data store and functionalities. To get better understanding, let's look at the simple comparison of data storage for a relational database schema described in the following screenshot:



Employee		
emp_id	Name	dept_id
1	John	1
2	Vicky	1
3	Simon	2

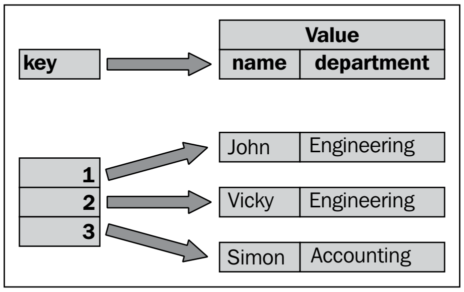
Department	
dept_id	Name
1	Engineering
2	Accounting

The key-value store or column store

Typically, the storage of data goes in terms of (key-value) pairs, where each key uniquely identifies each record, and the value is the record itself. This is one the most widely used database types while working with Big Data. There are further subtypes to this store such as hierarchical, tabular, volatile (in-memory), and persistent (storage). Implementation of key-value stores is Apache HBase, levelDB, Dynamo, and so on.

> BigTable implementation is proprietary column-driven data storage system based on the Google file system or similar. This type of database stores data as a sorted map, with three dimensions (row, column, and timestamp) into a highly compressed storage. This is also one of the widely used data store for Big Data storage and analysis.

The key-value pair for the store will look like the following diagram:

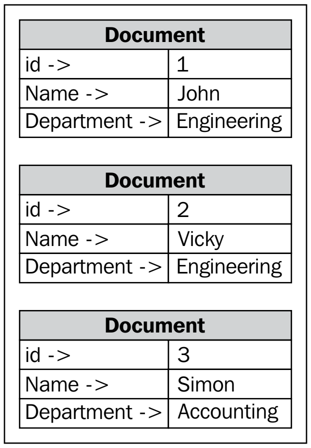


	Value	
key	name	department
1	John	Engineering
2	Vicky	Engineering
3	Simon	Accounting

The document-oriented store

A term **document** in document-oriented store represents a data record. However, a document may not stick to a standard schema; it can have its own structure. Each document is identified via a unique key (URI or a path). This type of store allows data to be queried through a API layer. The implementations are Apache Cassandra, CouchDB, MongoDB, and so on.

The example storage of relational schema is shown in the following screenshot:



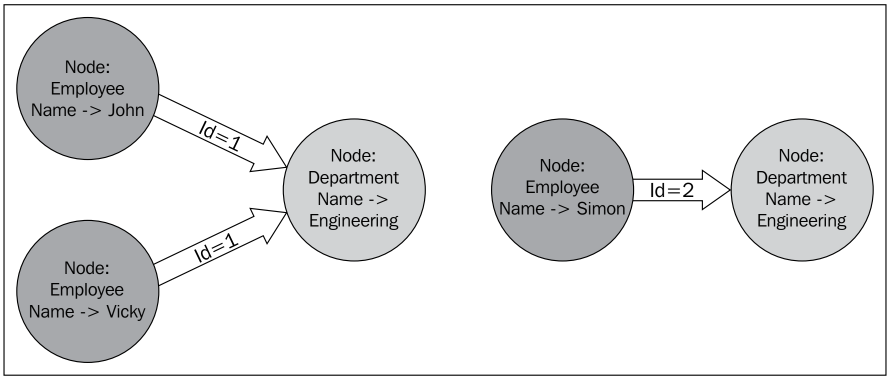
Document	
id ->	1
Name ->	John
Department ->	Engineering

Document	
id ->	2
Name ->	Vicky
Department ->	Engineering

Document	
id ->	3
Name ->	Simon
Department ->	Accounting

The graph database

This type of database allows users to define and link data records by means of graph nodes and edges. A node can contain record information, and a link (edge) between the nodes can also carry its weight or cost, as shown in the following screenshot. Other flavors of graph database include triple store and network databases. The implementations are Allegrograph, InfiniteGraph, and Neo4J.



Why NOSQL databases for Big Data?

As we have seen, data models for NOSQL differ completely from that of the relational database. With the flexible data model, it becomes very easy for developers to quickly integrate with the NOSQL database, and bring in heavy data from different data sources. This enables NOSQL databases to be ideal for Big Data storage since it demands different data types to be brought together under one umbrella.

In addition to the flexible schema, NOSQL offers scalability and high performance which is again one of the most important factors to be considered while running Big Data. NOSQL was developed to be a distributed type of database. When traditional relational stores rely on high computing power of CPUs and high memory focus on centralized system, NOSQL can run on your low cost, commodity hardware. These servers can be added or removed dynamically from the cluster running NOSQL, enabling NOSQL database easier to scale. NOSQL enables most advanced features of database such as data partitioning, index sharding, distributed query, caching, and so on.

Although NOSQL offers optimized storage for Big Data, it does not replace the relational database. While relational databases offer transactional ACID properties, high CRUD, data integrity, structured database design approach, which are required in many applications, NOSQL does not ensure any one of them. Hence, it is most suited for Big Data where there is less possibility of need for the data being transactional.

How Solr can be used for Big Data storage?

As NOSQL does not support any kind of SQL way of querying, it provides various ways of querying user data such as API based or SQL like querying. However, since the data is unstructured, it becomes difficult for users to query for data with the given querying capabilities. In such cases, a fast, efficient search on this data becomes the need for the users.

By design, Solr supports any data to be loaded in the search engine through different handlers making it a data format agnostic. Solr can be scaled easily on top of the commodity hardware as we have seen in previous chapter. Thus, Solr becomes one of the most efficient and eligible NOSQL-based search available today. The data can be stored in the Solr indexes, and can be queried through Lucene search APIs. Solr does perform joins, because of its denormalization of data. Additionally, its rich faceting provides a good drill down for end users to the desired results. Solr provides interesting features such as ranking results, fault tolerance, dynamic fields, high availability, atomic updates, specialized queries, and its compliance with the entire NOSQL definition that makes Apache Solr one of the best suited NOSQL data store available today. Roughly, Solr ensures CP in the CAP theorem. For availability, shards are replicated on multiple nodes.

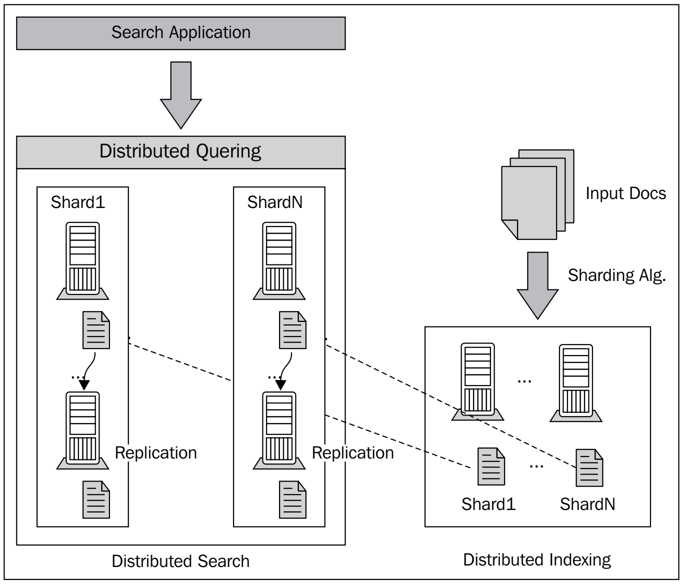
Many NOSQL databases such as MongoDB offer better performance in terms of data loading, data updates, transactional environment, and so on; whereas Solr performs better in terms of unanticipated queries, search on data, and various filter/sort combinations of data. Apache Solr's commit operation is costly, and its optimization is also costlier. The frequency of these operations has an impact on the overall performance of the system. The data is unavailable unless committed to the Solr repository. Sometimes, organizations use a combination of NOSQL and Solr together, to achieve the best of both worlds.

Understanding the concepts of distributed search

Distributed search is considered an option when search with single index store becomes difficult to operate in terms of speed and sizing. There are two major operations that take place in any search engine; first is indexing the data, and second is searching.

Distributed search architecture

When running search on the distributed systems, any or all of the operations can be run in a distributed manner depending upon why you wish to run your search in a distributed environment. Let's look at the architecture for distributed search in the following diagram:



To utilize the distributed search, the indexing must be split into multiple shards and should be kept across multiple nodes of a distributed system. In order to generate this index, a search application may use a distributed system such as Apache Hadoop, and then based on the generated index, it may push it to the search engine repository.

Similar to the distributed index generation, even search can happen in a distributed manner. The shard is a complete index, and it can be queried independently; however, it does not form a complete result set of the search, so the search application has to be smart to query multiple nodes, combine the results, and return them to the client. We looked at different approaches in *Chapter 3*, *Making Big Data Work for Hadoop and Solr*; the following table describes their support for the distributed search:

Particular	Distributed indexing	Distributed search
Solr-1045: map-side indexing	Supported	Not applicable since it is limited to index generation
Solr-1301: reduce-side indexing	Supported	Not applicable since it is limited to index generation
Katta	Supported	Supported
SolrCloud	Supported	Supported

Distributed search scenarios

We have seen how distributed search architecture functions. Let's look at some of the scenarios of distributed search in the following table:

Scenario	Information
Single machine	Index generation and search that goes on one machine
Master-slave	Index generation on one machine, and search happens to be on another machine
Multi-node	All nodes are masters and index is divided among them
Sharding-replica	Index generation is distributed and there are replicas of master to ensure high availability
Multi-tenant	In this, multiple indexes are run by different users. Multi-tenant architecture can be used in combination with any of the architectures listed previously

Single machine architecture is difficult to scale it starts slowing down as the size of the index grows. Master-slave architecture provides features such as high availability, although it does not ensure near real-time search. Master-slave architecture can further be extended for one master-multiple slaves, one master-one slave, and so on. In multi-node architecture, index is distributed among multiple nodes based on some hashing algorithm. Sharding replica-based architecture allows you to control how wide (distributed) and how deep (replication) you intend to configure your search architecture.

Lily – running Solr and Hadoop together

Lily is an open source distributed application by NGDATA that brings in together the capabilities of Apache Hadoop, HBase, ZooKeeper, and Solr together to allow end user applications (web portals, content management systems, and so on) to enable enterprise-wide access to its distributed search through standard interfaces.

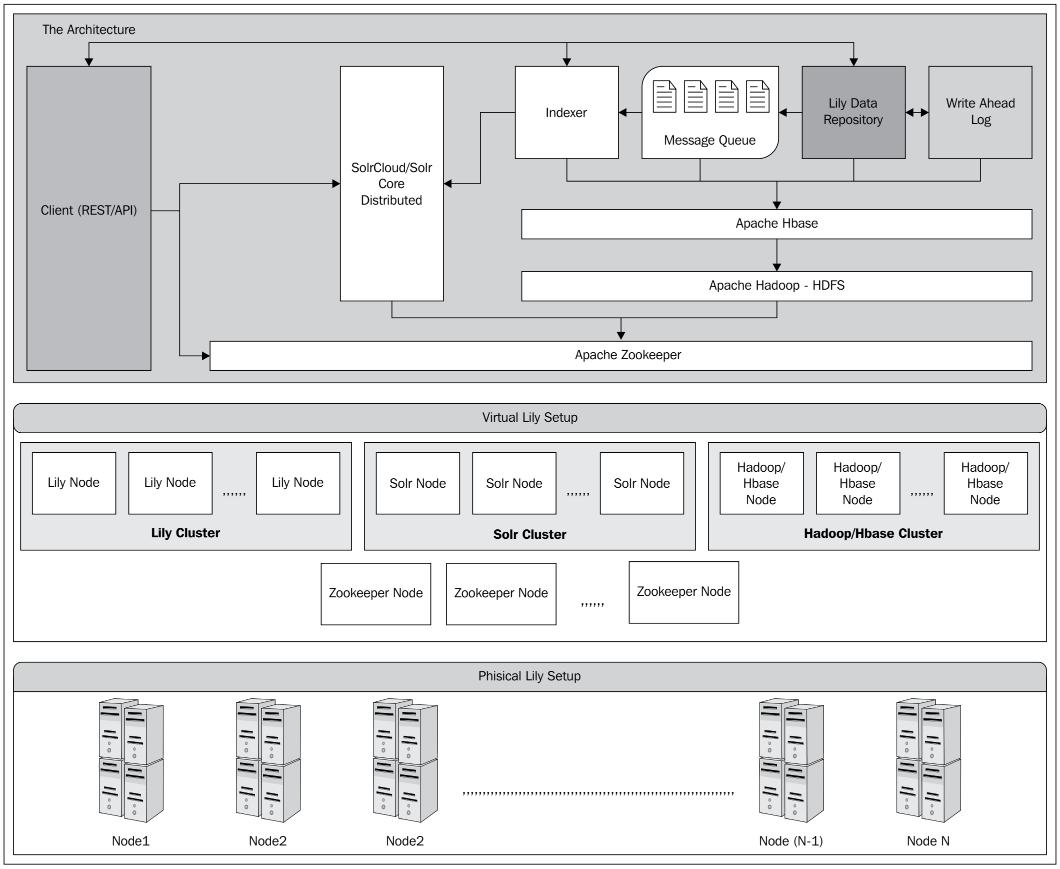
The architecture

Lily provides scalability and replication through its distributed architecture. Lily has multiple nodes; each node is responsible for participating in one or more of the functionalities. Primarily, Lily is designed to work as a content management system. The storage is Apache HBase which is running on top of the Hadoop framework, and the query/search mechanism is based on Apache Solr. Lily exposes complete functionality of Apache Solr on top of its record base. Lily provides functional layering, scalability, and fault tolerance on top of these. Lily provides basic record management, with support for open standards such as **JCR** (**Java Content Repository**). It exposes the functionality to its client through RESTful APIs. Let's look at the components of the Lily application.

> **Lily Data Repository** (**Lily DR**) provides a distributed scalable storage to store, query, and retrieve the data records using the Apache Solr indexes.

Lily uses HBase for storing its record data. Since HBase is a NOSQL database, Lily can allow any type of rich schema to be added to Lily Data Repository. HBase, although not a transactional database, offers data scalability through its architecture. HBase runs on top of HDFS effectively using a Hadoop framework for running the Lily application. Each record in Lily Data Repository must have a unique identification. Lily allows the record information to be stored in HBase or HDFS.

Lily can be contacted by a client for many reasons. The client may use RESTful APIs to query for data, update the records, and delete the records.



As the preceding diagram depicts, the physical setup can be completely different from the virtual node and the Lily architecture. A physical node might be running one or more Lily system processes.

> Apache Avro is a data serialization system supported by Lily. Lily clients can use Avro-based protocols to connect to Lily.

Write-ahead Logging

Since HBase is not transactional, ACID is difficult to ensure in Lily. Lily uses a write-ahead log to improve upon its support for transactions. Whenever Lily receives a request from the client to insert or update new records, it first updates **Write-Ahead Log** (**WAL**) of Lily with the intentions of what it is planning to do for the request. Then it goes ahead and makes the change; later, it again requests WAL to complete the action. This kind of design is useful in case of failures in between the update/insert record operations. In such cases, Lily first reviews the write-ahead log, and performs the pending tasks.

The message queue

Operations such as adding new records for indexing and updating the indexes do not require a synchronous wait for the client. Lily achieves asynchronous work execution by extensive use of message queue. All the messages are pushed to the message queue. Lily uses its own message queue for this.

Querying using Lily

While running a user query, the following steps are performed by the Lily server:

1. Lily Data Repository and Solr publishes available nodes to ZooKeeper
2. The client requests ZooKeeper for available nodes for performing query
3. ZooKeeper points to the available node for querying
4. The client connects to the requested node performs the query on the Solr node
5. Solr node processes and returns the matched indexes

Updating records using Lily

Each operation of updating the record in Lily performs the following steps:

1. Client requests ZooKeeper for available nodes
2. Client runs insert/update/delete on data by talking with Lily Data Repository
3. Lily first notifies the write-ahead log of its intentions
4. Lily updates its repository by making a change
5. Once done, Lily updates the write-ahead logger about completion
6. Lily inserts a message in the queue about requested operation
7. Indexer listeners on the queue request SolrCloud/Solr to update the index accordingly with the changes
8. Solr/SolrCloud runs the update on the distributed cluster

Lily indexer runs on the listening end of the Lily message queue. Based on the message, it requests the Solr instance to update the indexes. Indexer also stores the mapping between Lily records and Solr documents. Lily supports denormalization of records. With denormalization, records linked with each other can be brought together.

Installing and running Lily

To install Lily on your cluster, all the nodes participating in Lily should have:

- JDK 1.6 and above
- Operating system such as Linux, MacOS, and Unix flavors
- Apache/Cloudera Hadoop with HBase setup
- Apache/Cloudera Solr/SolrCloud/ZooKeeper ensemble

> When multiple ZooKeepers are running in high availability and fault tolerance model, they are called ZooKeeper ensemble.

Now, you can download Lily from its site. It comes in two flavors; namely Lily open source and Lily enterprise. Lily enterprise is not free; it offers additional features such as support for Hive, ETL connectors, and support from NGDATA.

First, you need to define the schema. While setting up HBase, you need to copy the following jars:

- `lib/org/lilyproject /lily-hbase-ext/lily-hbase-ext-VERSION.jar`
- `lib/org/lilyproject /lily-bytes/lily-bytes-VERSION.jar`
- `lib/org/lilyproject /lily-util/lily-util-VERSION.jar`
- `lib/org/lilyproject /lily-repository-api/lily-repository-api-VERSION.jar`
- `lib/org/lilyproject /lily-repository-id-impl/ lily-repository-id-impl-VERSION.jar`
- `lib/org/lilyproject /lily-hbaseindex-base/ lily-hbaseindex-base-VERSION.jar`
- `lib/com/gotometrics/orderly/orderly/VERSION/ orderly-VERSION.jar`
- `lib/org/lilyproject/lily-indexer-derefmap-indexfilter/VERSION/lilyindexer-derefmap-indexfilter-VESION.jar`

Also, remove the Avro jars from the `Hadoop` folder. Now, update the following configuration files with point to correct the instances of other subsystems:

- `conf/general/hbase.xml`
- `conf/general/mapreduce.xml`
- `conf/general/zookeeper.xml`
- `conf/repository/repository.xml`
- `conf/rpc/rpc.xml`

Now you can start the Lily server.

```
bin/lily-server
```

The next step is to create some fields and record types that you can do by calling the `lily-import` script in the `bin` folder of Lily. Once you do that, you need to add indexes to Lily so that it can be searched that can be achieved by running the `lily-add-index` command. This call takes a parameter, a configuration for the indexer, and Solr instances. Indexer configuration tells Lily about which data should be indexed, and it also tells mapping of the fields of this data with the Solr fields. You can then use Solr to query your databases.

Deep dive – shards and indexing data of Apache Solr

We have already understood what sharding is in *Chapter 3*, *Making Big Data Work for Hadoop and Solr*. As the data gets populated in Apache Solr, the size of the Solr index grows, given that each Solr index contains many files/documents/records, and it becomes large enough to fit on a single machine. Additionally, with the growth of the indexes, it is possible that the performance of search query can slow down. Single Solr machine also suffers from concurrency issues and low I/O support. This, in turn, demands distributing the index across multiple machines. Solr can run a distributed query across multiple machines aggregating the results into one.

With the release of Solr 4.1, lots of these things are automated. SolrCloud does index distribution to the appropriate shard; it also takes care of distributing search across multiple shards. Search is possible with near real time, after the document is committed. ZooKeeper provides load balancing and failover to the Solr cluster making the overall setup more robust. The index partitioning can be done in multiple ways in Apache Solr:

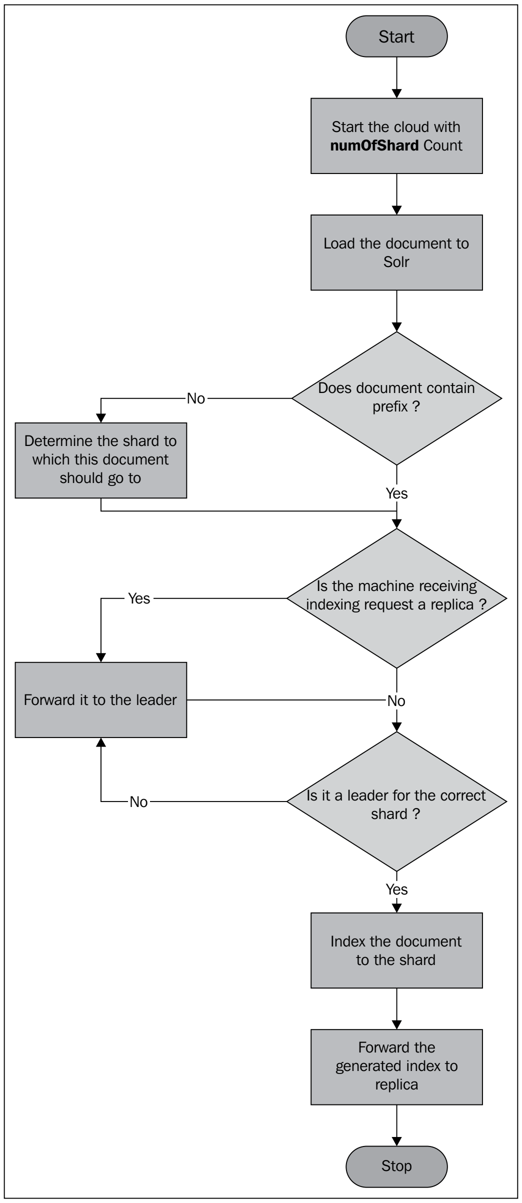
- **Simple partitioning**: It makes use of hashing function to a fixed number of shards
- **Prefix-based partitioning**: It is the partitioning based on the document ID, that is Red!12345 and White!22321. Red and White are the prefixes used for partitioning
- **Custom partitioning**: It is based on custom-defined partitioning such as document creation time

The sharding algorithm

In the new releases of SolrCloud, there are no masters and slaves. ZooKeeper holds the complete responsibility of choosing the leader as seen in *Chapter 3*, *Making Big Data Work for Hadoop and Solr*. Leaders are automatically elected using first come first served basis initially, and then later, all the nodes are assigned a sequence number when they are created. When a leader fails, the nodes in the cluster look for the next lowest sequence number. In a cluster, there are replicas or leaders for each of the shards.

When SolrCloud is started, you can start it with `numOfShards`, controlling how many shards to run in the cloud; you can choose `compositeId` while choosing a collection. When a Solr instance is started, it first registers itself with the ZooKeeper, creating **ephemeral node** or **znodes**. During the lifecycle, when a user sends his documents for indexing/sharding, he/she can specify a prefix for his/her document ID. This in turn directs Solr to perform hashing on the document to push it to an appropriate shard. It helps users in influencing the storage for their document indexes. Users can choose various strategies for distributing the index across multiple machines.

The following flow chart describes how sharding is performed on Apache Solr.


Start
Start the cloud with **numOfShard** Count
Load the document to Solr
Does document contain prefix ?
No
Determine the shard to which this document should go to
Yes
Is the machine receiving indexing request a replica ?
Yes
Forward it to the leader
No
Is it a leader for the correct shard ?
No
Yes
Index the document to the shard
Forward the generated index to replica
Stop

When not specified, Solr chooses the appropriate shard for indexing. If a leader goes down, the associated replica becomes the leader. When new nodes are added to the cloud, they are synced before they start participating in the cluster.

Adding a document to the distributed shard

To add a document in Solr, you can simply choose any node part of your cluster, and run the following command:

```
curl http://node1:8983/solr/update/json -H 'Content-
  type:application/json' -d
[
  {"id" : "1", "text" : "This is a test document"},
]'
```

When `node1` in the Solr cluster receives a request for indexing the document, if the document is a replica, it forwards it to the leader of the shard. Each leader performs hashing on the document ID, based on its prefix or automatically, and if the leader does not own the responsibility of that shard, it has to forward it to the leader of the shard. Once the correct leader receives the document, it updates its transactional log, and forwards the document to its replica for replication. While a document is received first, it is assigned a version ID; the leader first tries to see if it has a higher version. If it does, the leader will simply ignore the uploaded document.

> Solr transactional log is an append-only log of the write operations per node in a cluster. Solr records all the write operations before the write commits, and marks it post commit. If the indexing process is stopped for some reason, next time, Solr first reviews the transaction logs and then completes the pending indexing.

Configuring SolrCloud to work with large indexes

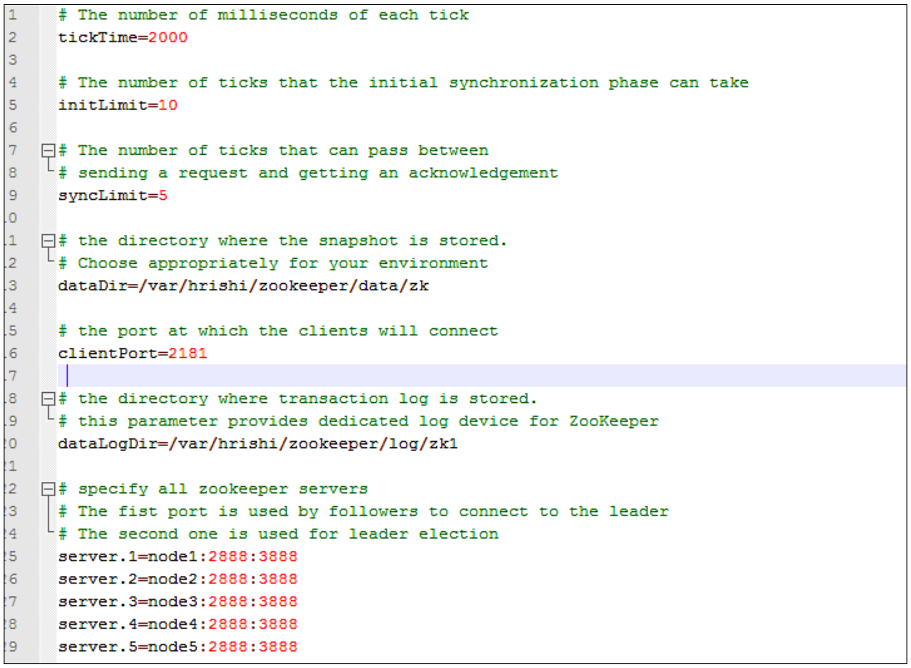
In order to configure SolrCloud to run with large indexes, it is important to first design the system based on the requirements. The design has to be based on the following factors:

- Number of nodes participating in the cloud
- Distribution of shards and their replicas over nodes
- Replication factors and leader
- ZooKeeper setup

Prerequisites for this would require Apache Solr, ZooKeeper, J2EE container (optional).

Setting up the ZooKeeper ensemble

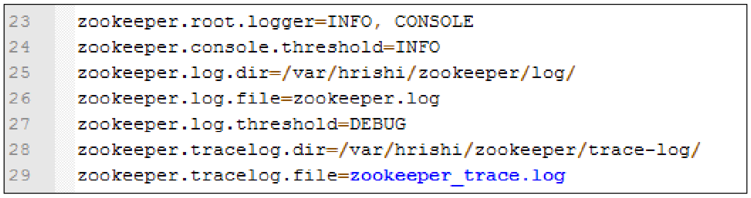
First, we need to set up a ZooKeeper ensemble on all the nodes. Although Apache Solr ships with embedded ZooKeeper, for large indexes and scalability requirements, it is recommended to go ahead with a full ZooKeeper set up. You can download the latest version of Apache ZooKeeper. Now, unzip the download on all the nodes, and edit the `zoo.cfg` file in your `ZKHOME | conf` folder; in that file, you need to specify the list of ZooKeeper servers as shown in the following screenshot. You must also specify correct `dataDir`, `clientPort`, and `dataLogDir`.



```
# The number of milliseconds of each tick
tickTime=2000

# The number of ticks that the initial synchronization phase can take
initLimit=10

# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5

# the directory where the snapshot is stored.
# Choose appropriately for your environment
dataDir=/var/hrishi/zookeeper/data/zk

# the port at which the clients will connect
clientPort=2181

# the directory where transaction log is stored.
# this parameter provides dedicated log device for ZooKeeper
dataLogDir=/var/hrishi/zookeeper/log/zk1

# specify all zookeeper servers
# The fist port is used by followers to connect to the leader
# The second one is used for leader election
server.1=node1:2888:3888
server.2=node2:2888:3888
server.3=node3:2888:3888
server.4=node4:2888:3888
server.5=node5:2888:3888
```

Here, `dataDir` is the folder where ZooKeeper will store data about cluster. `clientPort` is the port where Apache Solr will access the ZooKeeper instance.

You may also choose to configure logger for ZooKeeper as shown in the following screenshot. This will in turn help you to find out the issues quickly for the initial start.



```
zookeeper.root.logger=INFO, CONSOLE
zookeeper.console.threshold=INFO
zookeeper.log.dir=/var/hrishi/zookeeper/log/
zookeeper.log.file=zookeeper.log
zookeeper.log.threshold=DEBUG
zookeeper.tracelog.dir=/var/hrishi/zookeeper/trace-log/
zookeeper.tracelog.file=zookeeper_trace.log
```

Now, start ZooKeeper with the following command on all servers:

```
cd $ZKHOME/bin
./zkServer.sh start
```

You may also open additional windows to tail the `zookeeper.out` file.

Setting up the Apache Solr instance

First, start with the downloading of Apache Solr on your machine; you can use the CURL utility for that, and unzip the `solr` file. Next, you need to start editing the `solr.xml` file and the `solrconfig.xml` file, as described in *Chapter 3, Making Big Data Work for Hadoop and Solr*. Once the configuration is changed, you need to upload it to the ZooKeeper instance. We are going to use Apache ZooKeeper's command line interface to achieve that. This can be done by running the following command:

```
java -classpath .:/var/hrishi/zookeeper/zk-cli-library/*
  org.apache.solr.cloud.ZkCLI -cmd upconfig -zkhost node1:2181,node2:
     2181,node3: 2181,node4: 2181,node5: 2181-
      confdir .:/var/hrishi/node1/solr/collection1/conf/-confname
        clusterconf
```

Next step is to link the configuration with the collection that you will be using for your Solr index store. You can again do that using `zkCLI`.

```
java -classpath .:/var/hrishi/zookeeper/zk-cli-library/*
  org.apache.solr.cloud.ZkCLI -cmd linkconfig –collection
    clustercollection –confname clusterconf -zkhost node1:2181,node2:
      2181,node3: 2181,node4: 2181,node5: 2181
```

Once done, this will map your collection with the configuration through ZooKeeper. Next step is to deploy SolrCloud on the J2EE container, or you can choose to use in-built J2EE container that ships with Solr, that is Jetty. Now, you should run your container with Solr in it; while starting the container, you may even optimize the instance by running JVM in the server mode. Create `solr.xml` for your Solr home, and start all the containers. While running Apache Solr, you must point it to corrected ZooKeeper instance by specifying the `zkHost` property.

```
java -Djetty.port=8983 -DzkHost=node1:2181 -jar start.jar
```

Creating shards, collections, and replicas in SolrCloud

You can create shards, collections, and their replicas on SolrCloud through the web-based handlers provided by Solr by uploading those using the CURL utility. First, we need to start with the creation of collection (that is, `clusterCollection`) assuming the replication of `3`, and maximum shards per node of `2`.

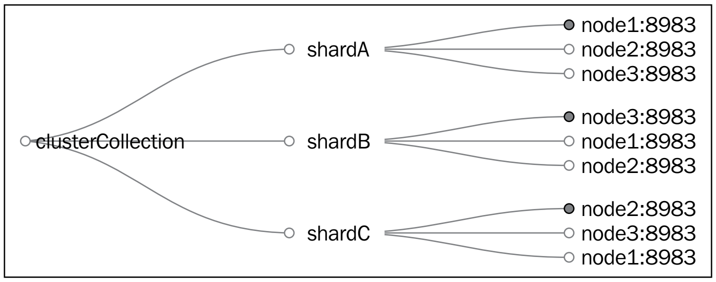
```
curl 'http://node1:8983/solr/admin/collections?
  action=CREATE&name=clusterCollection&numShards=3&
    replicationFactor=3&maxShardsPerNode=2'
```

This will create a collection with the name `clusterCollection` on Solr. We have already linked its configuration through ZooKeeper earlier.

Now, let's create replicas of the shards by running the following command; this command has to run for each replica you intend to create in your Solr instance.

```
curl 'http://node1:8983/solr/admin/cores?
  action=CREATE&name=shardA-Replica1&collection=clusterCollection
    &shard=shardA'

curl 'http://node2:8983/solr/admin/cores?
  action=CREATE&name=shardB-Replica2
    &collection= clusterCollection &shard=shardA'
```

The following diagram shows how the admin UI will show the shard distribution of your indexes:



Now the documents can directly be posted to any of the nodes hosting the Solr index; the following example shows uploading of default documents shipped with Solr on this cloud instance:

```
cd $SOLR_HOME/example/exampledocs/
java -Durl=http://node1:8983/solr/clusterCollection/update -jar
  post.jar ipod_video.xml
java -Durl=http://node2:8983/solr/clusterCollection/update -jar
  post.jar monitor.xml
```

You can simply verify it by accessing Solr instance with the wildcard query:

```
http://node1:8983/solr/clusterCollection/select?q=*:*
```

Summary

In this chapter, we have gone through the deep dive of Apache Solr with the Hadoop system. We also looked at Lily, which brings these two worlds together. We looked at the configuration of cloud instances for Big Data. We will now look at speeding up of your Big Data search by improving the performance of your setup in the next chapter.

5

Improving Performance of Search while Scaling with Big Data

As the data grows, it impacts your time taken for search, as well as to create new indexes along with the size of the repository. The simplest way to preserve the same performance of the search while scaling your data is to keep increasing your hardware, which includes higher processing power and higher memory size. This is not a cost-effective alternative. So, we look for optimizing the running of Big Data search instance. We have also seen different architectures of Solr in *Chapter 4*, *Using Big Data to Build Your Large Indexing*, among which the most suitable architecture can be chosen based on the requirements and the usage patterns.

Overall optimization of the technology stack which includes Apache Hadoop and Apache Solr helps you maintain more data with reasonable performance. The optimization is most important while scaling your instance for Big Data with Hadoop and Solr. We are going to look at different techniques of improving performances for your Big Data search. Optimization can be done at the different levels:

- Optimizing the search schema
- Optimizing the indexes
- Optimizing the J2EE container
- Optimizing the search runtime
- Monitoring your setup for performance and impact

Understanding the limits

Although you can have a completely distributed system for your Big Data search, there is a limit in terms of how far you can go. As you keep on distributing the shard, you may end up facing what is called "laggard problem" for indexes for your instance.

This problem states that the response to your search query, which is an aggregation of results from all the shards is controlled by the following formulae:

```
QueryResponse = avg(max(shardResponseTime))
```

This means, if you have many shards, the odds of having one of them responding slowly (due to some anomaly) to your queries will impact your query response time, and it will start increasing.

The distributed search in Apache Solr has many limitations. Each document uploaded on the distributed Big Data must have a unique key, and that unique key must be stored in the Solr repository. To do that, Solr `schema.xml` should have `stored=true` against the `key` attribute. This unique key has to be unique across all shards. Some of the features, such as More Like This, Join, and Query Elevation Component do not work in Solr distributed environment.

Running Solr in a distributed manner may lead to the issue of **distributed deadlock**. When a query is passed to a shard, it can make subqueries to all other shards. Now once the work is assigned, and the shards are busy serving their own request that depends upon completing other's request, it would have indefinite wait time for search query. Let's say there are two shards, and each of them got a job for processing; now they create subtasks which are then assigned to each other's threads. Both the requests are waiting for other shard to complete the task; thus, we have a distributed deadlock.

Apache Lucene has a cap on the size of index (approximately limiting it to 2 billion documents). However, theoretically, there is no limit to the number of documents that can be loaded on Big Data search indexing while running in a distributed mode.

Optimizing the search schema

When Solr is used in the context of a specific requirement; for example, a log search for an enterprise application, it holds a specific schema, which can be defined in `schema.xml` and copied over to nodes. The schema plays a vital role in the performance of your Solr instance, because based on the schema, attributes are indexed.

Specifying the default search field

In `schema.xml` of Solr configuration, the system allows you to specify the `<defaultSearchField>` parameter. This is the parameter that controls when you search without an explicit field name in your query, which field to pick up for searching. This is an optional parameter, if this is not specified, for all of the queries that are not providing the field name, search will run them on all of the available fields in the schema. This will not only consume more CPU time, but overall slow down the search performance.

Configuring search schema fields

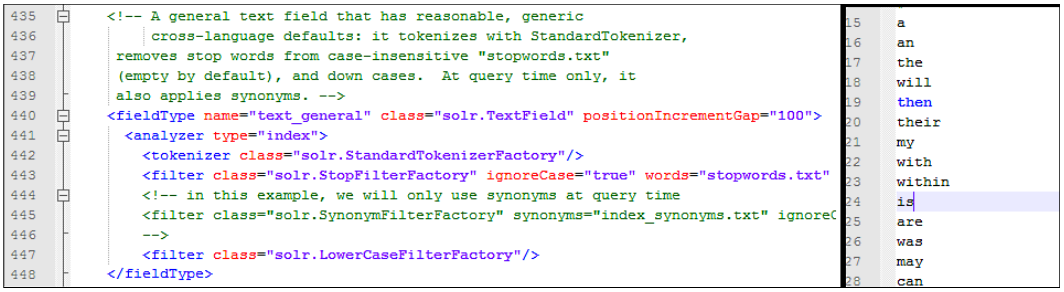
In custom schema, having more number of fields for indexing brings direct impact on the index size and the amount of memory needed to create your index and segments. You can control the amount of indexing of fields to be done on that by specifying `indexed=true` or `indexed=false` appropriately for each schema attribute. Avoid indexing unnecessary fields which you do not intend to use in search.

Similarly, you can set `stored=false` for those fields which are not returned as search results. Setting this function will not stop you querying for these fields, but you won't be able to retrieve the original value of these fields. For larger fields, there is significant value in terms of disk space and search speed for lookup.

The fields that are larger are difficult to fit in memory while indexing, so one has to ensure that all of the fields of the document fit in the memory. Each field can have `maxFieldLength` in the schema configuration; this in turn might help you control the sizing of the fields.

Stop words

We have already seen stop words in *Chapter 2*, *Understanding Solr*. *Appendix B*, *Creating Enterprise Search Using Apache Solr*, provides more details about them. They play a significant role in optimizing your Solr instance for performance. While performing the inverted index creation, the stop words are not considered by Solr because they do not add any value to your search. The stop words can be specified in any file and the file can be pointed out in `schema.xml` of the Solr configuration, as shown in the following screenshot:



```
<!-- A general text field that has reasonable, generic
     cross-language defaults: it tokenizes with StandardTokenizer,
 removes stop words from case-insensitive "stopwords.txt"
 (empty by default), and down cases.  At query time only, it
 also applies synonyms. -->
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt" ignoreC
    -->
    <filter class="solr.LowerCaseFilterFactory"/>
</fieldType>
```

```
a
an
the
will
then
their
my
with
within
is
are
was
may
can
```

Having a large set of stop words can significantly save space in terms of index size creation.

Stemming

Stemming is a process of reducing the derived word into its original form. By enabling word stemming with Apache Solr, it not only saves you search time, but also improves your query performance. Stemming also improves the accuracy of the result. For example, words such as walking, walked, and walks can be stemmed to walk. *Appendix B*, *Creating Enterprise Search Using Apache Solr* provides a detailed explanation about `protwords.txt`, which is used for stemming examples. Based on the requirements, a right stemming algorithm should be chosen for your instance. Here are some of the available algorithms for stemming:

Algorithm	Description
Porter	This rule-based algorithm transforms any form of the word into its stem. For example, the words talking and talked are marked as talk.
KStem	Similar to Porter, with less aggressiveness.
Snowball	This is all language supported string processing language for running your words. Using this, you can create new stemming algorithms.
Hunspell	Opens Office dictionary-based algorithm. Works with all languages, the only condition is the health of the dictionary.

Overall, the workflow and the mandatory fields mapping is shown in the following table. The true value indicates the presence of this attribute while defining the field. In *Chapter 2*, *Understanding Solr*, we have already explained the terms multi-valued, omit-norms, term vector, and so on.

Use case	Indexed	Stored	Multi-valued	Omit norms	Term vectors	Term Positions	Term offsets
search within field	TRUE						
retrieve contents		TRUE					
use as unique key	TRUE		FALSE				
sort on field	TRUE		FALSE	TRUE			
use field boosts				FALSE			
document boosts affect searches within field				FALSE			
highlighting	TRUE	TRUE					
Faceting	TRUE						
add multiple values, maintaining order			TRUE				
field length affects doc. score				FALSE			
MoreLikeThis		TRUE			TRUE		
term frequency					TRUE		
document frequency					TRUE		
tf*idf					TRUE		
term positions					TRUE	TRUE	TRUE
term offsets					TRUE	TRUE	TRUE

Index optimization

The indexes used in Apache Solr are inverted indexes. In the case of inverted indexing technique, all your text will be parsed and words will be extracted out of it. These words are then stored as index items, with the location of their appearance. For example, consider the following statements:

1. Mike enjoys playing on a beach
2. Playing on ground is a good exercise
3. Mike loves to exercise daily

The index with location information for all these sentences will look like the following (The numbers in brackets denote (sentence no, word no):

Mike (1,1), (3,1)

enjoys (1,2)

playing (1,3), (2,1)

on (1,4), (2,2)

a (1,5), (2,5)

beach (1,6)

ground (2,3)

is (2,4)

good (2,6)

loves (3,2)

to (3,3)

exercise (2,7), (3,4)

daily (3,5)

When you perform delete on your inverted index, it does not delete it, it only marks the document as deleted. It will get cleaned only when the segment, the index is part of are merged. When you create index, you should avoid modifying the index.

Limiting the indexing buffer size

As the index size grows, Solr instance starts hogging more CPU time and memory to perform faceted search. When the indexes are first created, the overall operation runs in a batch mode. All the documents are kept in memory, until it exceed RAM buffer size specified in `solr-config.xml`.

```
<ramBufferSizeMB>100</ramBufferSizeMB>
```

Once the size is exceeded, Solr creates a new segment or merges the index with the current segment. The default value of RAM buffer size is 100 megabits (Solr 1.4 and above). Similarly, there is another parameter that controls the maximum number of documents in buffer of Solr while indexing.

```
<maxBufferedDocs>1000</maxBufferedDocs>
```

If either of them, that is, maximum documents in buffer, and RAM size cross the predefined limit, then it will flush the changes. You can also control the maximum number of threads used for indexing the document by tuning the `maxIndexingThread`, the default value is `8`. By setting these parameters to optimal, as per your usage, you can speed up your indexing process. By setting this parameter, you can use clients which can connect concurrently to the search server for uploading the data using multiple threads. Solr provides the `ConcurrentUpdateSolrServer` class for the same.

The frequency of commit operation should also be controlled, as high frequency may end up eating more CPU time, and low frequency may increase the memory size of your instance.

When to commit changes?

Commit is the operation that ensures all the updates/uploads to Solr are stored on the disk. With Solr, you can perform commit in following different ways:

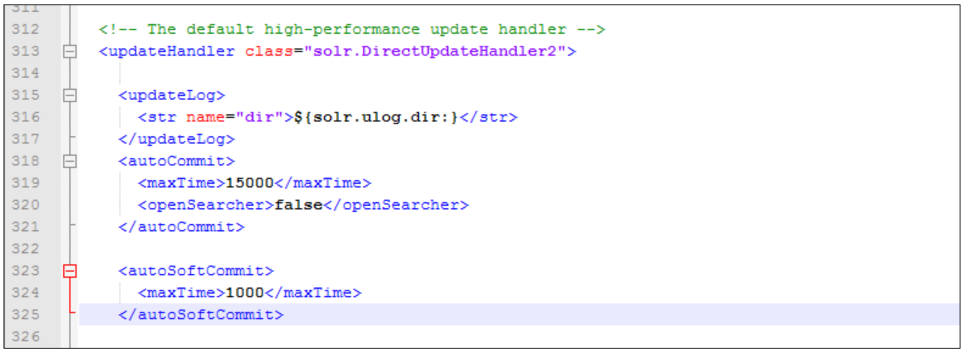
- Automatic commit
- Soft commit

When automatic commit is enabled, any document uploaded to Apache Solr gets written to the storage automatically by Solr based on certain conditions. In case of a cluster environment, a hard commit will replicate the indexes across all the nodes. This condition is maximum time (`maxTime`) or maximum documents (`maxDocs`) after which commit should take place. Choosing the value for these on the lower side works well for environment where you have continuous index updates; it incurs a significant performance bottleneck for batch updates in a distributed environment. At the same time, having the value of `maxTime` or `maxDocs` at highest side may pose a high risk of losing indexed documents in case of failure.

There is also an option called `openSearcher`; when `true`, it allows a new searcher to get an initialized post commit of changes, and enables the committed changes available for search immediately. Each handler also has an `updateLog`, which is a transaction log it enables recovery of updates in case of failure that is durability.

> To achieve a maximum durability of Solr instance, it is recommended to have hard commit size limit based on the log size of update log.

Similar to a hard commit, there is also a soft commit. A soft commit is a faster alternative, which, unlike hard commit, only makes the index changes visible for searches. It does not perform any sync of indexes across nodes. In case of power failure of machine, the changes made using soft commit are lost. With soft commit, Solr can achieve near real-time search capabilities. You should have soft commit `maxTime` less than hard commit time. The configuration file would look like the following:

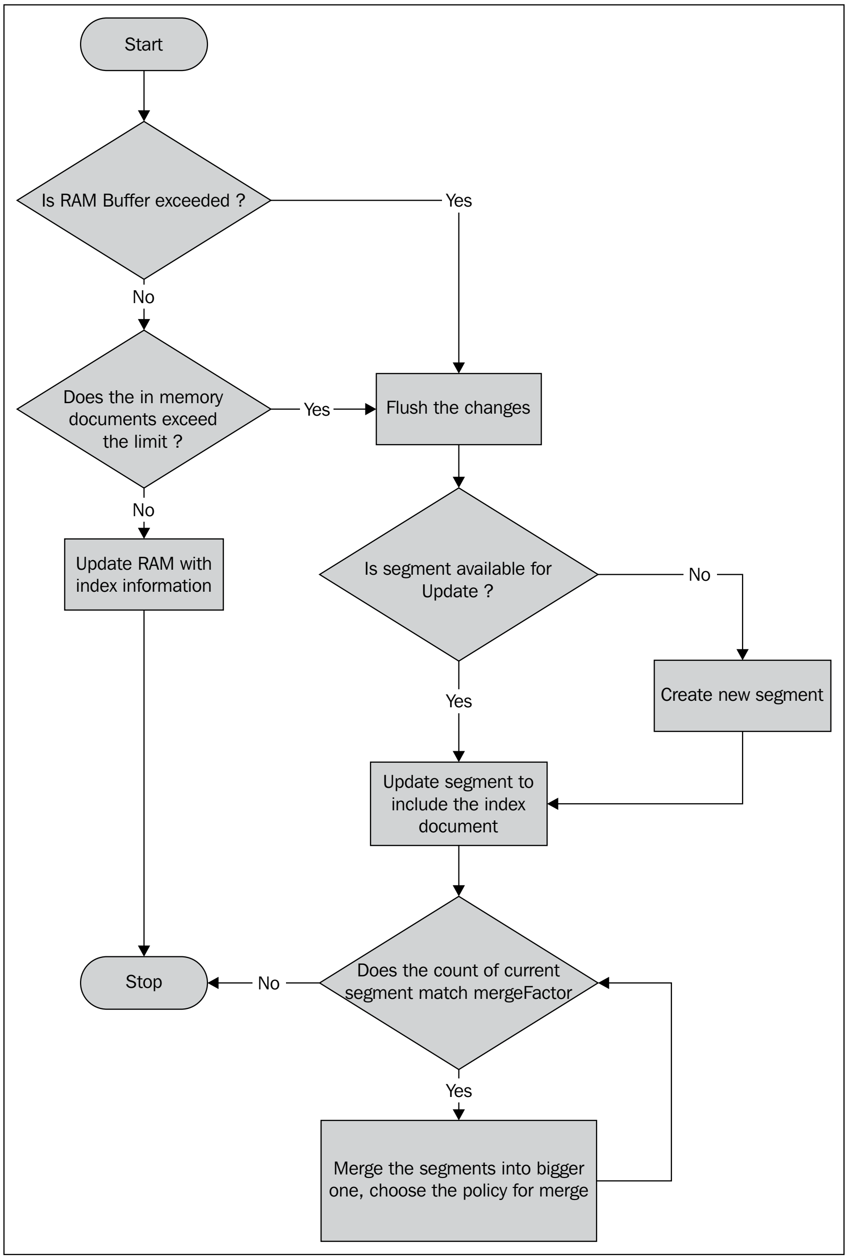


```
<!-- The default high-performance update handler -->
<updateHandler class="solr.DirectUpdateHandler2">

  <updateLog>
    <str name="dir">${solr.ulog.dir:}</str>
  </updateLog>
  <autoCommit>
    <maxTime>15000</maxTime>
    <openSearcher>false</openSearcher>
  </autoCommit>

  <autoSoftCommit>
    <maxTime>1000</maxTime>
  </autoSoftCommit>

```

Solr also allows you to pass the commit request in your update request itself.

Optimizing the index merge

While creating index segments, the following algorithm depicts how the Solr functions:


Start
Is RAM Buffer exceeded ?
Yes
No
Does the in memory documents exceed the limit ?
Yes
Flush the changes
No
Update RAM with index information
Is segment available for Update ?
No
Create new segment
Yes
Update segment to include the index document
Stop
No
Does the count of current segment match mergeFactor
Yes
Merge the segments into bigger one, choose the policy for merge

Solr keeps the newly updated index in the most recent segment; if the segment is filled up, it will create a new one. Solr performs the merging of segments as and when the number of lowest level segments touch mergeFactor specified in Solr configuration file. If so, it will merge all the segments into one. In the following case:

```
<mergeFactor>20</mergeFactor>
```

This is nothing but a scale; the segments are merged when the number of lowest level segments touches 20. This process keeps continuing. mergeFactor directly carries impact on your search query time and indexing time. If you have high mergeFactor, your index creation process is faster, as it does not really need to perform merging of index; however, for a search; Solr has to look into multiple files in file store. If you have low mergeFactor, it will slow down your indexing process due to the need to perform merge over huge indexes. The search will be relatively faster as it has to look at few files.

Optimize an option for index merging

When this option is called, Solr runs the index merge operation, and it forces the entire index segments to get merged into a single segment. This is an expensive operation, which in turn reads and rewrites all the indexes of Solr. It impacts the functioning of search instance, so it is recommended to run this operation when there is no/less load on the instance. It provides additional attributes such as waitFlush (blocks the instance until index changes are flushed to disk), waitSearcher (blocks until the new searcher with all the changes visible is made available), and maxSegment (you can choose to optimize your instance to maximum segment listed). Solr also allows you to call optimize through URL call itself:

```
curl
  'http://localhost:8983/solr/update?optimize=true&maxSegments=2&wai
    tFlush=false'
```

While running in a SolrCloud environment, you should be careful while running optimize (forced merge) on your own; instead, you can rely on Solr to perform an optimization and partial merge (that it does in the background).

Optimizing the container

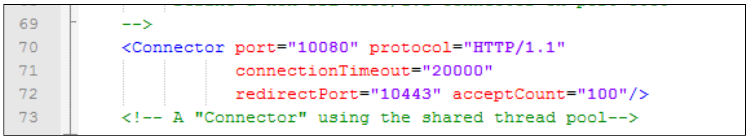
Most of the Big Data implementations including Solr and Hadoop run under J2EE container with some JDK. While scaling your instance for more data and more indexes, it becomes important to optimize your containers as well to ensure you get optimal high speed performance out of the system. Choosing the right JVM is one of the important factors.

There are many JVMs available in the market today which can be considered, such as Oracle Java HotSpot, BEA JRockit, Open Source JVM, and so on. Interestingly, Solr allows you to run multiple Solr instances on their own JVMs. Zing JVM from the Azul system is considered to be a high performance JVM for Solr/Lucene implementations.

Optimizing concurrent clients

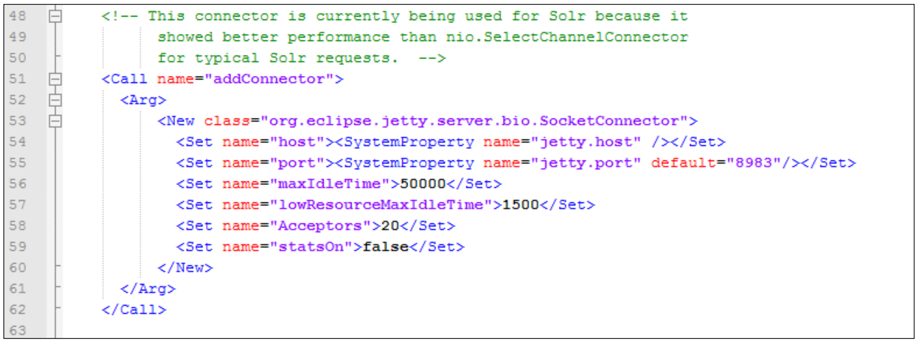
You can control the amount of concurrent connections that can be made to your container. This in turn reduces traffic on your instance which may be running in the standalone/distributed environment.

In Tomcat server, you can simply modify the following entries in `server.xml` for changing the number of concurrent connections:



```
    -->
    <Connector port="10080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="10443" acceptCount="100"/>
    <!-- A "Connector" using the shared thread pool-->
```

Similarly, in Jetty, you can control number of connections held by modifying `jetty.xml` in the following way:



```
    <!-- This connector is currently being used for Solr because it
          showed better performance than nio.SelectChannelConnector
          for typical Solr requests.  -->
    <Call name="addConnector">
      <Arg>
          <New class="org.eclipse.jetty.server.bio.SocketConnector">
            <Set name="host"><SystemProperty name="jetty.host" /></Set>
            <Set name="port"><SystemProperty name="jetty.port" default="8983"/></Set>
            <Set name="maxIdleTime">50000</Set>
            <Set name="lowResourceMaxIdleTime">1500</Set>
            <Set name="Acceptors">20</Set>
            <Set name="statsOn">false</Set>
          </New>
      </Arg>
    </Call>
```

Optimizing the Java virtual memory

One of the key optimization factors is controlling the virtual memory size of your Big Data Solr instance. This is applicable for instances running in distributed environment as well as the instances running as standalone search instance. As your Big Data search instance scales with data size, it requires more and more memory, and it becomes important to optimize the same. Apache Solr has built-in cache which is one of the factors considered for optimization. Since both Hadoop and Solr run on JVMs, one has to look at optimization of **Java Virtual Machine** (**JVM**).

All Solr instances run inside J2EE container. As an application, all the common optimizations for applications are applicable to it. It starts with choosing the right heap size for your JVM. Heap size for JVM can be controlled by the following parameters:

Parameter	Description
`-Xms`	The minimum heap size required with which the container is initialized
`-Xmx`	The maximum heap size up to which the container is allowed to grow

When you choose the minimum heap size to be low, the initialization of the application itself might take a longer time. Similarly, having a higher minimum heap size may unnecessarily block the huge memory segment which might be useful for your other processes. However, it will reduce the calls to resize the heap when heap is full, since the heap holds more memory at the start time. Similarly, having a low maximum heap size may fail your application running in between, throwing `Out Of Memory` exceptions for large indexes/objects of your search. When providing the memory size for the JVM, you need to ensure that you keep sufficient memory for your operating system and other processes to avoid them going into the thrashing mode.

> When you are running optimized Solr instances in a container, it is recommended not to install any other applications on the same container.

When heap is full, JVM tries to grab more memory based on the `-Xmx` parameter. Before doing that it performs garbage collection. Garbage collection in JVM is a process through which JVM reclaims the memory consumed by objects that is unused/expired/not referred by any of your application processes running in memory. Today's Java virtual machines trigger the garbage collection process automatically as and when needed. The process can explicitly be called from the application code through the `System.gc()` call, this will explicitly trigger the garbage collection process cleaning up the garbage. Such explicit calls to garbage collection should be avoided because:

- There is no control over when the garbage collection process is run while your search/indexing is run.
- When garbage collection process is running, it will end up taking your CPU and memory, which impacts the overall functioning of search.
- Heap size influences the time for running the garbage collection process. Longer heap size will take more time for garbage collector to identify and clean the VM objects. New releases of Java (1.7 and above) have done some optimization over the garbage collection.

If you are using Solr faceting, or features such as sorting, you will require more memory. An operating system performs memory swapping based on the need of processors. This can bring in huge latency in your search with large indexes. Many of the operating systems allow users to control the swapping of programs.

Optimization the search runtime

The search runtime speed is one of the primary concerns. It should be performed. You can also perform optimization at various levels at runtime. When Solr fetches results for the queries passed by the user, you can limit the fetching of results to a certain number by specifying the rows attribute in your search. The following query will return 10 rows of results from 10 to 20.

```
q=Scaling Big Data&rows=10&start=10
```

This can also be specified in `solrconfig.xml` as `queryResultWindowSize` by setting the size to a limited number of query results.

Let's look at various other optimizations possible in search runtime.

Optimizing through search queries

Whenever a query request is forwarded to a search instance, Solr can respond in various ways that is XML, JSON. A typical Solr response not only contains information about matched results, but also information about your facets, highlighted text, and many other things which are used by client (by default a velocity template based client provided by Solr). This in turn is a heavy response and it can be optimized by providing a compression over the result. Compressing the result, however, incurs more CPU time, and it may impact the response time and query performance. However, there is significant value in terms of response size that passes over the network.

Filter queries

A normal query on Solr will perform the search, and then it applies complex scoring mechanism to determine the relevance of the document with the search results. A filter query on Solr will perform the search and apply the filter, this does not apply to any scoring mechanism. A query can easily be converted into a filter query:

```
Normally: q=name:Scaling Hadoop AND type:books
Filter Query: q=name:Scaling Hadoop&fq=type:books
```

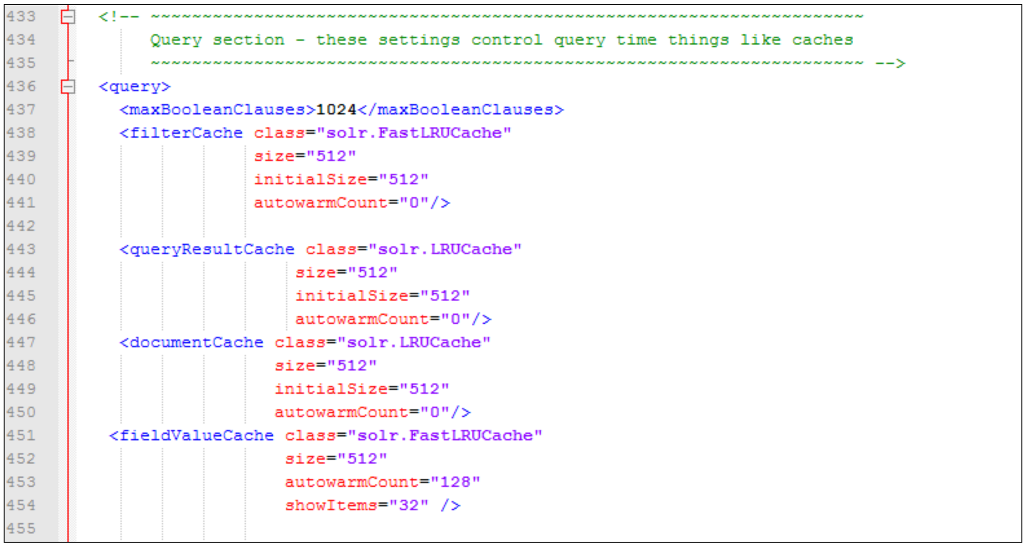
The processing required for scoring is not required; hence, it is faster compared to normal query. Since the scoring is no more applicable with filter queries, if the same query is passed again and again, the results are returned from filter cache directly.

Optimizing the Solr cache

Solr provides caching at various levels as a part of its optimization. For caching at these levels, there are multiple implementations available in Solr by default. `LRUCache`, least recently used (based on the synchronized `LinkedHashMap`), `FastLRUCache`, and `LFUCache`, least frequently used (based on the `ConcurrentHashMap`). Among these `FastLRUCache` is expected to be faster than all others. These caches are associated with search (index searchers).

> Cache autowarming is a feature by which a cache can pre-populate itself with objects from old search instances/cache.

These cache objects do not expire; they live till the time, and index searches are alive. The configuration for different cache can be specified in `solrconfig.xml`, as shown in the following screenshot:



```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     Query section - these settings control query time things like caches
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<query>
  <maxBooleanClauses>1024</maxBooleanClauses>
  <filterCache class="solr.FastLRUCache"
               size="512"
               initialSize="512"
               autowarmCount="0"/>

  <queryResultCache class="solr.LRUCache"
                   size="512"
                   initialSize="512"
                   autowarmCount="0"/>
  <documentCache class="solr.LRUCache"
                 size="512"
                 initialSize="512"
                 autowarmCount="0"/>
 <fieldValueCache class="solr.FastLRUCache"
                  size="512"
                  autowarmCount="128"
                  showItems="32" />
```

There are common parameters to the cache:

Parameter	**Description**
class	You can specify the type of cache you wish to attach that is `LRUCache`, `FastLRUCache`, or `LFUCache`
size	This is the maximum size a cache can reach to
initialSize	The initial size of the cache when it is initialized
autowarmCount	The number of entries to seed from old cache
minSize	Applicable for `FastLRUCache`, after cache reaches its peak size, it tries to reduce the cache size to minSize. The default value is 90 percent of size
acceptableSize	If `FastLRUCache` cannot reduce to minSize when cache reaches its peak, it will at least touch to acceptableSize

All cache are initialized when a new index searcher instance is opened. Let's look at different cache in Solr and how you can utilize them for speeding up your search.

The filter cache

This cache is responsible for storing the documents for filter queries that are passed to Solr. Each filter is cached separately; when queries are filtered, this cache returns the results and eventually based on the filtering criteria, the system performs intersection of them. If you have faceting, use of filter cache can improve performance. This cache stores the document IDs in an unordered state.

The query result cache

This cache will store the top N query results for each query passed by the user. It stores an ordered set of document IDs. For queries are that are repeated again and again, this cache is very effective. You can specify the maximum number of documents that can be cached by this cache in `solrconfig.xml`.

```
<queryResultMaxDocsCached>200</queryResultMaxDocsCached>
```

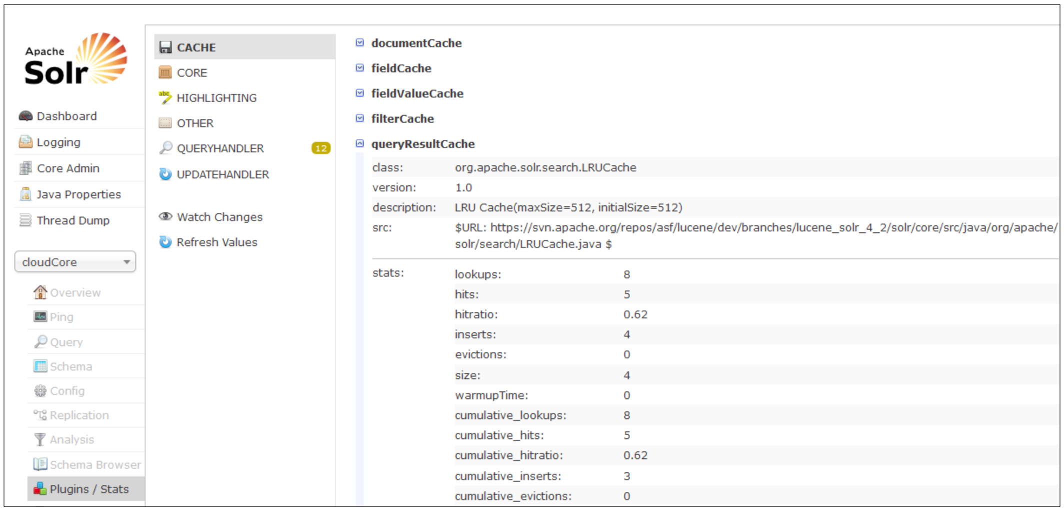
The document cache

This cache primarily stores the documents that are fetched from the disk. Once a document loads into a cache; next time, the search does not need to fetch it from the disk again reducing your overall disk IOs. This cache works on IDs of documents, so the autowarming feature does not really seem to have any impact, since the document IDs keep changing as and whenever there is a change in index.

> The size of the document cache should be based on your size of results and the size of max number of queries allowed to run, this will ensure that there is no refetch of documents by Solr.

The field value cache

This cache is used mainly for faceting. If you have a regular use of faceting, it makes sense to enable caching for field levels. This cache can also be used for sorting. It supports multi-valued fields. You can monitor the caching status in the administration of Solr. It provides information such as current load, hit rations, hits, and so on. This is shown in the following screenshot:



class:	org.apache.solr.search.LRUCache	
version:	1.0	
description:	LRU Cache(maxSize=512, initialSize=512)	
src:	$URL: https://svn.apache.org/repos/asf/lucene/dev/branches/lucene_solr_4_2/solr/core/src/java/org/apache/solr/search/LRUCache.java $	
stats:	lookups:	8
	hits:	5
	hitratio:	0.62
	inserts:	4
	evictions:	0
	size:	4
	warmupTime:	0
	cumulative_lookups:	8
	cumulative_hits:	5
	cumulative_hitratio:	0.62
	cumulative_inserts:	3
	cumulative_evictions:	0

Lazy field loading

By default, Solr reads all stored fields and then filters the ones which are not needed. This becomes a performance overhead for a large number of fields. When this flag is set, only fields that are requested will be loaded immediately, the rest of the fields are loaded lazily. This offers significant improvement over speed of search. This can be done by setting the following flag in `solconfig.xml`.

```
<enableLazyFieldLoading>true</enableLazyFieldLoading>
```

In addition to these options, you can also define your cache implementation.

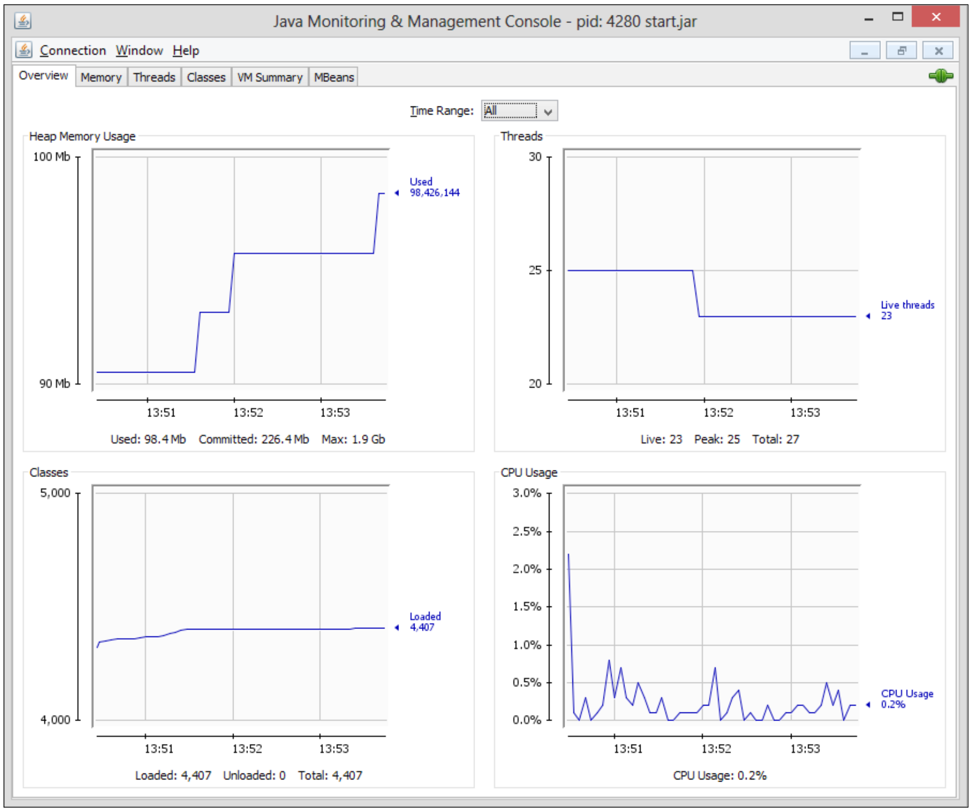
Optimizing search on Hadoop

When running Solr with Hadoop for indexing (Solr patches) or for search (Katta/Lily), the optimization of Hadoop adds performance benefits to Big Data search instance. The optimization can be done at the storage level that is HDFS as well as at the MapReduce programs.

While storing the indexes in a distributed environment such as Hadoop, storing in a compressed format can improve the storage space, as well as memory footprint. This storage in turn reduces your disk IO and bytes transferred over wires by adding an overhead for extracting it as and when needed. You can do that by enabling `mapred.compress.map.output=true`. Another interesting parameter is controlling the block size of a file for HDFS. This needs to be defined well, considering the fact that all indexes are stored in HDFS files, defining the appropriate block size (`dfs.block.size`) will help. The number of MapReduce tasks can also be optimized based on input size (the batch size of Solr documents for indexing/sharding). In case of `Solr-1301`, the output of reduce tasks are passed to `SolrOutputFormat`, which calls `SolrRecordWriter` for writing the data. After completing the reduce task, `SolrRecordWriter` calls `commit()` and `optimize()` for performing index merging.

Monitoring the Solr instance

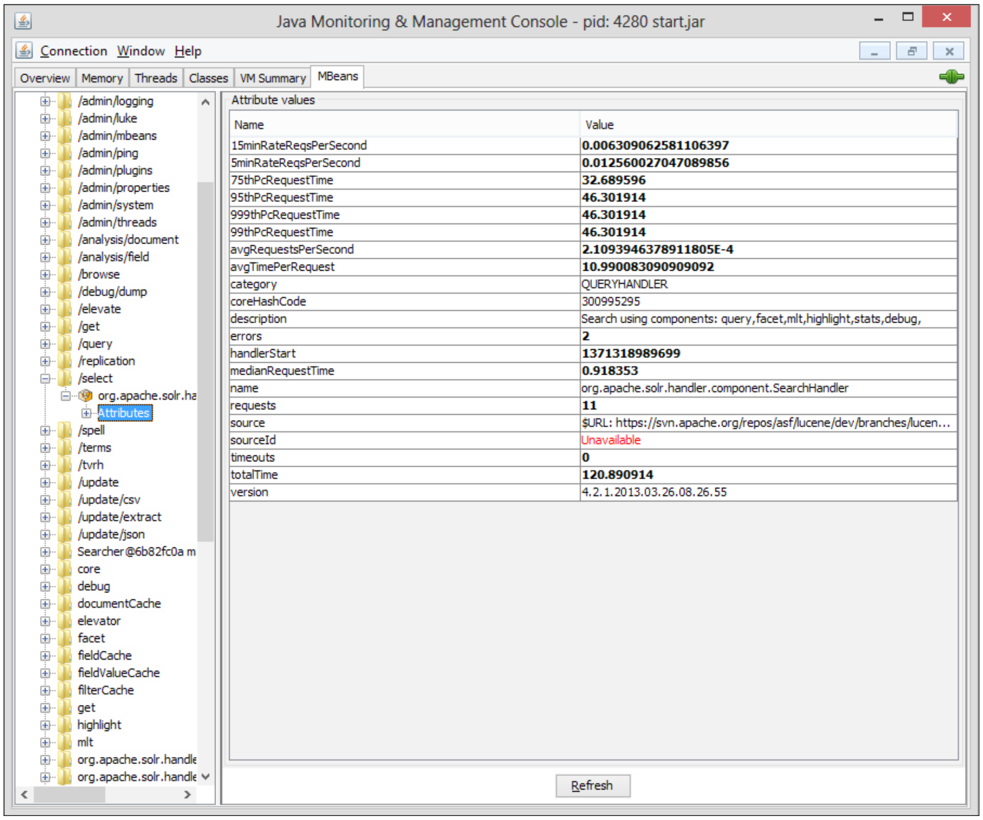
You can monitor the Solr instance for the purpose of memory and CPU usage. There are various ways of doing it; a simple administration of Solr provides you with some statistics for the usage. Using standard tools such as JConsole and JVisualVM, you can connect to the Solr process for monitoring of memory usage, threads, CPU usage, and so on, as shown in the following screenshot:



With JConsole, you can also look at different JMX-based MBeans supported by Solr. On a sample Jetty setup, you can simply connect Solr using the following procedure:

1. Open the JDK folder which is being used by Solr.
2. Go to the bin directory and run JConsole.
3. In JConsole, connect to Solr process; in the case of default Jetty implementation, connect to start.jar.
4. Once connected, switch to the **MBean** tab.

You will find the **MBean** browser as shown in the following screenshot:



Attribute values

Name	Value
15minRateReqsPerSecond	0.006309062581106397
5minRateReqsPerSecond	0.012560027047089856
75thPcRequestTime	32.689596
95thPcRequestTime	46.301914
999thPcRequestTime	46.301914
99thPcRequestTime	46.301914
avgRequestsPerSecond	2.1093946378911805E-4
avgTimePerRequest	10.990083090909092
category	QUERYHANDLER
coreHashCode	300995295
description	Search using components: query,facet,mlt,highlight,stats,debug,
errors	2
handlerStart	1371318989699
medianRequestTime	0.918353
name	org.apache.solr.handler.component.SearchHandler
requests	11
source	$URL: https://svn.apache.org/repos/asf/lucene/dev/branches/lucen...
sourceId	Unavailable
timeouts	0
totalTime	120.890914
version	4.2.1.2013.03.26.08.26.55

For a clustered search instance, you can connect remotely through JConsole. However, while starting JVM, you need to pass the following parameters to JVM (to bypass authentication and SSL):

```
-Dcom.sun.management.jmxremote.port=<port-no>
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

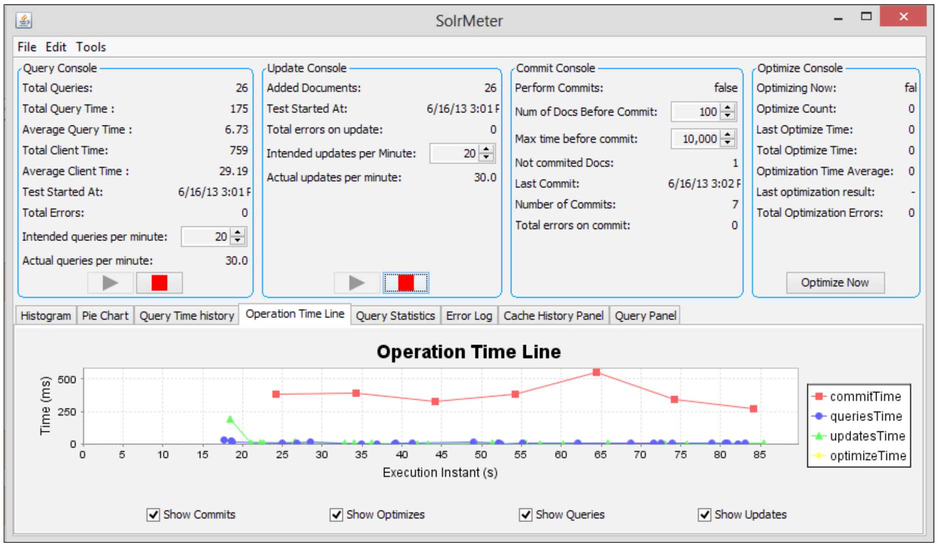
Using SolrMeter

SolrMeter is a tool that can be used by administrators to access the Solr instance running in a distributed environment for performing stress testing and get the search related statistics out of it. This tool can be downloaded from `http://code.google.com/p/solrmeter` and it can simply run by calling:

```
java -jar solrmeter-<version-no>.jar
```

This tool is one of the most powerful tools as it includes both loading and monitoring of your Big Data search instance. There are primarily four consoles, as shown in the following screenshot:

- **Query console**: This shows query related information such as time taken and queries ran
- **Update console**: This provides information regarding newly added documents, errors on updates, and so on
- **Commit console**: This provides commit history of documents, time taken, documents for pending commits, and so on
- **Optimize console**: This provides history for optimization, the count of optimize call run, average time taken, errors, and so on



Query Console

Total Queries: 26
Total Query Time : 175
Average Query Time : 6.73
Total Client Time: 759
Average Client Time : 29.19
Test Started At: 6/16/13 3:01 F
Total Errors: 0
Intended queries per minute: 20
Actual queries per minute: 30.0

Update Console

Added Documents: 26
Test Started At: 6/16/13 3:01 F
Total errors on update: 0
Intended updates per Minute: 20
Actual updates per minute: 30.0

Commit Console

Perform Commits: false
Num of Docs Before Commit: 100
Max time before commit: 10,000
Not commited Docs: 1
Last Commit: 6/16/13 3:02 F
Number of Commits: 7
Total errors on commit: 0

Optimize Console

Optimizing Now: fal
Optimize Count: 0
Last Optimize Time: 0
Total Optimize Time: 0
Optimization Time Average: 0
Last optimization result: -
Total Optimization Errors: 0

SolrMeter also displays performance measurements in a nice graphical manner, that is, histogram, pie chart, query time history, operation time line, query statistics, errors, and cache history. The charts together provide a detailed view on query performance. It also provides an option to optimize the indexes by providing an Optimize Now button.

Summary

In this chapter, we have gone through different ways of optimizing your Big Data instance to perform high speed data search and analysis. We also looked at how you can consistently keep track of performance of your system through various tools.

A
Use Cases for Big Data Search

Many organizations across the globe in different sectors have successfully adapted to the Apache Hadoop and Solr-based architectures, to provide a unique browsing and searching experience over their rapidly growing and diversified information. Let's look at some of the interesting use cases where Big Data search can be used.

E-commerce websites

E-commerce websites are meant to work for different types of users. These users visit the websites for multiple reasons:

- Visitors are looking for something specific, but they can't really describe what it is
- Visitors are looking for a specific product price/features
- Visitors come looking for good discounts, what's new, and so on
- Visitors wish to compare multiple products on cost/features/reviews

Most e-commerce websites are used to be built on custom developed pages running on a SQL database. Although a database provides excellent capabilities to manage your data structurally, it does not provide high speed searching and faceting like Solr. In addition to that, it becomes difficult to keep up with the queries for high performance. As the size of data grows, it hampers the overall speed and user experience.

Apache Solr in a distributed scenario provides excellent offerings in terms of browsing and searching experience. Solr can work easily, integrate with the database, and it can provide high speed search with real-time indexing. Advanced in-built features of Solr such as suggestions, a more like this search, and spelling checker can effectively help customer reach the merchandise he/she was looking for. The instance can easily be integrated with the current sites; faceting can provide interesting filters based on highest discount items, price range, type of merchandise, products from different companies, and so on, enabling a unique shopping experience for the end users. Many of the e-commerce based companies such as `buy.com`, `dollardays.com`, and `macys.com` have acquired distributed Solr-based solution over the traditional approach for providing customers with better browsing experience.

Log management for banking

Today banking software landscape scenario deals with many enterprise applications that play an important role in automating banking processes. Each of these applications talk with each other over wire. A typical enterprise architecture landscape consists of software for core banking application, CMS, credit card management, B2B portal, treasury management, HRMS, ERP, CRM, business warehouse, accounting, BI tools, analytics, custom applications, and various other enterprise applications fused together to ensure smooth business processes. The data center for such a complex landscape is usually placed across the globe in different countries with high performance servers having backup and replication. It, in turn, brings in a completely diversified set of software together in a secured environment.

Most of the banks today offer web-based interactions; they not only automate their own business processes, but also access various third party software of other banks and vendors. There is a dedicated team of administrators working 24 x 7 over monitoring and handling of issues/failures and escalations. A simple application of transferring money from your savings bank account to a loan account may touch upon at least twenty different applications. These systems generate terabytes of data everyday, which include transactional data, change logs, and so on.

The problem

The problem arises when any business workflow/transaction fails. With such a complex system, it becomes a big task for system administrators/managers to:

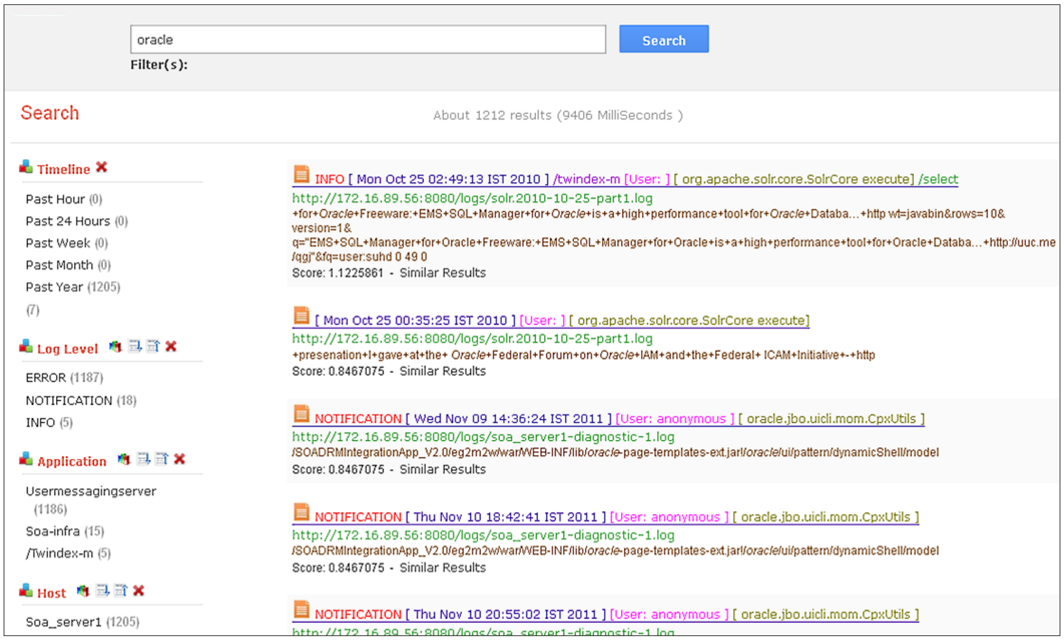
- Identify the error(s)/issue(s)
- Look for errors at different log files, and find out the real problem

- Find out the cause of failure
- Find correlation among the failures
- Monitor the workflow for occurrence of same issue

Product-based software provides nice user interface for administration, monitoring, and log management. However, most of the software including custom built applications and packaged software do not provide any such way. Eventually, the administrators have to get down to operating system file level and start looking for logs.

How can it be tackled?

This is the classic case where Big Data search (Apache Hadoop with Apache Solr) over a distributed environment can be used for effectively monitoring these applications. A sample user interface satisfying some of the expectations is shown in the following screenshot:



oracle

About 1212 results (9406 MilliSeconds)

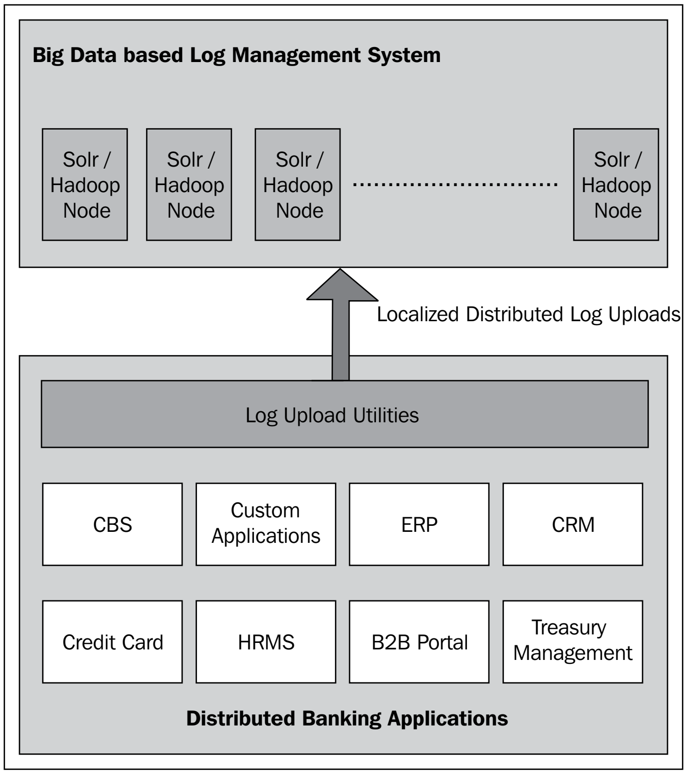
```
INFO [ Mon Oct 25 02:49:13 IST 2010 ] /twindex-m [User: ] [ org.apache.solr.core.SolrCore execute] /select
http://172.16.89.56:8080/logs/solr.2010-10-25-part1.log
+for+Oracle+Freeware:+EMS+SQL+Manager+for+Oracle+is+a+high+performance+tool+for+Oracle+Databa...+http wt=javabin&rows=10&
version=1&
q="EMS+SQL+Manager+for+Oracle+Freeware:+EMS+SQL+Manager+for+Oracle+is+a+high+performance+tool+for+Oracle+Databa...+http://uuc.me
/qgj"&fq=user:suhd 0 49 0
Score: 1.1225861 - Similar Results

[ Mon Oct 25 00:35:25 IST 2010 ] [User: ] [ org.apache.solr.core.SolrCore execute]
http://172.16.89.56:8080/logs/solr.2010-10-25-part1.log
+presenation+I+gave+at+the+ Oracle+Federal+Forum+on+Oracle+IAM+and+the+Federal+ ICAM+Initiative+-+http
Score: 0.8467075 - Similar Results

NOTIFICATION [ Wed Nov 09 14:36:24 IST 2011 ] [User: anonymous ] [ oracle.jbo.uicli.mom.CpxUtils ]
http://172.16.89.56:8080/logs/soa_server1-diagnostic-1.log
/SOADRMIntegrationApp_V2.0/eg2m2w/war/WEB-INF/lib/oracle-page-templates-ext.jar!/oracle/ui/pattern/dynamicShell/model
Score: 0.8467075 - Similar Results

NOTIFICATION [ Thu Nov 10 18:42:41 IST 2011 ] [User: anonymous ] [ oracle.jbo.uicli.mom.CpxUtils ]
http://172.16.89.56:8080/logs/soa_server1-diagnostic-1.log
/SOADRMIntegrationApp_V2.0/eg2m2w/war/WEB-INF/lib/oracle-page-templates-ext.jar!/oracle/ui/pattern/dynamicShell/model
Score: 0.8467075 - Similar Results

NOTIFICATION [ Thu Nov 10 20:55:02 IST 2011 ] [User: anonymous ] [ oracle.jbo.uicli.mom.CpxUtils ]
http://172.16.89.56:8080/logs/soa_server1-diagnostic-1.log
```

The following reasons enable us to qualify Apache Solr-based Big Data search as the solution:

- Apache Hadoop provides an environment for distributed storage and computing for banking global landscape. It also makes your log management scalable in terms of organization growth. This means even if the logs are lost due to rotational log management system from applications or cleaned automatically by your application server, they remain available in a distributed environment of Hadoop.
- Apache Solr supports storage of any type of schema making it work with different types of applications having different model layer, that is, application specific log files with their own proprietary schema for each application.
- Apache Solr provides efficient searching capabilities with highlighted text and snippets of matched results; this can be suitable while looking for the right set of issues and their occurrences in the past.
- Apache Solr provides rich browsing experience in terms of faceted search to drill down to the correct set of results one is looking for. In this case, administrators will be blessed with different types of facets such as timeline-based, application-scoped, based on error types, and severity.
- Apache Solr's near real-time search capabilities add value in terms of monitoring and hunt for new logs. One can develop custom utilities that can alert the administrator in case he/she receives a log with high severity. Overall the system gets proactive instead of being reactive.
- The overall cost of building this system is less, as none of these technologies require high-end servers, and they are open source.

High-level design

The overall design, as shown in the following diagram, can have a schema containing common attributes across all the log files such as date and time of log, severity, application name, user name, type of log, and so on. Other attributes can be added as dynamic text fields.



Since each system has different log schema, these logs have to be parsed periodically, and then uploaded to the distributed search. For that, we can either write down the utilities which will understand the schema, and extract the field data from logs. These utilities can feed the outcome to distributed search nodes which are nothing but the Solr instances running on a distributed system like Hadoop. To achieve near real-time search, the Solr configuration requires a change accordingly.

B
Creating Enterprise Search Using Apache Solr

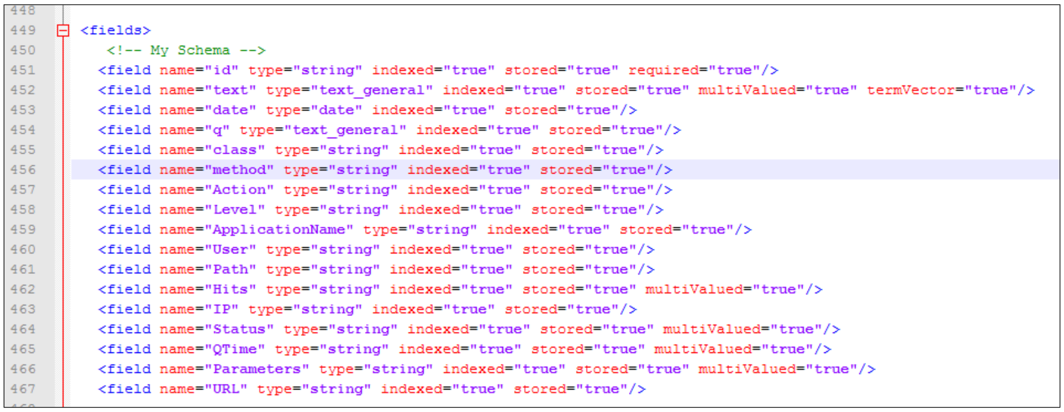
Let's look at some of the real configuration files. We are only going to look at the additions or changes to these files.

schema.xml

Broadly `schema.xml` contains following information:

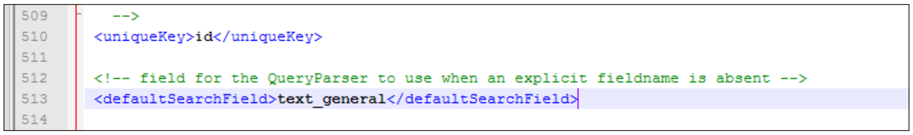
- Different types of field names of schema and data types (`<fields>`…`<field>`)
- Definition of user/seeded defined data types (`<types>`…`<fieldTypes>`)
- Dynamic fields (`<fields>`….`<dynamicField>`)
- Information about `uniqueKey` to define each document uniquely (`<uniqueKey>`)
- Information regarding `QueryParser` for Solr (`<solrQueryParser>`)
- Default search field is used when the user does not pass the field name (`<defaultSearchField>`)
- Information about copying a field from one to another (`<copyField>`)

In *Chapter 2*, *Understanding Solr*, we have already explained important attributes of the `schema.xml` file. Here is a sample `schema.xml` file in which the fields will look like the following screenshot:



```
<fields>
   <!-- My Schema -->
  <field name="id" type="string" indexed="true" stored="true" required="true"/>
  <field name="text" type="text_general" indexed="true" stored="true" multiValued="true" termVector="true"/>
  <field name="date" type="date" indexed="true" stored="true"/>
  <field name="q" type="text_general" indexed="true" stored="true"/>
  <field name="class" type="string" indexed="true" stored="true"/>
  <field name="method" type="string" indexed="true" stored="true"/>
  <field name="Action" type="string" indexed="true" stored="true"/>
  <field name="Level" type="string" indexed="true" stored="true"/>
  <field name="ApplicationName" type="string" indexed="true" stored="true"/>
  <field name="User" type="string" indexed="true" stored="true"/>
  <field name="Path" type="string" indexed="true" stored="true"/>
  <field name="Hits" type="string" indexed="true" stored="true" multiValued="true"/>
  <field name="IP" type="string" indexed="true" stored="true"/>
  <field name="Status" type="string" indexed="true" stored="true" multiValued="true"/>
  <field name="QTime" type="string" indexed="true" stored="true" multiValued="true"/>
  <field name="Parameters" type="string" indexed="true" stored="true" multiValued="true"/>
  <field name="URL" type="string" indexed="true" stored="true"/>
```

Remove all the copy fields, if not needed. The `uniqueKey` field is used to determine each document uniquely and will be required unless it is marked as `required=false`. The default search field provides a field name that Solr will use for searching when the user does not specify any field. Specify unique key and default search as shown in the following screenshot:



```
   -->
 <uniqueKey>id</uniqueKey>

 <!-- field for the QueryParser to use when an explicit fieldname is absent -->
 <defaultSearchField>text_general</defaultSearchField>
```

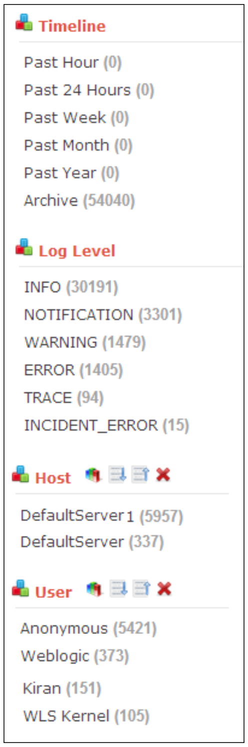
solrconfig.xml

Chapter 2, *Understanding Solr*, of this book explains the `solrconfig.xml` file in detail. We will look at the sample configuration in this section for log management. In the Solr configuration, interesting part will be the introduction of facets. For log management, you may consider the following facets to make overall browsing interesting:

Facet	Description
Timeline based	With this facet, users will be able to effectively filter their search based on the time. For example, options such as past 1 hour, past 1 week, and so on.
Levels of log	Levels of log provide you with the severity: for example, SEVERE, ERROR, INFO, and so on.

Facet	Description
Host	Since this system provides a common search for multiple machines, this facet can provide filtering criteria if an administrator is looking for something specific
User	If an administrator knows about the user, extracting user information from log can add better filtering through the user facet
Application	Similar to host, administrators can filter the logs based on an application using this facet
Severity	Severity can be another filtering criteria; most severe errors can be filtered with this facet

In addition to this, you will also use features of highlighting logs, spelling correction, suggestions (`MoreLikeThis`), and so on. The following screenshot shows a sample facet sidebar of Apache Solr to give us a better understanding over how it may look:



Timeline

Past Hour (0)
Past 24 Hours (0)
Past Week (0)
Past Month (0)
Past Year (0)
Archive (54040)

Log Level

INFO (30191)
NOTIFICATION (3301)
WARNING (1479)
ERROR (1405)
TRACE (94)
INCIDENT_ERROR (15)

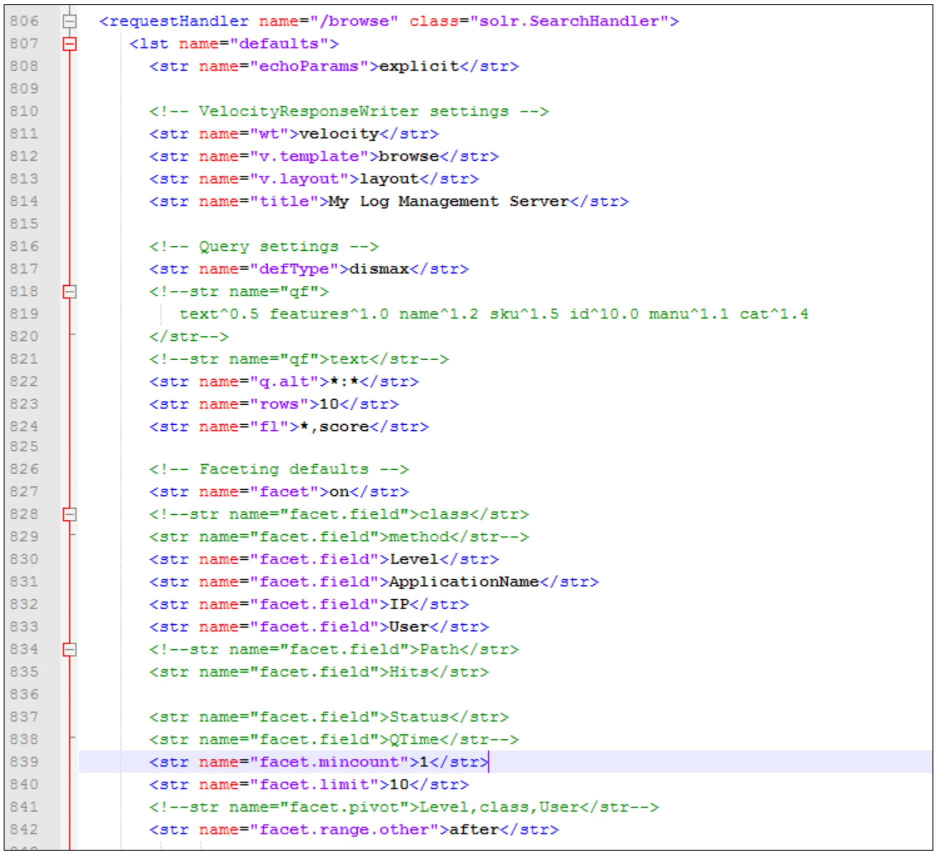
Host

DefaultServer1 (5957)
DefaultServer (337)

User

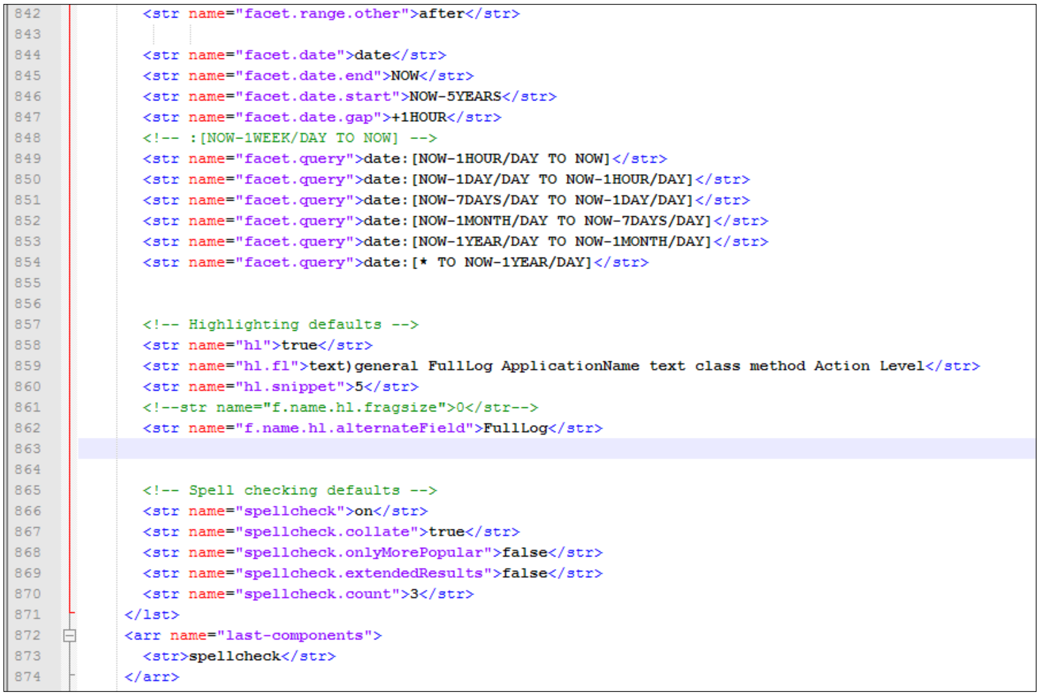
Anonymous (5421)
Weblogic (373)
Kiran (151)
WLS Kernel (105)

The following sample configuration for Solr shows different facets and other information when you access/browse:



```
<requestHandler name="/browse" class="solr.SearchHandler">
   <lst name="defaults">
     <str name="echoParams">explicit</str>

     <!-- VelocityResponseWriter settings -->
     <str name="wt">velocity</str>
     <str name="v.template">browse</str>
     <str name="v.layout">layout</str>
     <str name="title">My Log Management Server</str>

     <!-- Query settings -->
     <str name="defType">dismax</str>
     <!--str name="qf">
        text^0.5 features^1.0 name^1.2 sku^1.5 id^10.0 manu^1.1 cat^1.4
     </str-->
     <!--str name="qf">text</str-->
     <str name="q.alt">*:*</str>
     <str name="rows">10</str>
     <str name="fl">*,score</str>

     <!-- Faceting defaults -->
     <str name="facet">on</str>
     <!--str name="facet.field">class</str>
     <str name="facet.field">method</str-->
     <str name="facet.field">Level</str>
     <str name="facet.field">ApplicationName</str>
     <str name="facet.field">IP</str>
     <str name="facet.field">User</str>
     <!--str name="facet.field">Path</str>
     <str name="facet.field">Hits</str>

     <str name="facet.field">Status</str>
     <str name="facet.field">QTime</str-->
     <str name="facet.mincount">1</str>
     <str name="facet.limit">10</str>
     <!--str name="facet.pivot">Level,class,User</str-->
     <str name="facet.range.other">after</str>
```

Similarly, the following configuration shows a timeline-based facet, and features such as highlighting and spell check:



```
    <str name="facet.range.other">after</str>

    <str name="facet.date">date</str>
    <str name="facet.date.end">NOW</str>
    <str name="facet.date.start">NOW-5YEARS</str>
    <str name="facet.date.gap">+1HOUR</str>
    <!-- :[NOW-1WEEK/DAY TO NOW] -->
    <str name="facet.query">date:[NOW-1HOUR/DAY TO NOW]</str>
    <str name="facet.query">date:[NOW-1DAY/DAY TO NOW-1HOUR/DAY]</str>
    <str name="facet.query">date:[NOW-7DAYS/DAY TO NOW-1DAY/DAY]</str>
    <str name="facet.query">date:[NOW-1MONTH/DAY TO NOW-7DAYS/DAY]</str>
    <str name="facet.query">date:[NOW-1YEAR/DAY TO NOW-1MONTH/DAY]</str>
    <str name="facet.query">date:[* TO NOW-1YEAR/DAY]</str>


    <!-- Highlighting defaults -->
    <str name="hl">true</str>
    <str name="hl.fl">text)general FullLog ApplicationName text class method Action Level</str>
    <str name="hl.snippet">5</str>
    <!--str name="f.name.hl.fragsize">0</str-->
    <str name="f.name.hl.alternateField">FullLog</str>


    <!-- Spell checking defaults -->
    <str name="spellcheck">on</str>
    <str name="spellcheck.collate">true</str>
    <str name="spellcheck.onlyMorePopular">false</str>
    <str name="spellcheck.extendedResults">false</str>
    <str name="spellcheck.count">3</str>
  </lst>
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
```

spellings.txt

The `spellings.txt` file provides file-based spellcheck and it can be enabled by specifying the following code in `solrconfig.xml`:

```
<searchComponent name="spellcheck"
  class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="classname">solr.FileBasedSpellChecker</str>
    <str name="name">file</str>
    <str name="sourceLocation">spellings.txt</str>
    <str name="characterEncoding">UTF-8</str>
    <str name="spellcheckIndexDir">./spellcheckerFile</str>
  </lst>
</searchComponent>
```

In this file, you can write a list of correct words. This file is used to define a dictionary for the users. You need to enter each word in a new line shown as follows:

- `solr`
- `solar`

Once the dictionary is created, it needs to be built by calling `spellcheck.build` through the URL:

```
http://<solr-url>/select?q=*:*&spellcheck=true&spellcheck.build=true
```

Now, you can simply check the spellchecker by calling the following URL:

```
http://<solr-url>/select?q=solar&spellcheck=true
```

synonyms.txt

The `synonyms.txt` file is used by synonym filter to replace the tokens with their synonyms. For example, a search for DVD may expand to DVD, DVDs, and Digital Versatile Disk depending on your mapping in this file. Here is how you can specify synonyms:

- `GB, gib, gigabyte, gigabytes`
- `MB, mib, megabyte, megabytes`
- `Television, Televisions, TV, TVs`
- `Incident_error => error`

In this file, you can also do spelling corrections; the following example shows how it can be done:

- `assasination => assassination`

protwords.txt

You can protect the words that you do not want to be stemmed. For example, a stemming will cut a word "manager" to manage. If you do not wish to protect them, you can specify those words in this file line-by-line:

- `manager`
- `Exception`
- `Accounting`

stopwords.txt

Using the `stopwords.txt` file, you can avoid the common words of your language, which do not add a significant value to your search. For example, a, an, the, you, I, am, and so on. You can specify them in this file line-by-line.

- `a`
- `an`
- …

C

Sample MapReduce Programs to Build the Solr Indexes

In this appendix, we are going to look at sample MapReduce programs to build Solr indexes. We will start with an example of a MapReduce program.

Let's say we have three files containing the following text, and we have to get a word count of each word:

- [I enjoy walking on the beach sand. The Maya beach is what I enjoy most.]
- [John loves to play volleyball on the beach.]
- [We enjoy watching television.]

The results are then split into blocks and replicated on multiple data nodes. The map function then extracts a count of words from each file. The following <key, value> pairs are outcomes of the map function of Hadoop:

- <I ,2> <enjoy, 2> <walking,1> <on,1> <the,2> <beach,2> <sand,1> <maya,1> <is,1> <what,1> <most,1>
- <John,1> <loves,1> <to,1> <play,1> <volleyball,1> <on,1> <the,1> <beach,1>
- <we,1> <enjoy,1> <watching,1> <television,1>

Now, reduce task merges all these together and reduces the input to a single set of <key, value> pairs, getting us the count of words:

<I ,2> <enjoy, 3> <walking,1> <on,2> <the,3> <beach,3> <sand,1> <maya,1> <is,1> <what,1> <most,1> <John,1> <loves,1> <to,1> <play,1> <volleyball,1> <we,1> <watching,1> <television,1>

Now, we will look at some samples for different implementations.

The Solr-1045 patch – map program

The following sample program will work with the Hadoop Version 0.20:

```
SolrConfig solrConfig = new SolrConfig();
  Configuration conf = getJobConfiguration();
  FileSystem fs = FileSystem.get(conf);

  if (fs.exists(outputPath))
    fs.delete(outputPath, true);
  if (fs.exists(indexPath))
    fs.delete(indexPath, true);

  for (int noShards = 0; noShards < noOfServer; noShards++)
  {
    //Set initial parameters
    IndexUpdateConfiguration iconf = new
      IndexUpdateConfiguration(conf);
    iconf.setIndexInputFormatClass(SolrXMLDocInputFormat.class);
    iconf.setLocalAnalysisClass(SolrLocalAnalysis.class);
    //configure the indexing for SOlr
    SolrIndexConfig solrIndexConf = solrConfig.mainIndexConfig;
    if (solrIndexConf.maxFieldLength != -1)
      iconf.setIndexMaxFieldLength(solrIndexConf.maxFieldLength);
    iconf.setIndexUseCompoundFile(solrIndexConf.useCompoundFile);
    iconf.setIndexMaxNumSegments(maxSegments);
    //initialize array
    Shard[] shards = new Shard[numShards];
    for (int j = 0; j < shards.length; j++)
    {
      Path path = new Path(indexPath, NUMBER_FORMAT.format(j));
      shards[j] = new Shard(versionNumber, path.toString(),
        generation);
    }
    //An implementation of an index updater interface which
      creates a Map/Reduce job configuration and run the
    //Map/Reduce job to analyze documents and update Lucene
      instances in parallel.
    IIndexUpdater updater = new SolrIndexUpdater();
    updater.run(conf, new Path[]
    { inputPath }, outputPath, numMapTasks, shards);
```

The Solr-1301 patch – reduce-side indexing

The patch provides `RecordWriter` to generate Solr index. It also provides `OutputFormat` for outputting your indexes. With Solr-1301 patch, we only need to implement the reducer since this patch is based on reducer.

You can follow the given steps to achieve reduce-side indexing using Solr-1301:

1. Get `solrconfig.xml`, `schema.xml` and other configurations in the `conf` folder, and also get all the Solr libraries in the `lib` folder.
2. Implement `SolrDocumentConverter` that takes the <key, value> pair and returns `SolrInputDocument`. This converts output records to Solr documents.

```
public class HadoopDocumentConverter extends
  SolrDocumentConverter<Text, Text> {
  @Override
  public Collection<SolrInputDocument> convert(Text key,
    Text value) {
    ArrayList<SolrInputDocument> list = new
      ArrayList<SolrInputDocument>();
    SolrInputDocument document = new SolrInputDocument();
    document.addField("key", key);
    document.addField("value", value);
    list.add(document);
    return list;
  }
}
```

3. Create a simple reducer as follows:

```
public static class IndexReducer {
  protected void setup(Context context) throws IOException,
    InterruptedException {
    super.setup(context);
    SolrRecordWriter.addReducerContext(context);
  }
}
```

4. Now configure the Hadoop reducer and configure the job. Depending upon the batch configuration (that is, `solr.record.writer.batch.size`), the documents are buffered before updating the index.

```
SolrDocumentConverter.setSolrDocumentConverter(
  HadoopDocumentConverter.class, job.getConfiguration());
job.setReducerClass(SolrBatchIndexerReducer.class);
job.setOutputFormatClass(SolrOutputFormat.class);
File solrHome = new File("/user/hrishikes/solr");
SolrOutputFormat.setupSolrHomeCache(solrHome,
  job.getConfiguration());
```

The `solrHome` is the patch where `solr.zip` is stored. Each task initiates the `EmbeddedServer` instance for performing the task.

Katta

Let's look at the sample indexer code that creates indexes for Katta:

```
public class KattaIndexer implements MapRunnable<LongWritable,
  Text, Text, Text> {
  private JobConf _conf;
  public void configure(JobConf conf) {
    _conf = conf;
  }

  public void run(RecordReader<LongWritable, Text> reader,
    OutputCollector<Text, Text> output, final Reporter report)
    throws IOException {
      LongWritable key = reader.createKey();
      Text value = reader.createValue();
      String tmp = _conf.get("hadoop.tmp.dir");
      long millis = System.currentTimeMillis();
      String shardName = "" + millis + "-" + new
        Random().nextInt();
      File file = new File(tmp, shardName);
      report.progress();
      Analyzer analyzer = IndexConfiguration.getAnalyzer(_conf);
      IndexWriter indexWriter = new IndexWriter(file, analyzer);
      indexWriter.setMergeFactor(100000);
      report.setStatus("Adding documents...");
      while (reader.next(key, value)) {
        report.progress();
        Document doc = new Document();
        String text = "" + value.toString();
        Field contentField = new Field("content", text,
          Store.YES, Index.TOKENIZED);
        doc.add(contentField);
        indexWriter.addDocument(doc);
      }
      report.setStatus("Done adding documents.");
      Thread t = new Thread() {
        public boolean stop = false;
        @Override
        public void run() {
          while (!stop) {
            // Makes sure hadoop is not killing the task in case
            the
            // optimization
            // takes longer than the task timeout.
            report.progress();
            try {
              sleep(10000);
            } catch (InterruptedException e) {
              // don't need to do anything.
              stop = true;
            }
          }
        }
      };
      t.start();
      report.setStatus("Optimizing index...");
      indexWriter.optimize();
      report.setStatus("Done optimizing!");
      report.setStatus("Closing index...");
      indexWriter.close();
      report.setStatus("Closing done!");
      FileSystem fileSystem = FileSystem.get(_conf);

      report.setStatus("Starting copy to final destination...");
      Path destination = new Path
        (_conf.get("finalDestination"));
      fileSystem.copyFromLocalFile(new
        Path(file.getAbsolutePath()), destination);
      report.setStatus("Copy to final destination done!");
      report.setStatus("Deleting tmp files...");
      FileUtil.fullyDelete(file);
      report.setStatus("Delteing tmp files done!");
      t.interrupt();
    }
}
```

Here is a sample Hadoop job that creates the Katta instance:

```
KattaIndexer kattaIndexer = new KattaIndexer();
String input = <input>;
String output = <output>;
int numOfShards = Integer.parseInt(args[2]);
kattaIndexer.startIndexer(input, output, numOfShards);
```

You can use the following search client to search on the Katta instance:

```
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_CURRENT);
  Query query = new QueryParser(Version.LUCENE_CURRENT, args[1],
  analyzer).parse(args[2]);

  ZkConfiguration conf = new ZkConfiguration();
  LuceneClient luceneClient = new LuceneClient(conf);
  Hits hits = luceneClient.search(query,
    Arrays.asList(args[0]).toArray(new String[1]), 99);

  int num = 0;
  for (Hit hit : hits.getHits()) {
    MapWritable mw = luceneClient.getDetails(hit);
    for (Map.Entry<Writable, Writable> entry : mw.entrySet()) {
      System.out.println("[" + (num++) + "] key -> " +
        entry.getKey() + ", value -> " + entry.getValue());
    }
  }
```

Index

Thank you for buying

Scaling Big Data with Hadoop and Solr

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

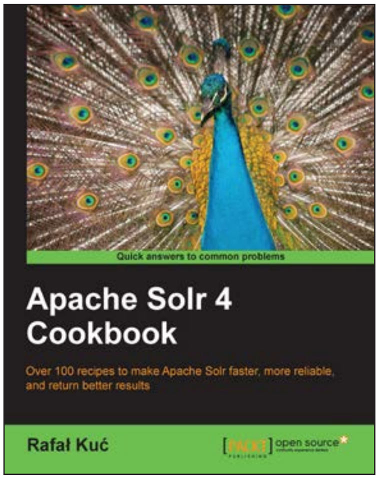
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

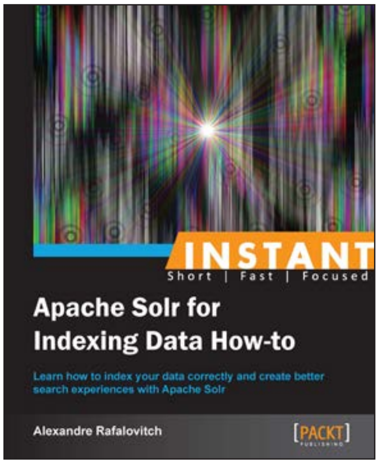

Quick answers to common problems
Apache Solr 4 Cookbook
Over 100 recipes to make Apache Solr faster, more reliable, and return better results
Rafał Kuć
[PACKT] open source

Apache Solr 4 Cookbook

ISBN: 978-1-78216-132-5 Paperback: 328 pages

Over 100 recipes to make Apache Solr faster, more reliable, and return better results

1. Learn how to make Apache Solr search faster, more complete, and comprehensively scalable
2. Solve performance, setup, configuration, analysis, and query problems in no time
3. Get to grips with, and master, the new exciting features of Apache Solr 4


INSTANT
Short | Fast | Focused
Apache Solr for Indexing Data How-to
Learn how to index your data correctly and create better search experiences with Apache Solr
Alexandre Rafalovitch
[PACKT]

Instant Apache Solr for Indexing Data How-to

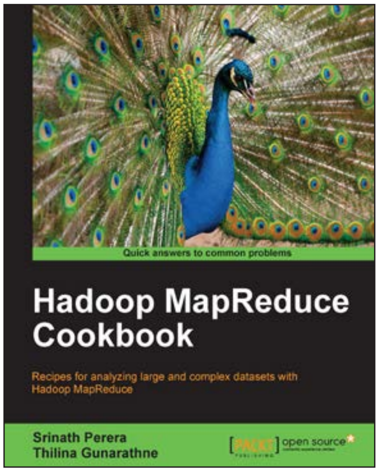
ISBN: 978-1-78216-484-5 Paperback: 90 pages

Learn how to index your data correctly and create better search experiences with Apache Solr

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Take the most basic schema and extend it to support multi-lingual, multi-field searches
3. Make Solr pull data from a variety of existing sources
4. Discover different pathways to acquire and normalize data and content

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
PUBLISHING community experience distilled

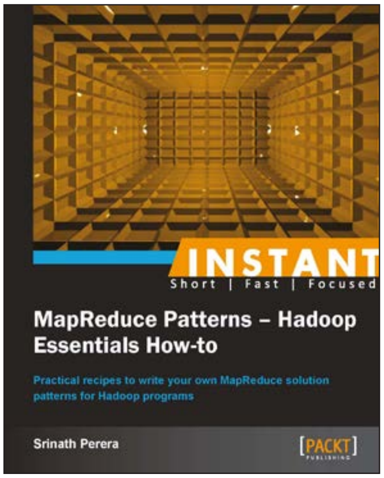

Quick answers to common problems
Hadoop MapReduce Cookbook
Recipes for analyzing large and complex datasets with Hadoop MapReduce
Srinath Perera
Thilina Gunarathne
[PACKT] open source

Hadoop MapReduce Cookbook

ISBN: 978-1-84951-728-7 Paperback: 300 pages

Recipes foe analyzing large and complex datasets with Hadoop MapReduce

1. Learn to process large and complex data sets, starting simply, then diving in deep
2. Solve complex big data problems such as classifications, finding relationships, online marketing and recommendations
3. More than 50 Hadoop MapReduce recipes, presented in a simple and straightforward manner, with step-by-step instructions and real world examples


INSTANT
Short | Fast | Focused
MapReduce Patterns – Hadoop Essentials How-to
Practical recipes to write your own MapReduce solution patterns for Hadoop programs
Srinath Perera
[PACKT]

Instant MapReduce Patterns – Hadoop Essentials How-to

ISBN: 978-1-78216-770-9 Paperback: 60 pages

Practical recipes to write your own MapReduce solution patterns for Hadoop programs

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Learn how to install, configure, and run Hadoop jobs
3. Seven recipes, each describing a particular style of the MapReduce program to give you a good understanding of how to program with MapReduce
4. A concise introduction to Hadoop and common MapReduce patterns

Please check **www.PacktPub.com** for information on our titles