TUHH
Technische Universität Hamburg-Harburg

Master Thesis

Cascalog for sensor data processing

August 20, 2014

Supervised by:
Prof Dr. Ralf Moeller
Christian Neuenstadt

Submitted by:
Sindhu Hosamane,
Information and
Media Technologies (IMT),
Matriculation nr: 46442

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS Software Technology Systems

Statutory Declaration

I, Sindhu Hosamane, declare that I have authored this thesis independently, that I have not used other than the declared sources/resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. Neither this thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, August 20, 2014

..... ……

Sindhu Hosamane

Acknowledgment

On the successful completion of my Master thesis, I would like to thank the University with gratitude. I wish to acknowledge all of them. However, I wish to make special mention of the following.

I take this opportunity to express my profound gratitude and deep regards to my guide Prof. Ralf Moeller for allowing me to write thesis in his department and encouraging me to explore the topic, which became of deep interest to me.

I must make special mention of Christian Neuenstadt my project supervisor for giving me his valuable time, attention and for providing a systematic way for completing my thesis. I appreciate his exemplary guidance and monitoring throughout the course of this Thesis.

I am obliged to staff members of University, for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my master thesis.

Contents

List of Figures

Tables

Chapter 1

Introduction

In today's world the amount of data has been exploding and analysing large datasets, so called big data has become a key basis for competition. The world of big data is dramatically changing right in front of our eyes. Data production is expanding at an astonishing pace. This data growth presents enormous challenges [21] and also provides enormous business opportunities. The dramatic increase of unstructured data like photos, videos and social media data creates the necessity of new breed of non-relational databases, which allows the data to reveal its own structure and patterns. Also today data is increasing beyond the capabilities of traditional databases. To tackle this a new breed of technologies emerged namely many projects from open source community like Hbase [29], Mongo DB [27], Cassandra [28], Hadoop [1] and many more.

There are many tools, which use Hadoop as their execution environment. One such tool is Cascalog, which is used for processing big data on top of Hadoop. The whole project is about understanding Cascalog queries, its structure and simplicity by formulating some Cascalog queries, which queries sensor data used in Optique project [30] by Siemens. Also some performance tests are conducted at the end.

1.1 Outline

Chapter 2 begins with the discussion of what is big data, the way data is exploding today, some current facts and why are traditional databases not suitable for big data. Chapter 3 provides the understanding of Hadoop, its architecture and further on Chapter 4 explains some basics of Clojure programming language. Chapter 5 is an introduction to Cascalog and its query structure. Chapter 6 explains the experiment conducted, that is querying sensor data using Cascalog, which uses Hadoop as execution environment. Results of benchmarking are shown in Chapter 7.

Chapter 2

Big Data

2.1 Definition

Big data is huge amount of data which is difficult to collect, store, manage and analyse via traditional database systems. Big data has the capacity to deal with data, which is in petabytes and Exabyte.

According to, leading IT industry research group Gartner [31] Big Data is defined as: "Big Data are high-volume, high-velocity and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization." [31]

2.1.1 Dimensions of Big Data [5]

Big data relates to data creation, storage, retrieval and analysis that are remarkable in terms of Volume, Velocity and Variety. Any data that fits one or more of above three dimensions is called Big Data.

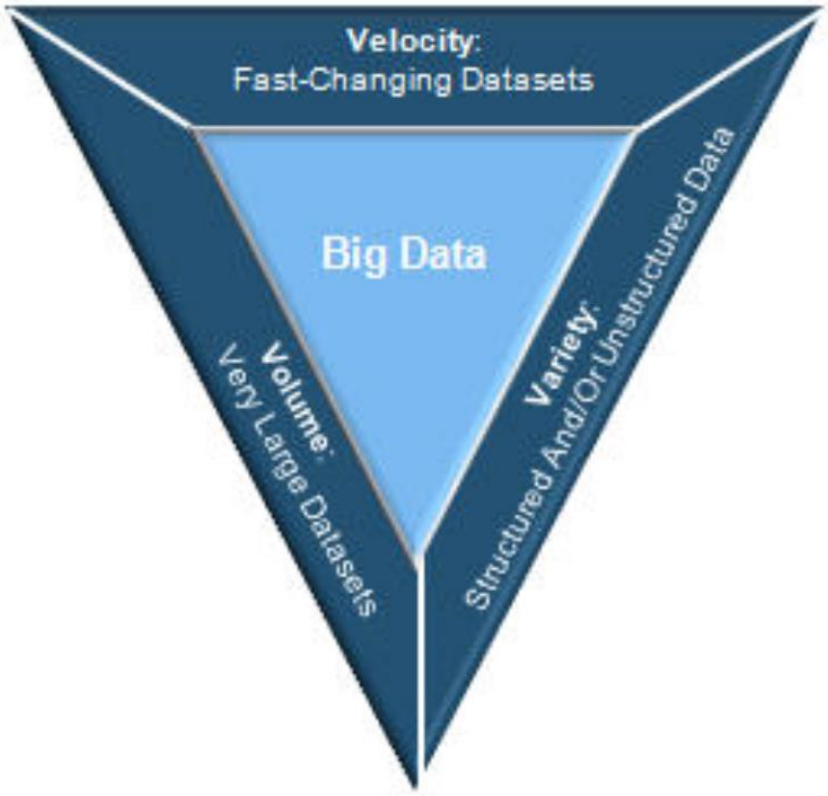


Fig 1: Dimensions of Big Data [5]

Volume: Size of data (how big it is)

- Machine generated data is produced in much larger quantities in petabytes per day.
- A single jet engine can generate 10TB of tracking data in 30 minutes; 25,000 airline flights per day lead to daily volume to Petabytes.
- Facebook ingests 500 terabytes of new data every day.
- A typical PC might have had 10 gigabytes of storage in 2000.
- The proliferation of smart phones, the data they create and consume.
- Sensors embedded into everyday objects will soon result in billions of new, constantly-updated data feeds containing environmental, location and other information.

Velocity: How fast data is being generated

- High velocity of media data streams.
- Even at 140 characters per tweet, the high velocity of Twitter data ensures large volumes (over 8 TB per day).
- On-line gaming systems support millions of concurrent users, each producing multiple inputs per second.
- Sensors generate massive log data in real-time.
- Machine to machine processes exchange data between billions of devices.
- Stock trading algorithms reflect market changes within microseconds.
- Clickstreams and impressions capture user behaviour at millions of events per second.

Variety: Variation of data types to include source, format and structure

- As new services are added, new sensors are deployed; new data types are needed to capture the resultant information. The data arriving is not only structured but also unstructured, semi-structured and multi-structured.
- Big data does not mean a lot of one type of data, but a lot of data of different types. Big data is not only Strings, numbers but also refers to geospatial data, 3D data, audio, video and unstructured text including log files and social media.

Big data is important; those who can harness big data will have edge in critical decision-making. IT companies are investing billions of dollars on research and development of big data. [32] Companies have to extract business critical information from big data that their company requires.

Possible sources of big data are: [5]

- Social media Facebook posts, Twitter tweets.
- Scientific sensors such as global mapping, meteorological tracking, medical imaging and DNA research.
- Intranet and Internet websites across the world.

2.2 Reasons for growth in digital data

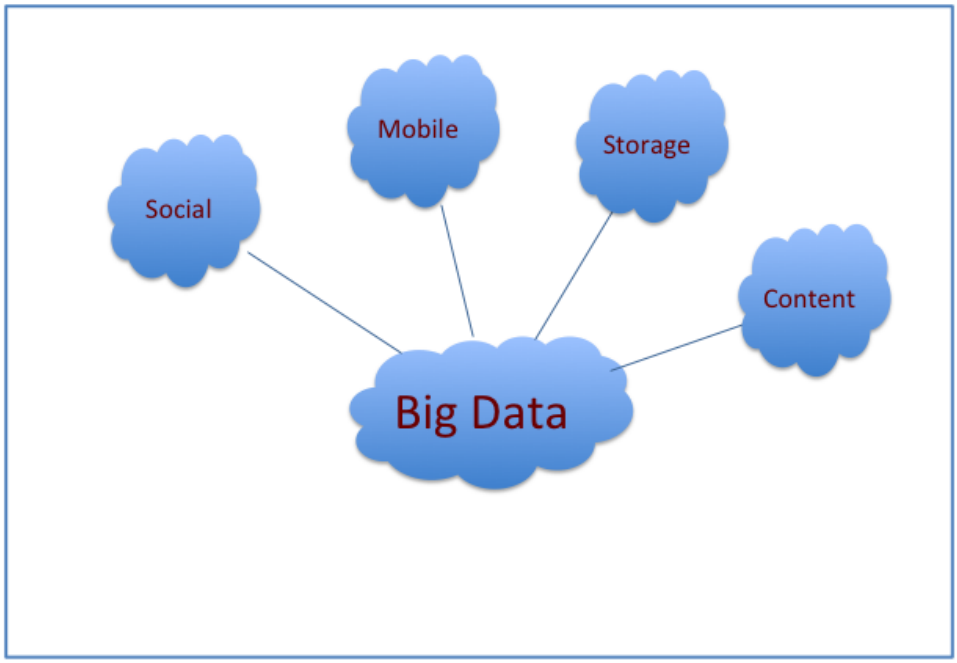


Fig 2: Reasons for growth in digital data

1) Increasing growth of Internet usage, social networks and smartphone adoption:

- Almost everyone is using social networking sites especially Facebook and Twitter. More than 9000 tweets are being generated every second and average number of tweets per day is 58 million. [3] Nowadays, around 3 lac status are updated and more than 1 lac photos are uploaded and 510 comments are posted on Facebook every minute. Five new profiles are created every second. [4]
- For the very first time in history, Smartphones were sold more than feature phones.

2) Falling costs of the technology devices that create digital data:

- Digital products are getting cheaper and cheaper -Smartphones are available for cheaper prices. Smartphones can take images, create documents and do all sorts of things thus increasing digital data.
- We upload lots of videos on YouTube as it's free and lots of comments and likes are posted for videos uploaded and thus lot of data is being generated.

3) Growth of machine generated data:

- Data from Satellites, sensor readings from factories, data generated by equipment (tractors, vehicles) like fuel consumption, temperature etc. all contributes to the increasing data. This data is growing at a great speed with more industrialisation.
- The machine-generated data will account for 40% of the Digital Universe by 2020, up from just 11% in 2005 as per Digital Universe estimation. [5]

2.3 Some facts about Big Data

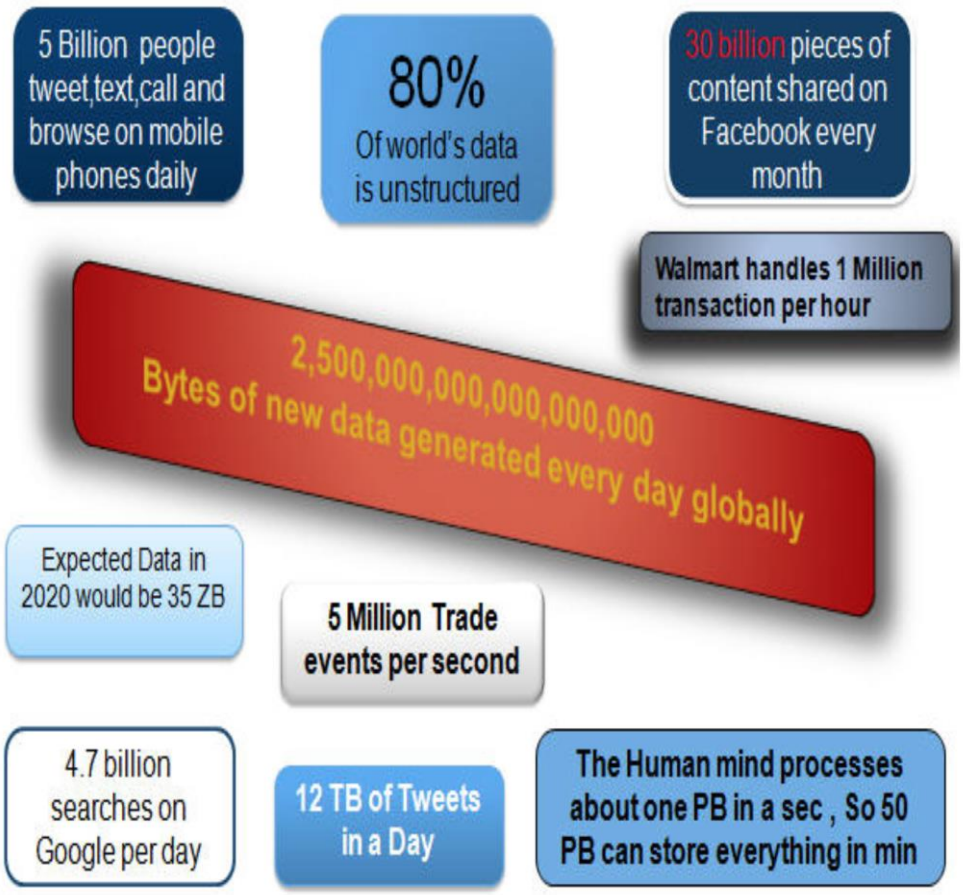


Fig 3: Facts about Big Data [5]

Fig 3 shows how digital data is growing enormously. By 2011, more than 1.8 Exabyte was created and by 2020 more than 35,000 Exabyte of information will be created, which is 20 times more than in 2011, which obviously says the information to be managed by business increases.

Situation today: According to the Digital Universe Ticker, around 3.2 Zettabyte of information has been created since 1st January 2013, which in turn also shows that Terabytes of data is being generated every second. [2]

As per 2012 Digital Universe Study:

- The amount of information in the Digital Universe is doubling every two years, currently growing at a rate of more than 7,600 Petabytes per day.
- Less than 1% of the world's data is analysed today, which presents an enormous opportunity for Big Data analytics.

According to Foreignaffairs: [35]
In year 2000, only one-quarter of the entire world's stored information was digital. The rest of the information was preserved on paper, film and other analog media. Today, less than two percent of all stored information is non-digital, because of the enormous amount of digital data.

According to MGI Survey: [36]
The use of big data will underpin new waves of productivity growth and consumer surplus.

According to InformationWeek: [34]
A recent survey by database vendor says that the better management of big data can make smarter business decisions of the organization. And also more than 30% of the people said analysing big data is a challenge.

According to Gartner: [31]
By 2015, 4.4 million IT jobs will be created globally to support big data, generating 1.9 million IT jobs in the US. In addition, every big data-related role in the U.S will create employment for three people outside of IT, so over the next four years a total of 6 million jobs in the U.S will be generated by the information economy.

Nowadays, big data is being used in all the sectors like Healthcare, banking, Retail and many more. [33] Based on the above big data facts, it is evident that in the near future big data is going to play a very important role.

Big data is the biggest trend in IT right now. We all know the amount of data and the speed at which it is accumulating is growing exponentially. This can be a difficult problem to tackle in and of itself, however, failing to take advantage of the Variety of data, the Volume and Velocity become much more of a downside than a value-add.
Big data is very much needed because traditional databases can't analyse data from Social media, data from Videos, data from sensors as this type of data grows at a great speed and it is also beyond the processing capacity of traditional databases.

According to CEO at Genalice: The challenge for us is to find the answer we‘re looking for from that mountain of data. There is a lot of knowledge encrypted in that data, and our challenge is to pick it out. Our concern is not the size of the data, but its diversity and complexity. He also mentions the speed at which we receive medical data is so rapid that you could almost call it a data tsunami.

2.4 Big Data for the Enterprise [23]

Big data is about using the huge amount of data to extract valuable information from it and to get the right answer to the questions that can take company a step forward. Big data is a technology, a new database technology that uses new hardware platforms to access huge amounts of data quickly.
Big data is about asking the right questions so that the correct analysis of both structured and unstructured data produces the right answer.

The competitiveness of the company stands or falls based on the quality and speed of information provided. Nowadays data growth is accelerating; so it is important for the

companies to be able to respond faster, better and creatively over the changes in the market, which gives a major advantage over the competition. Big data is the ultimate weapon. Companies must therefore be able to retrieve increasingly refined information from an increasing number of sources. If Companies don't understand their customers and do not respond adequately, then there are risks of losing the customers.

According to French American company Alcatel-Lucent, "No organization can avoid Big Data ". The rapidly expanding market for diverse mobile devices, the increasing mobility of their users, the growing need for real-time information and the wide availability of more and more sources of information including social media, all place high demands to use new technologies to handle big data.

Big data is no hype or revolution. It is simply an evolution. Companies should not underestimate or ignore it. But instead try to develop big data applications in order to remain competitive. Big data is used in variety of organizations. For example, banks use big data to improve risk analysis and fraud detection. It is also used in Weather forecasts where weather keeps varying constantly and in high frequency stock markets, which keeps changing for every microsecond.

2.5 Why not traditional databases for Big Data? [40]

The Internet, ultimate source of data is incomprehensively large. There are varieties of data we have to deal with. Users create blogs, tweets, conversations on social networking sites, upload photos. Servers produce data continuously about what is going on. This astonishing amount of growth in data is termed as big data, which we understood in previous sections.

Traditionally, Databases and Business Intelligence tools were used to give a sense of the data. Big data goes beyond the conventional analytical tools and databases and it deals with a variety of data. If the data is sensibly arranged and is easy to partition the set of data, such data is homogeneous. This data is easily handled by traditional databases. But in today's world we have a lot of unstructured data from social media, videos, audios which all doesn't fall under homogeneous category of data. Not only is the data too unstructured and too voluminous for a traditional RDBMS, the software and hardware costs require to crunch through these new data sets using traditional RDBMS technology are prohibitive. So here comes the necessity of a technology that deals with data, which is not necessarily as homogeneous or as structured as needed in traditional databases. Choosing wrong databases for wrong workload may end up burning pockets.

Big data will be in real time whereas; traditional databases were historic in reporting. Data is coming in at faster rate than it can be processed. Big data exceeds the processing capacity of our traditional database systems. So, traditional database systems such as relational databases are pushed to their limits to deal with Big Data. They can be fine-tuned to some extent but ultimately it hits a wall. Sometimes these traditional databases break under the pressure of "Big Data". It is difficult for traditional databases to scale to big data.

2.5.1 Scaling with traditional databases

We consider an example to see how scaling works with traditional databases. Consider building a web application that gives the pageviews to any URL that the user wish to track. Then when someone views the page the customer's webpage pings the webserver with its URL. In addition we also want the application to tell top 100 URL's by the number of pageviews.

We start with a traditional relational schema for the pageviews that looks something like in Table 1. Whenever a webpage is being viewed, then webpage pings the webserver with a pageview, then webserver in turn increments the corresponding row in the RDBMS. [37]

Column Name	Type
id	integer
user_id	integer
url	Varchar (255)
pageviews	bigint

Table 1: Relational schema for simple web analytics application

This seems to make sense atleast in the world before big data. But with big data it might run into problems in terms of scale and complexity.

2.5.2 Scaling with a queue

Since it is a web analytic product there could be a lot of traffic to the application. When the data is growing enormously, then the database can't keep up with the load, so write requests to increment pageviews are timing out. The fix for this problem could be instead of webserver hit the database directly; a queue can be inserted between webserver and database. When a pageview occurs, an event is added on to the queue. Then a worker process can be created that reads 1000 events at a time off the queue and batches them into a single database update. This could resolve timing out problem. The problem with this is when the database gets overloaded again and then the queue grows bigger without timing out the webserver and potentially losing data. Adding a queue and creating a worker to do batch updates is simply a band-aid to the scaling problem.

2.5.3 Scaling by sharding the database

When the database gets overloaded then the worker can't keep up with the writes. So more workers need to be added to parallelize the updates, which doesn't seem to work because the database is a bottleneck.

Next method would be using multiple database servers, which spread the table across all servers. Each server will have a subset of data for the table. This is known as sharding. Using this technique, write load is spread across several machines. The technique used to shard the database is to choose the shard of the key. A script has to be written to map over all the rows in our single database instance and split the data into shards.

As the application becomes more popular, the database has to be reshard into more

number of shards to keep up with the huge write load. Then running a single script would be slow. Multiple scripts have to be managed which obviously increases complexity.

Fault-tolerance issues begin

When there are multiple database servers, if one of the disks on one of the database machine goes bad, so that portion of data is not available as that machine is down. Then we should think of methods that address this issue. All the time needs to be spent in solving the problems of reading and writing the data whereas building new features for customers is a far thought.

Corruption issues

While working on a worker/queue if a bug is deployed, which increments pageview by two instead of one and we don't notice it for a day then many values in the database are inaccurate. Also no previous backups could help to resolve this issue, as there is no way of knowing which data was corrupted. So here we see there is no resilience in the system to human making a mistake.

Analysis of problems with traditional architecture

In developing a web analytics application we started with a webserver, one database and then ended up with queues, creating workers, replicas and multiple servers. Scaling the application forced the backend to become much more complex and operating the backend becomes even more complex. Some of the challenges while developing this application:

- Fault-tolerance is hard
- Complexity pushed to application layer
- Lack of human fault-tolerance
- Maintenance is an enormous amount of work

Considering above example it is proved that traditional databases are perfectly not suitable for big data. In order to capitalize on the Big Data trend, a new breed of big data companies has emerged, leveraging commodity hardware, open source and proprietary technology to capture and analyse these new data sets, the one of which is discussed in chapter 3.

Database pioneer and researcher Michael Stonebraker discusses the limitations of traditional database architectures. "Generally, traditional databases scale up with more expensive hardware, but have difficulty scaling out with more commodity hardware in parallel and are limited by legacy software architecture that was designed for an older era." He contends that the big data era requires multiple new database architectures that take advantage of modern infrastructure and optimize for a particular workload. [38]

Data analyst Goldmacher's argument that big data will crush traditional database companies revolves around cost. Emerging big data players can price better than large database players like Oracle that have margins to protect. In other words, Oracle would have to charge 9 times more than the blended average of big data vendors to solve data conundrums. [39]

Chapter 3

Hadoop

3.1 Introduction to Hadoop

As seen in the previous chapter in section 2.5, it is clear that it is very complex to deal with big data with traditional databases. So to tackle the challenges of big data, a new breed of technologies emerged. In some way these new technologies are complex than traditional databases and in some ways they are simpler. Google pioneered many of the big data systems. And in the later years open source community responded with many projects like Hadoop [1], Hbase [42], Mongo DB [27], Cassandra [28] and many other projects. In this chapter we understand the details of Hadoop.

Hadoop is a Java-based programming framework, which supports the processing of large data sets in a distributed computing environment. It processes large data sets across clusters of computers using simple programming models. It is designed to scale up from single server to thousands of machines, each of which provides local computation and storage. Hadoop began its life at Yahoo and now is a part of the Apache project sponsored by the Apache Software Foundation. [1]

Hadoop makes it possible to run applications on systems with thousands of nodes involving petabytes of data. It also facilitates rapid data transfer rates among nodes, because of its distributed file system. Hadoop has a high degree of fault tolerance, rather than relying on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer. This allows the system to continue operating without any interruptions in case of a node failure. This approach lowers the risk of catastrophic system failure, even if a significant number of nodes become inoperative.

Google's MapReduce, a software framework in which an application is broken down into numerous small parts, inspired Hadoop. Any of these parts (also called fragments or blocks) can be run on any node in the cluster. Doug Cutting, Hadoop's creator, named the framework after his child's stuffed toy elephant. Currently Apache Hadoop ecosystem consists of the Hadoop kernel, MapReduce [44], the Hadoop distributed file system (HDFS) [43] and a number of related projects such as Apache Hive [29], Hbase [42] and Zookeeper. [41] Together they form a powerful framework for running large datasets ranging from terabytes to petabytes. The Hadoop framework is used by major players largely for both research and production namely Google, Yahoo, IBM and many other. The preferred operating systems for Hadoop are Windows and Linux but can also work with BSD and OS X.

3.1.1 Salient characteristics of Hadoop:

Hadoop changes the economics and dynamics of large scale computing. Hadoop enables a computing solution that is:

- Scalable –New nodes can be added without changing the data format, how data is loaded, how jobs are written.

- Cost effective –Hadoop results in a sizeable decrease in the cost per terabyte of storage, which in turn makes it affordable to model all the data.
- Flexible –Hadoop absorbs any type of data, structured or unstructured from any number of sources. So Hadoop can be regarded as schema-less.
  It enables deeper analysis of data by joining data from multiple sources and aggregating it in arbitrary ways.
- Fault tolerant –When a node in a cluster goes down, then the system redirects the work to another node, which contains the data and continues without any human interruptions.

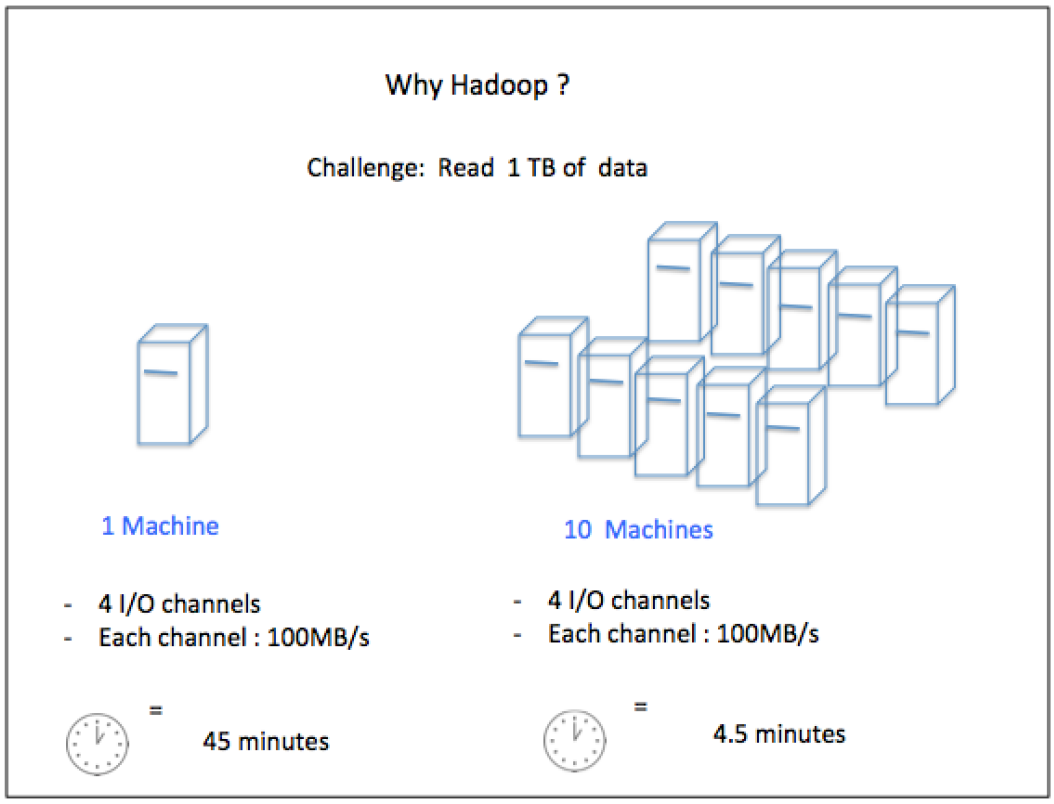


Fig 4: Why Hadoop?

The Fig 4 explains the scenario when 1 terabyte of data is to be read. On the left there is one machine with 4 hard drives and each drive having 100MB/s capability of reading files from the disk. So if 1 terabyte of data is to be read, then data is scattered evenly on each drive. So each hard drive will have 250GB of data. So to read 1 terabyte of data from this machine, it would take 45 minutes. This is just a theoretical calculation based on how long it would take to read from a 250GB drive at 100MB/s. If we take the same 1 terabyte of data and scatter it across for example a 10-node cluster with 10 worker machines as shown on right side of Fig 4, so now we are putting one-tenth of a terabyte on each of the machines, further more each machine also has 4 hard drives. So now across the 10 machines there are 40 hard drives each holding 25GB of data. Now to read 1 terabyte of data from 10 machines in parallel would take only 4.5 minutes. So by harnessing the power of multiple machines together, reads and writes can be processed significantly faster. The above scenario clearly explains the power of distributed computing environment and thereby Hadoop.

3.2 Hadoop Installation: [14]

3.2.1 Single node Hadoop installation: (on Linux)

Prerequisites:

Java
Java 1.5x is required. If it is already installed, try typing java -version at the command line to see if we have it installed. It should say version "1.5.x" (e.g., 1.5.0_14).

Adding a dedicated Hadoop system user
Create a dedicated Hadoop user account for running Hadoop. While that's not required it is recommended, because it helps to separate the Hadoop installation from other software applications and user accounts running on the same machine.

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop hduser
```

This will add the user hduser and the group hadoop to our local machine.

Configuring SSH
Hadoop requires SSH access to manage its nodes, i.e. remote machines plus our local machine if we want to use Hadoop on it (which is what we want to do in this short tutorial). For single-node setup of Hadoop, we therefore need to configure SSH access to localhost.

Assuming SSH is up and running on our machine and configured it to allow SSH public key authentication. First, we have to generate an SSH key for the hduser user.

```
user@ubuntu:~$ su - hduser
hduser@ubuntu:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hduser/.ssh/id_rsa):
Created directory '/home/hduser/.ssh'.
Your identification has been saved in /home/hduser/.ssh/id_rsa.
Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.
The key fingerprint is:
9b:82:ea:58:b4:e0:35:d7:ff:19:66:a6:ef:ae:0e:d2 hduser@ubuntu
The key's randomart image is:
[...snipp...]
hduser@ubuntu:~$
```

Second, SSH access has to be enabled to our local machine with this newly created key as seen above.

```
hduser@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >>
$HOME/.ssh/authorized_keys
```

The final step is to test the SSH setup by connecting to our local machine with the hduser user. The step is also needed to save our local machine's host key fingerprint to the hduser user's known_hosts file.

```
hduser@ubuntu:~$ ssh localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is
d7:87:25:47:ae:02:00:eb:1d:75:4f:bb:44:f9:36:26.
Are you sure you want to continue connecting (yes/no)? Yes
Warning: Permanently added 'localhost' (RSA) to the list of known
hosts.
Linux ubuntu 2.6.32-22-generic #33-Ubuntu SMP Wed Apr 28 13:27:30 UTC
```

```
2010 i686 GNU/Linux
Ubuntu 10.04 LTS
[...snipp...]
hduser@ubuntu:~$
```

Installation:

Download Hadoop from the Apache Download Mirrors and extract the contents of the Hadoop package to a location to say /usr/local/hadoop. Make sure to change the owner of all the files to the hduser user and hadoop group, for example:

```
$ cd /usr/local
$ sudo tar xzf hadoop-1.0.3.tar.gz
$ sudo mv hadoop-1.0.3 hadoop
$ sudo chown -R hduser:hadoop hadoop
```

Update $HOME/.bashrc

Add the following lines to the end of the $HOME/.bashrc file of user hduser.

```
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin
```

Configuration:

hadoop-env.sh

JAVA_HOME environment variable should be set in conf/hadoop-env.sh

```
# The java implementation to use.  Required.
export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

conf/*-site.xml

In this we will configure the directory where Hadoop will store its data files, the network ports it listens to etc.

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/tmp</value>
  <description>A base for other temporary directories.</description>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:9000</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class.  The uri's authority is used
to
  determine the host, port, etc. for a filesystem.</description>
</property>
```

In file conf/mapred-site.xml:

```
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:9001</value>
  <description>The host and port that the MapReduce job tracker runs
  at.  If "local", then jobs are run in process as a single map
  and reduce task.
```

```
  </description>
</property>
```

In file conf/hdfs-site.xml:

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>Default block replication.
  The actual number of replications can be specified when the file is
created.
  The default is used if replication is not specified in create time.
  </description>
</property>
```

Formatting the HDFS filesystem via the Namenode:

The first step to start our Hadoop installation is formatting the Hadoop filesystem using below command

```
hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format
```

Starting single-node cluster:

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

Running this command will start Namenode, Datanode, Jobtracker and a Tasktracker on our machine. Jps is a tool for checking whether the expected Hadoop processes are running.

```
hduser@ubuntu:/usr/local/hadoop$ jps
2287 TaskTracker
2149 JobTracker
1938 DataNode
2085 SecondaryNameNode
2349 Jps
1788 NameNode
```

Stopping single-node cluster:

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

3.2.2 Multinode Hadoop installation

In our experiment we set up multiple datanodes on a single machine, which is useful for small test case scenarios. Below are the steps to be followed:

1. In HADOOP_HOME directory, copy the "conf" directory to, say, "conf2".
2. In the conf2 directory, edit as follows:
    a) In hadoop-env.sh, provide unique non-default HADOOP_IDENT_STRING, e.g. ${USER}_02
    b) In hdfs-site.xml, change dfs.data.dir to show the desired targets/volumes for datanode#2 and of course make sure the corresponding target directories exist. Also remove these targets from the dfs.data.dir target list for datanode#1 in conf/hdfs-site.xml.
    c) In hdfs-site.xml, set the four following "address:port" strings to something non-conflicting with the other datanode and other processes running on this box:
    - dfs.datanode.address (default 0.0.0.0:50010)

- dfs.datanode.ipc.address (default 0.0.0.0:50020)
- dfs.datanode.http.address (default 0.0.0.0:50075)
- dfs.datanode.https.address (default 0.0.0.0:50475)

3. At this point, launching with:
   bin/hdfs --config $HADOOP_HOME/conf2 datanode

3.3 What problems can Hadoop solve? [24]

According to cloudera CEO who explains it as "The Hadoop platform was designed to solve problems where you have a lot of data —perhaps a mixture of complex and structured data —and it doesn't fit nicely into tables. It's for situations where you want to run analytics that are deep and computationally extensive, like clustering and targeting. That's exactly what Google was doing when it was indexing the web and examining user behaviour to improve performance algorithms.

Hadoop applies to a bunch of markets. In finance, if you want to do accurate portfolio evaluation and risk analysis, you can build sophisticated models that are hard to jam into a database engine. But Hadoop can handle it. In online retail, if you want to deliver better search answers to your customers so they're more likely to buy the thing you show them, that sort of problem is well addressed by Hadoop. Those are just a few examples."

3.4 Hadoop Architecture: [25]

Hadoop consists of a common package, Mapreduce engine (MapReduce/YARN) and Hadoop distributed file system (HDFS). Hadoop common package consists of Java Archives (JAR'S) files and scripts needed to start Hadoop.

A small Hadoop cluster consists of a single master node and multiple worker nodes. Master node consists of Jobtracker, Tasktracker, Namenode and Datanode. Worker node or slave node acts as both Datanode and Tasktracker. Whereas in a larger cluster, the HDFS is managed through a dedicated NameNode server to host the file system index and a secondary NameNode that can generate snapshots of the namenode's memory structures, thus preventing file-system corruption and reducing loss of data. Fig 5 shows how a multinode Hadoop cluster looks like.

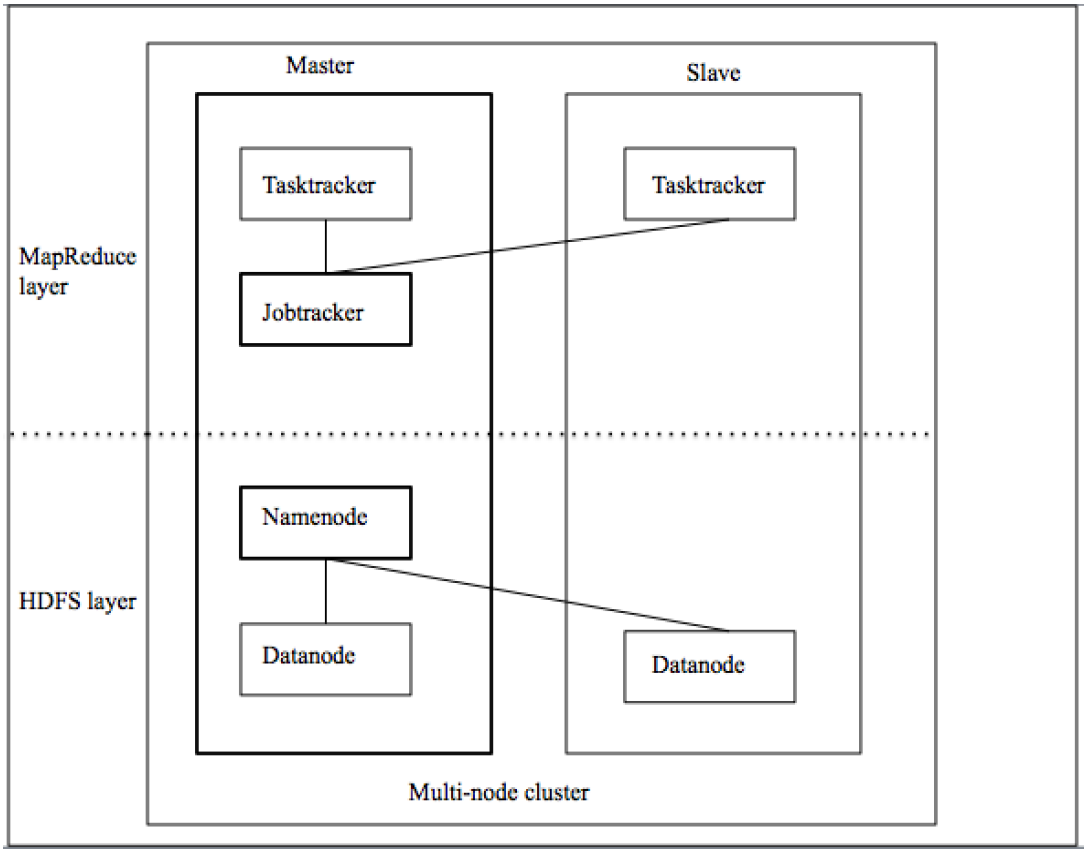


Fig 5: A multinode Hadoop cluster [25]

3.4.1 Hadoop Distributed file system (HDFS)

HDFS is a distributed file system that can store huge amounts of data like terabytes or even petabytes. Data on HDFS is stored in distributed fashion across multiple machines, which ensures high availability for parallel computation and makes it resilient to failures. It provides high throughput to application data and is very much suitable for large datasets. It remains economical at every size by providing distributed storage and computation across many servers.

A distributed file system (DFS) is designed to hold large amount of data that is in terabytes or petabytes and make this data accessible to all the clients across the network. The file system uses TCP/IP sockets for communication. Clients use remote procedure call (RPC) to communicate between each other. HDFS was designed to combat the problems of other DFS's like Network File system (NFS). It has similarities like that of other DFS, but it has significant differences as well like highly fault-tolerant and is designed to deploy on low-cost hardware. In particular:

- HDFS is designed to store large amounts of data (terabytes or petabytes). It also supports larger file sizes.
- HDFS stores data reliably. If individual machine in cluster goes down, data is still available.
- HDFS provides fast, scalable access to data. If needed to serve more number of clients then adding more machines to the cluster serves the purpose.
- HDFS integrates well with Hadoop MapReduce, which allows data to be read and computed locally.

HDFS has a master/slave architecture. HDFS cluster consists of one Namenode, a master server that manages file system and access to files by clients. There can be number of datanodes usually one per node in the cluster which manages storage attached to the node they run on. Also having a single Namenode greatly simplifies the architecture. Hadoop uses HDFS to store files efficiently in the cluster. HDFS is a block structured file system that is, when a file is placed on HDFS it is broken down into blocks of fixed size; 64 MB being the default block size. These blocks are stored across one or more machines with data storage capacity. These individual machines are called datanodes. It's not essential that all the blocks of a file are stored on same machine. The target for each block is chosen randomly on block –by-block basis. Thus access to one file may require cooperation of multiple machines. So if one of those machines goes down, then data in that machine would become unavailable. HDFS solves this problem by replicating blocks across different nodes (Datanodes) in the cluster. The default replication value is 3, which means there will be 3 copies of same block in the cluster.

3.4.2 MapReduce

It is a programming model that uses java as programming language for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It is batch-based modelled after Google's paper on MapReduce. It allows parallelizing work over large amounts of raw data. The MapReduce program consists of Map() and Reduce() procedure. Map() procedure performs filtering and sorting. Reduce() procedure performs a summary operation. The MapReduce paradigm is very powerful and has the ability to process data with distributed computing, without having to deal with concurrency, robustness and scale.

3.4.3 Different components of Hadoop Cluster

A fully configured cluster runs on a set of daemons: The daemons are described below:

Namenode: Namenode is the vital of Hadoop daemons which directs the slave Datanode daemons to perform low-level I/O tasks. Namenode operates file system namespace operations like opening, closing and renaming files and directories. It is a bookkeeper of HDFS. It keeps track of how a file is broken into blocks and which nodes store these blocks and the overall health of the distributed file system. The system is designed in such a way that, Namenode neither stores any data nor performs any computation for MapReduce programs. It is the repository of HDFS metadata as shown in Fig 6 below. In order to keep less metadata, Namenode only tracks filename, permissions and location of each block of each file. All this information can be stored in the main memory of the Namenode machine, allowing fast access to metadata. The figure says the files foo and bar are split into blocks 1, 2, 4 and 3, 5 respectively, which are stored in 3 datanodes with a replication value of 2. Whenever a file is placed in a cluster a corresponding entry of its location is maintained by Namenode. So, for files foo and bar there would be something like below in Namenode:

foo – Datanode1, Datanode2, Datanode3

bar – Datanode1, Datanode2, Datanode3

This information is needed when retrieving data from the cluster as the data is spread across multiple machines.

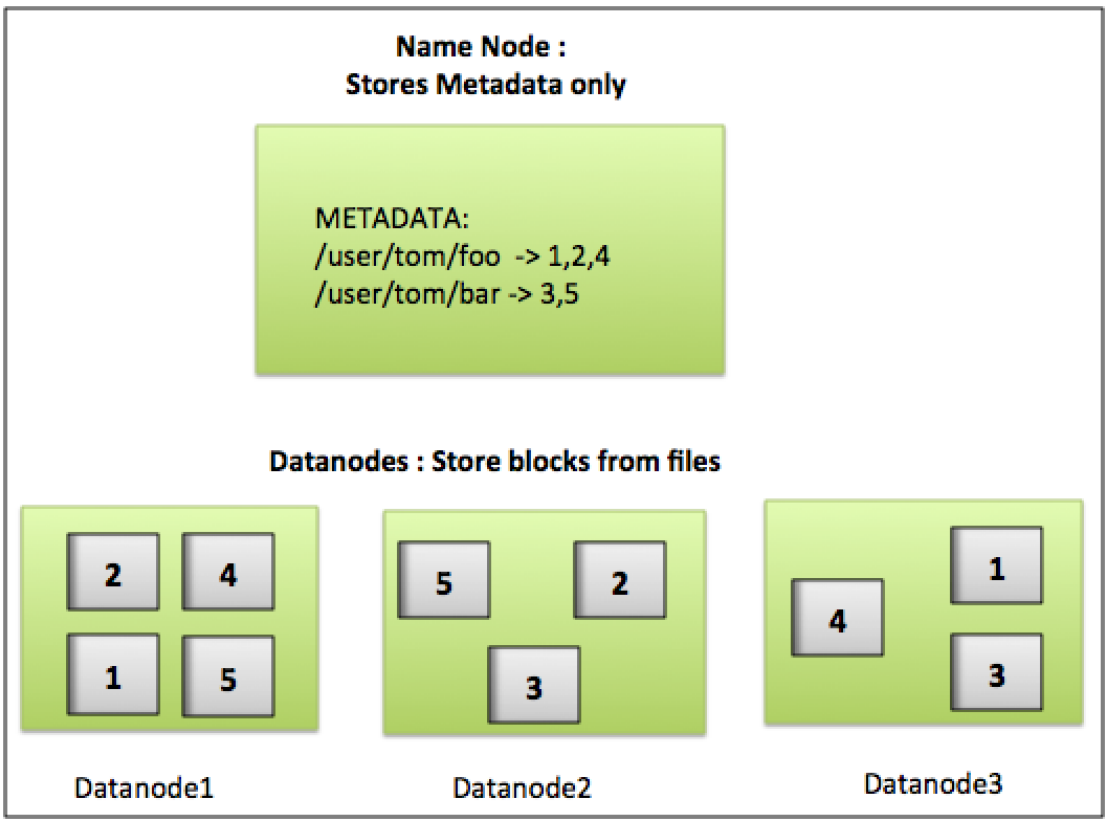


Fig 6: Mapping blocks of file on Datanode and Namenode [1]

To open a file, a client contacts the Namenode to obtain the list of locations of blocks that comprise the file. These locations identify datanodes that hold the blocks. Clients can directly read file data directly from Datanode servers, possibly in parallel. Namenode does not involve in these data transfers, keeping its overhead to a minimum. Since the involvement of a Namenode is relatively less, chances of Namenode failures are lower than that of Datanode failures.

The limitation of Namenode is, it is the single point of failure in the Hadoop cluster. There are many redundant systems that allow Namenode to preserve file systems metadata, even if the Namenode crashes unrecoverably.

Datanode: Datanode is basically the part of a slave machine in the cluster. It is responsible for storing the files in HDFS. It manages the blocks of files within the node. It can read or write the HDFS file. Datanode also performs block creation, move and deletion upon instructions from Namenode. During start-up it makes a connection to the Namenode and performs a handshake. The purpose of this handshake is to verify Namespace Id and software version of a Datanode. If it does not match, Datanode will automatically shut down. Namespace Id is assigned to the file system when it is formatted. This namespace Id is stored on all datanodes in the cluster. Datanode with different namespace Id's cannot join the cluster. This protects the integrity of the file system.

After the handshake Datanode registers itself with the Namenode. Datanodes stores storage ID's. This is assigned to the Datanode when it registers with the Namenode for the first time and remains unchanged after that. Storage ID is a unique identifier of

Datanode, through which Namenode identifies Datanode, even if it is restarted with a different IP address and port.

Datanode sends block report to Namenode, which identifies the block replicas it possess. Block report contains block ID, length of each block, generation stamp. First block report is sent immediately after Datanode registers with Namenode. And then subsequent block reports are sent every one hour which keeps Namenode updated about where the blocks are located.

During the normal operation Datanode sends heartbeat to the Namenode, which signifies Namenode that Datanode is in operation and data blocks in it are available. The interval for sending heartbeat is 3 sec. If the Namenode does not receive heartbeat within 10 minutes, then Namenode considers Datanode is not in operation and the data blocks in it are unavailable. Then Namenode schedules the replication of those blocks in other Datanodes. Thus ensuring even if one Datanode becomes unavailable on the network, data is still available. Heartbeat also contains information like total storage capacity, storage in use, number of data transfers currently. This helps Namenode to make decisions on block allocation and load balancing between datanodes. Namenode does not make direct request with Datanode. It sends replies to heartbeats, which include instructions to remove local block replicas, replicate the block on other datanodes, to send immediate block report, restart, or shut down. Fig 7 indicates the functions of Namenode and Datanode.

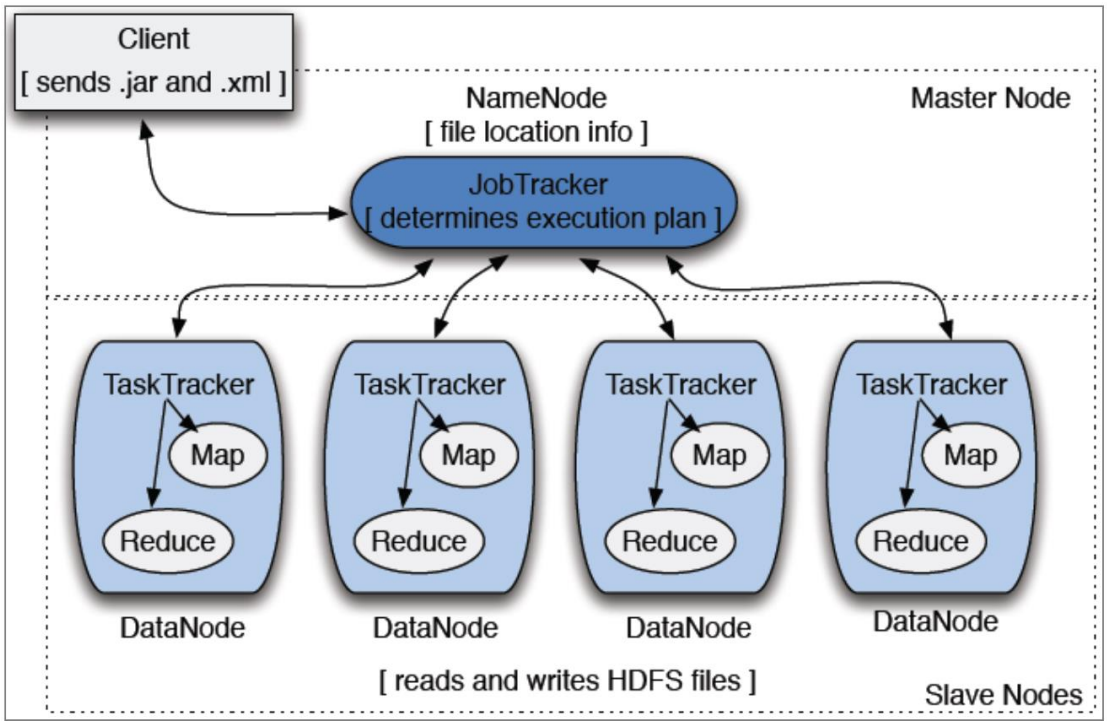


Fig 7: Functions of Namenode and Datanode [15]

Jobtracker: There is only one Jobtracker in a cluster, which is present on the server as a master node of the cluster. It is the communication between application and Hadoop. A Mapreduce engine consists of one Jobtracker to which client applications submit MapReduce jobs. The Jobtracker is responsible for taking in requests from

clients, determines the execution plan by determining which files to process, assigning which tasks are to be handled by which tasktracker and monitoring all tasks as they are running. If a task fails, then the Jobtracker tries to relaunch the task on a different node. Jobtracker tries to assign task to the tasktracker on Datanode where data is located. If not possible it assigns the task to another tasktracker. When the client calls the Jobtracker to start a data processing job, then it splits the work into different map and reduce tasks that is handled by each tasktracker in the cluster. Fig 8 indicates the interaction between Jobtracker and Tasktracker.

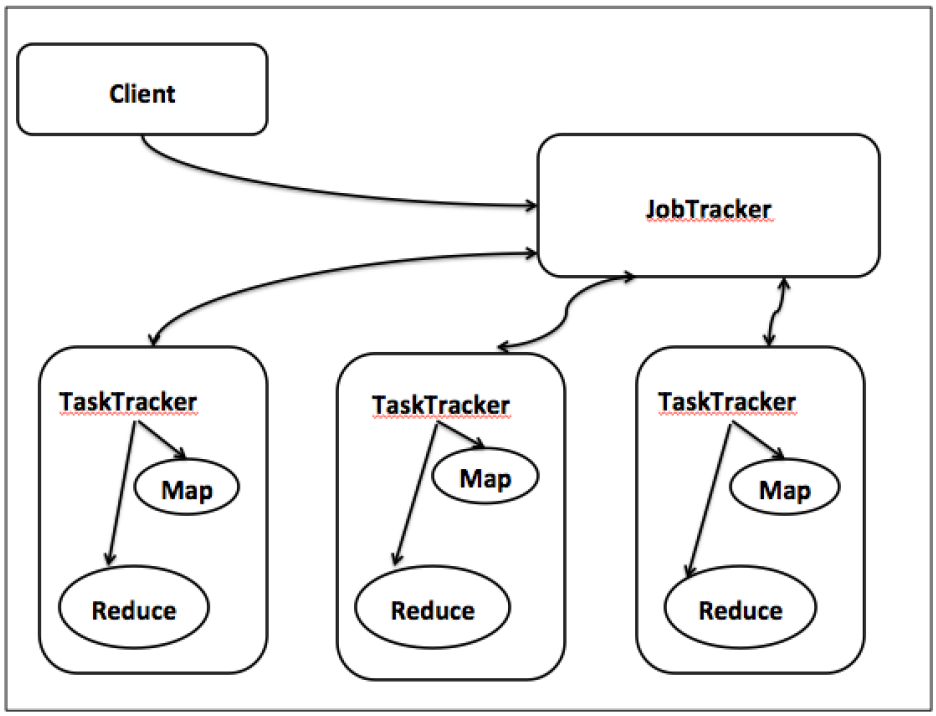


Fig 8: Jobtracker and Tasktracker interaction [15]

Tasktracker: Tasktracker daemon accepts tasks (Map, Reduce and shuffle) from Jobtracker. It is in constant communication with the Jobtracker. It is responsible for executing individual tasks assigned by Jobtracker and sends the progress/status back to the Jobtracker. Whereas Jobtracker is the master overseeing the overall execution of a MapReduce job. Even though there is only one Tasktracker per slave node, Tasktracker can spawn multiple JVM's to execute many map and reduce tasks in parallel. Tasktracker sends heartbeat to Jobtracker, which notifies Jobtracker that it is alive. If it does not receive within certain amount of time, then considers Tasktracker is not in operation and then assigns that task to another Tasktracker.

Secondary Namenode (SNN): It is an assistant daemon that monitors the state of clusters HDFS. There is only one secondary Namenode in the cluster. It is usually run on other machine than primary Namenode because its memory requirements are of same order as that of primary Namenode. SNN does not receive or record any real-time changes of HDFS. It just takes snapshots of HDFS metadata in time intervals specified by cluster configuration.

SNN is a poorly named Hadoop component. Secondary Namenode is just a helper node for Namenode which helps it to function better. It is never a backup or replacement for Namenode. Namenode stores metadata information in main memory

when in use. But it also stores it on disk for persistent storage. Fig 9 shows how Namenode stores information on disk.

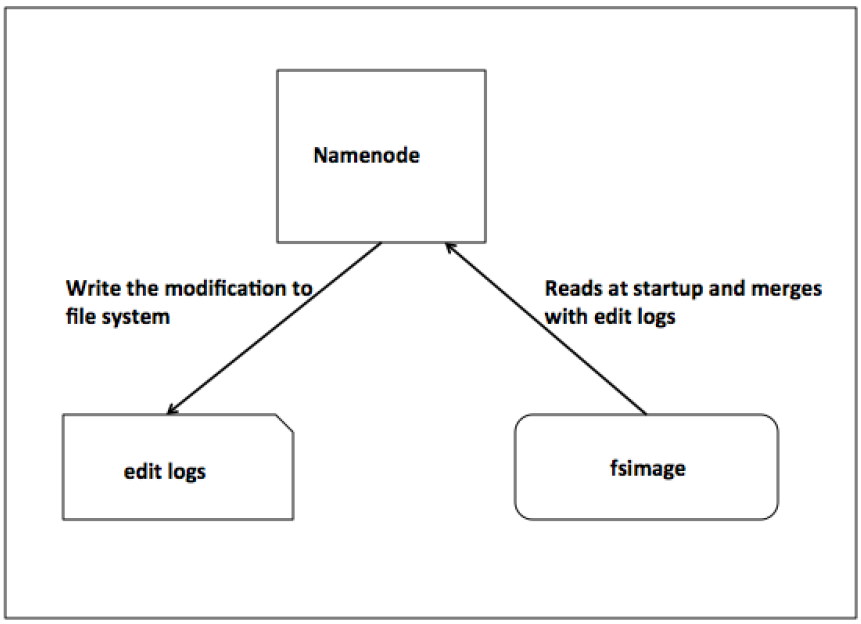


Fig 9: Namenode storing information on disk

fsimage - It is the snapshot of the file system when the Namenode started
edit logs – It is the sequence of changes made to the file system after the Namenode started.

Only during the restart of Namenode edit logs are applied to fsimage to get the updated fsimage. But normally in production clusters restart of Namenode is rare. So edit logs could grow larger and larger over a period of time on a busy cluster, which becomes difficult to manage. And during rare restarts of Namenode since lot changes in edit logs has to be applied, restart can take longer time. In case of crash, lot of metadata could be lost, since fsimage is very old. These were the challenges to keep the metadata up to date.

Secondary Namenode takes the responsibility of merging edit logs with fsimage from the Namenode. Secondary Namenode gets the edit logs from Namenode in regular intervals and applies it to fsimage. Once it has a new fsimage, it copies it back to Namenode. Namenode will use this updated fsimage on restart, which will reduce the start-up time. It keeps edit logs size within limit.

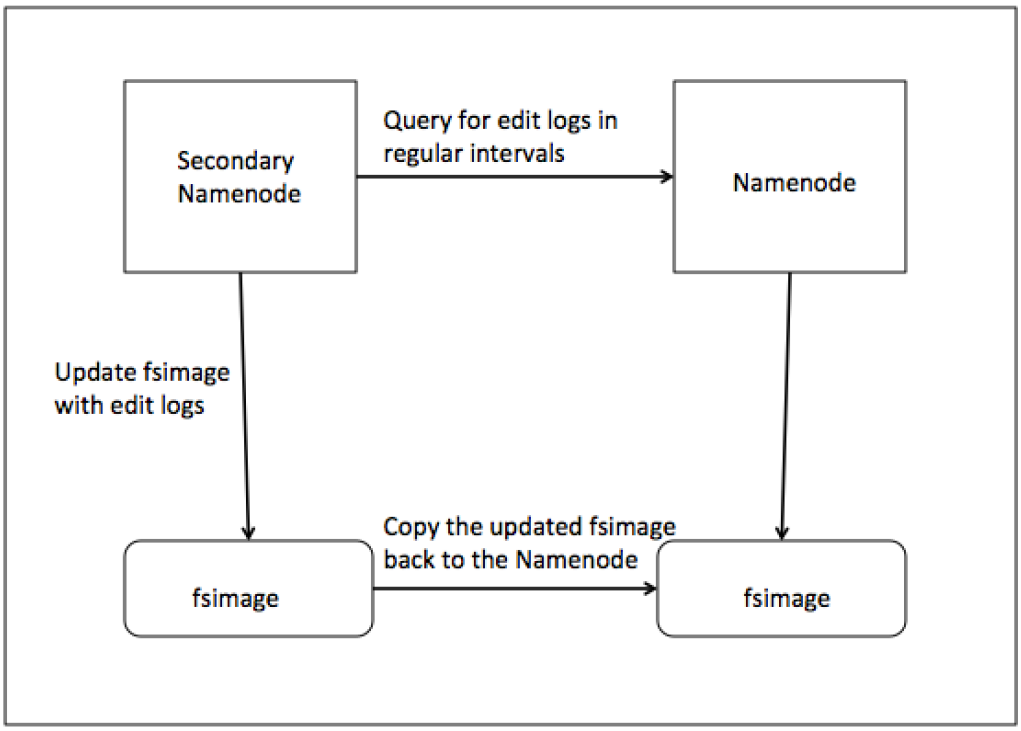


Fig 10: Working of Secondary Namenode

3.5 Advantages of Hadoop

- HDFS component of Hadoop is optimized for high throughput.
- Hadoop is highly scalable.
- Hadoop uses large block sizes, which is useful in manipulating large files. Also amount of metadata in Namenode is minimal.
- Hadoop uses MapReduce framework, which is a batch-based, distributed computing framework, which helps to rapidly process large amounts of data in parallel.
- Hadoop is highly fault-tolerant.
- It provides distributed storage and computing capabilities both.
- Can be deployed on large clusters of cheap commodity hardware as opposed to expensive, specialized parallel processing hardware.
- HDFS component of Hadoop has rock solid data reliability and costs extreme low per byte.

3.6 Disadvantages of Hadoop

- HDFS is inefficient for handling small files and it lacks transparent compression.
- HDFS is not suitable for small files also because there is an overhead in setting up Hadoop environment.
- Hadoop does not offer security model, which is a major concern.
- Hadoop does not provide storage or network level encryption, which is a very big concern for governmental sector application data.
- Master processes of both HDFS and MapReduce components of Hadoop are single points of failure.

Chapter 4
Clojure

Clojure is a dynamically typed, functional programming language that compiles to java bytecode and provides interoperability with java. It is a Lisp based. However, it has some departures from older Lisps. Clojure running on JVM provides portability, stability, performance and security. It also provides access to the wealth of existing java libraries supporting functionality including multithreading, database, GUI, I/O, web applications and more.

4.1 Setting up an environment

Clojure tools –some of the common tools to get started with Clojure include

- Leiningen
- Emacs with CIDER
- Vim with Fireplace
- Eclipse Counterclockwise
- Nightcode

Leiningen is the easiest way to use Clojure. Leiningen is a build automation and dependency management tool for the simple configuration of software projects written in the Clojure programming language. It is a standard tool used by Clojure community. Below steps indicate leiningen installation from the official leiningen webpage. [12]

1. Download the lein script.
2. Place it on $PATH where shell can find it.
3. Set it to be executable. (chmod a+x ~/bin/lein)
4. Run it (lein) and it will download the self-install package.

Leiningen Projects [12]

Leiningen works with projects. A project is a directory containing a group of Clojure (and possibly Java) source files, along with a bit of metadata about them. *project.clj* is a file in the root directory that contains metadata. Through this *project.clj* we can tell leiningen about things like

- Project name
- Project description
- What libraries the project depend on
- What Clojure version to use
- Where to find source files
- What's the main namespace of the app

and more.

Directory Layout of Clojure projects

A Clojure project contains a *src/* directory containing the code, a *test/* directory and a *project.clj* file which describes our project to Leiningen.

project.clj

project.clj file looks something like below

```
(defproject my-stuff "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :main ^:skip-aot my-stuff.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all}})
```

Clojure provides us with a fully dynamic programming environment called the REPL: The (R)ead, (E)valuate, (P)rint, (L)oop. The REPL reads input from the user, executes it and prints the result of the process. During "read" phase source code is converted into a datastructure. During "evaluate" phase this datastructure is first compiled into java byte code and then executed by JVM. Clojure is a compiled language and in many cases can have performance characteristics similar to java.

The REPL becomes the main sketchpad and development tool for many Clojure users. To start a REPL with leiningen, simply type *lein repl* on command line and after a few moments, we see a prompt like this:

```
$ lein repl
user=> _
```

4.2 Clojure basics [26]

Considering basic Clojure code (println "Hello world"). When typed on prompt and hit enter (R)ead phase begins, turning out entered text into a stream of symbols. Provided there are no error these symbols are (E)valuated according to rules, and (P)rinting the result of the code and then (L)oop giving new prompt for input.

```
user=> (println "Hello World")
Hello World
nil
user=> _
```

As we see above nil is outputted which is Clojure equivalent of null, which means println did not produce any computational results and "Hello World" is simply the side effect of executing println.

Considering few more simple examples like below

```
(+ 3 2)
; 5
(+ (+ 1 2) (+ 3 3))
; 9
(+ 1 2 3 4)
; 10
```

As seen above, the operators come first, this is called prefix notation and also it seems the number of parameters or arguments doesn't matter.

Basic data types

Clojure supports numbers, atoms, important atoms, strings, regular expressions, keywords, list, vector, set. Clojure has functions, special forms and macros.

Defining and calling functions

(defn function_name [arguments] expressions) –Syntax of named function. Value of the function is the value of last expression evaluated.

(function_name arguments) -Syntax of a function call. The name of the function being called is the first thing inside the parentheses.

(fn [arguments] expressions) –Syntax of anonymous function.

Built-in functions

Absolutely fundamental functions:

(first sequence) -Returns the first element of a nonempty sequence, or nil if the sequence is empty.
(rest sequence) -Returns a sequence containing all the elements of the given sequence but the first.
(cons value sequence) -Returns a new sequence created by adding the value as the first element of the given sequence.
(empty? Sequence) -Returns true if the list is empty and false otherwise.
(= value1 value2) -Tests whether two values are equal. Works for just anything except functions.

Macros and special forms:

(quote argument) -Returns its argument unevaluated.
(def name expression) -Defines the name to have the value of the expression, in the current namespace.
(if test thenPart elsePart?) -The test is evaluated; if it is a "true" value, the thenPart is evaluated; if it is a "false" value then elsePart is evaluated.
(when test expression ... expression) -If the test evaluates to a true value, the expressions are evaluated and the value of the last expression is returned.
(do exprs)* -Evaluates the expressions in order and returns the value of the last. If no expressions are supplied, returns nil.
(let [name value ... name value] expressions)* -Defines local names, evaluates the corresponding values and binds them to the names.
(throw expression) -The expression must evaluate to some kind of Throwable, which is then thrown.
(recur expressions)* -Performs a recursive call from the tail position, with the exact number of arguments.
(loop [bindings] expressions*)* -Like recur, but the recursion point is at the top of the loop.

4.3 Why Clojure?

As said by the core development team of Clojure, Complexity threatens to overwhelm the modern programmer. Rather than getting things done, it is all too easy to focus on tangential problems caused by the technology itself. Clojure was created to combat this state of affairs through: Simplicity, Empowerment and Focus.

- Simplicity: Clojure is built from ground up to be simple. Code is data. Data is immutable and state is explicit. Functions are easy to write and test.
- Empowerment: Clojure is built on top of JVM. Clojure provides fast access to java code, additionally new ways to use the code better.
- Focus: Clojure provides good level of abstraction. Focus can be directly on the problems rather than tool problems.

Chapter 5

Cascalog

Cascalog is an open source, fully featured data processing and querying library for Clojure or java. The main use cases for Cascalog are processing "Big Data" on top of Hadoop or doing some analysis on local computer. Cascalog is a replacement for tools like Pig, Hive and Cascading, which operates at a significantly higher level of abstraction than those tools. [6]

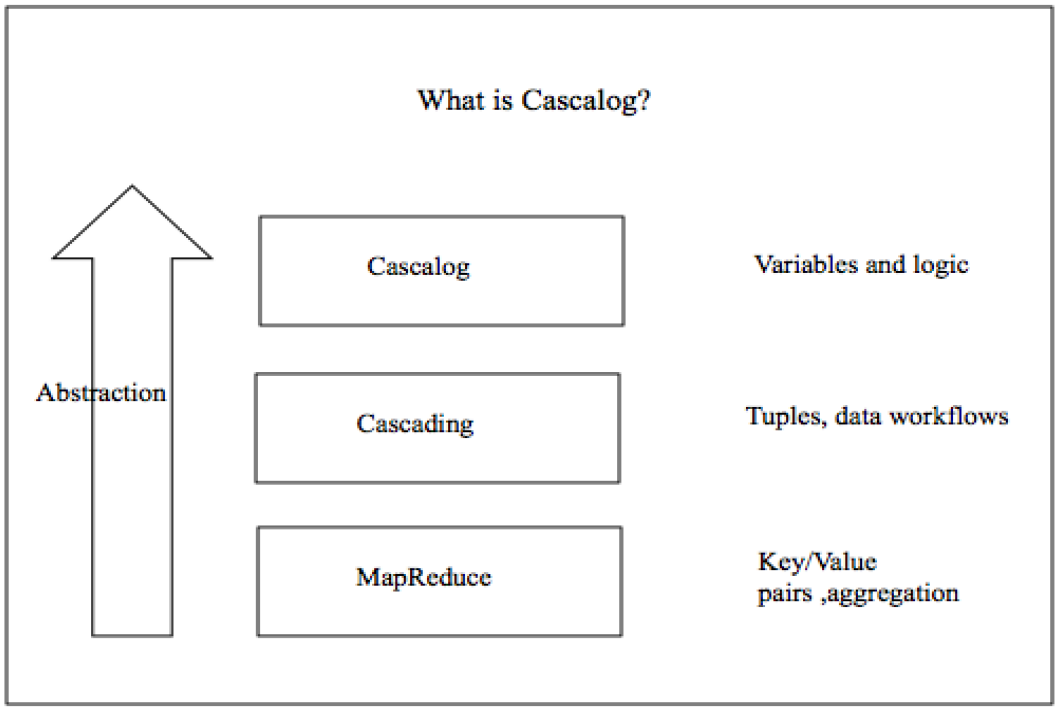


Fig 11: What is Cascalog? [11]

Cascalog is a powerful and easy-to-use data analysis tool for Hadoop. As seen from the figures 11 and 12, it is a declarative query language that is inspired by datalog syntax. Queries are written as regular Clojure code. Queries get compiled to one or more MapReduce tasks through the underlying cascading library. This approach is a big win over writing MapReduce ourselves. Most queries that we run require multiple MapReduce tasks chained together. With Cascalog we write a query declaratively while the underlying libraries take care to create (efficient!) chains of MapReduce tasks. Rather than writing a Mapper, Reducer, Combiner and job configuration for mapreduce jobs, with Cascalog we can write our job in a form of a query and we can write Clojure functions to do whatever we want with our data.
Cascalog can define even complex operations in simple code. Unlike alternatives like Pig or Hive, it's written within a general-purpose language, so there's no need for separate user-defined functions, but it's still a highly structured way of defining queries.

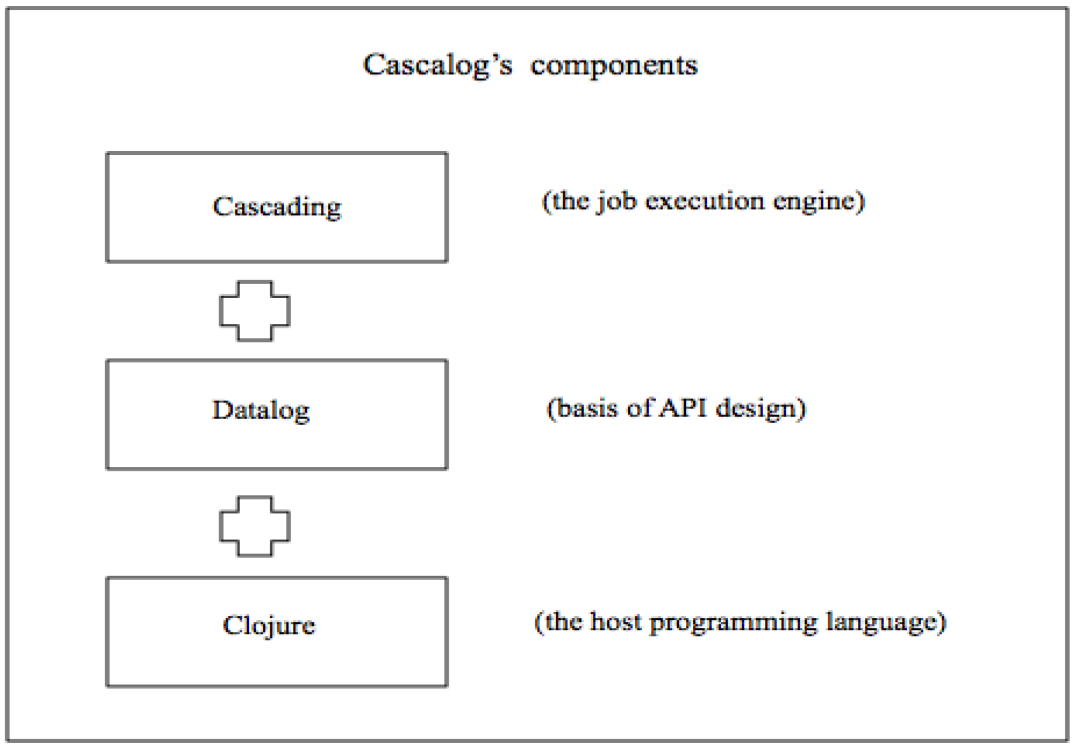


Fig 12: Cascalog's components [11]

5.1 History of Cascalog

The main use cases of MapReduce programs are to parse or analyse huge volumes of data. And since data started growing tremendously, masses of Mapreduce framework came into existence in the past years. A prominent among them was Hadoop. Hadoop is never considered "easy" when it comes to developing. It had and still has a very high learning curve. So this led to the creation of frameworks that provide higher level of abstraction when working with huge amounts of data.

Hadoop is written in java. Writing MapReduce jobs on Hadoop in any language is possible. Also writing MapReduce jobs in Clojure is possible. Writing raw MapReduce a job in any language is difficult because the conceptual model is primitive, complex computations require several MapReduce jobs chained together. That's why data processing tools on top of Hadoop came into existence like Cascading and other alternatives like Pig [45] and Hive [29].

Cascading is used to create and execute complex data processing workflows on top of Hadoop hiding the underlying complexities of MapReduce jobs. It provides data-flow style API using "pipe" metaphor, but the computation steps within pipe is written in java. Nathan Marz, author of Cascalog implemented Datalog on top of Cascading, which was an added layer; the result of this was called Cascalog. Cascalog stands higher on the abstraction ladder implementing Datalog, a truly declarative language on top of existing Cascading library.

Cascalog = Cascading + Datalog

Datalog is a declarative programming language, which is a subset of prolog, which is occasionally used as a database query language. Some interesting differences that differentiate it from prolog are "order of statements in Datalog doesn't matter" and

"all Datalog programs are guaranteed to terminate. " So writing in Datalog is easier than prolog. Cascalog compiles a Datalog-like language into Cascading workflows that can be run on Hadoop MapReduce.

5.2 Cascalog Features: [6]

- Super simple – Same syntax is used for functions, filters and aggregators. Joins are implicit and natural. Fully integrated in a general purpose programming language.
- Expressive - Logical composition is very powerful and arbitrary Clojure code can be run in query with little effort.
- Interactive – Run queries from Clojure REPL.
- Scalable – Cascalog queries run as a series of MapReduce jobs.
- Query anything – HDFS data, database data and/or local data can be queried by making use of cascading's "Tap" abstraction.
- Careful handling of null values - Null values can make life difficult. Cascalog has a feature called "non-nullable variables" that makes dealing with nulls painless.
- First-class interoperability with cascading - Operations defined for Cascalog can be used in a Cascading flow and vice-versa.
- First-class interoperability with Clojure - Can use regular Clojure functions as operations or filters and since Cascalog is a Clojure DSL, can use it in other Clojure code. Full power of Clojure is always available.
- Dynamic queries – Write functions that return queries. Manipulate queries as first-class entities in the language.
- Easy to extend with custom operations – No UDF interface. Just Clojure functions.
- Arbitrary inputs and outputs.
- Use Cascalog side by side with other code.

5.3 Getting started with Clojure Cascalog [6]

The native implementation of Cascalog is in Clojure. But Cascalog also provides a pure java interface called JCascalog, which is perfectly interoperable with Clojure version.

Clojure Cascalog queries run on Clojure REPL. Below steps would get us started with Clojure Cascalog:

- Install leiningen.
  - Since Cascalog programs are Clojure ones, the natural way to manage a Cascalog project is to use Leiningen (lein).
- Require java 1.6 (run java –version)
- Start a new leiningen project with lein new <project name>, replacing <project name>
- Include dependency on Cascalog in the project by adding [cascalog/cascalog-core "2.1.0"] into project's *project.clj* file.
- Get started with Cascalog queries.

5.4 Cascalog Queries [46]

Datasets that are going to be queried are shown below:

```
(def age
      [[“alice” 28] [“bob” 33] [“chris” 40] [“david” 25]  [“Emily”
      25][“george” 31] [“gary” 28] [“kumar” 27] [“Luanne” 36]])

(def person
       [["alice"]["bob"]["chris"] ["david"] ["emily"]["george"]
        ["gary"] ["harold"] ["kumar"]["luanne"]])

(def gender
       [["alice" "f"] ["bob" "m"]["chris" "m"]["david" "m"] ["emily"
       "f"]["george" "m"] ["gary" "m"] ["harold" "m"]["luanne" "f"]])

(def full-names
     [["alice" "Alice Smith"] ["bob" "Bobby John Johnson"]  ["chris"
      "CHRIS"]["david" "A B C D E"]["emily" "EmilyBuchanan"]["george"
      "George Jett"]])

(def location
     [["alice" "usa" "california" nil] ["bob" "canada" nil nil]
      ["chris" "usa" "pennsylvania" "philadelphia"]["david" "usa"
      "california" "san francisco"]["emily" "france"  nil nil]["gary"
      "france" nil "paris"] ["luanne" "italy" nil nil] ])

(def follows
    [["alice" "david"] ["alice" "bob"] ["alice" "emily"]["bob"
      "david"]["bob" "george"]["bob" "luanne"] ["david"
      "alice"] ["david" "luanne"]["emily" "alice"] ["emily"
      "bob"]["emily" "george"] ["emily" "gary"]["george"
      "gary"] ["harold" "bob"] ["luanne" "harold"] ["luanne"
      "gary"]])

(def num-pair
   [[1 2] [0 0] [1 1][4 4][5 10][2 7]
      [3 6][8 64] [8 3][4 0] ])

(def integer
    [[-1][0][1][2] [3] [4][5]
     [6][7][8] [9] ])

(def dirty-follower-counts
   [[2000 "gary" 56] [1100 "george" 124] [1900 "gary" 49]
     [3000 "juliette" 1002] [3002 "juliette" 1010][3001 "juliette"
      1011]])

(def gender-fuzzy
    [["alice" "f" 100] ["alice" "m" 102] ["alice" "f" 110]["bob" "m"
      100]["bob" "m" 101] ["bob" "m" 102]["bob" "f" 103]["chris" "f"
      100]["chris" "m" 200] ["emily" "f" 100] ["george" "m"
      100] ["george" "m" 101] ])
```

Consider the first basic query

```
user  => (?<- (stdout) [?person] (age  ?person  25 ))
```

This query can be read as find all the persons for whom the age of a person is equal to 25. When this query is executed we see logging from Hadoop as the job runs and then results are printed.

5.4.1 Structure of the query

The query operator being used <- is the query creation operator and ?- is the query execution operator. So query operator ?<- both defines and runs a query.

In the first part of the query we tell where the results should be emitted. In the above query it is said (stdout), which means "(stdout)" creates a Cascading tap, which writes its contents to standard output after the query finishes. Any cascading tap can be used for output. This means data can be outputted in any file format like sequence file, text format etc. and anywhere we want (locally, HDFS, database, etc.) The [?person] specified the variable to be printed; it can have many variables though.

Considering few more queries below: [6]

```
user=> (?<- (stdout) [?person] (age ?person ?age) (< ?age 30))
```

This range query tells find all the persons in age dataset who are younger than 30.
Here the age of a person is bounded to a variable ?age and a constraint has been added which says ?age should be less than 30.

```
user=> (?<- (stdout) [?person ?age] (age ?person ?age)(< ?age 30))
```

After defining where the results should be emitted say sink, next result variables of the query are defined in the Clojure vector. In the above query ?age variable has been added with ?person in the result variables, which tells find all the person along with their age from age dataset who are younger than 30.

```
user=> (?<- (stdout) [?person ?a2] (age ?person ?age)
            (< ?age 30) (* 2 ?age :> ?a2))
```

Cascalog has the ability to filter and constrain result variables using predicates. Next in the query are "predicates".
Cascalog has many flavours of functions, filters and aggregators. Three categories of predicates are [7]:

- **Generators:** It is a source of data or tuples. Various operations in the query act on tuples supplied by query's generator. A Cascalog generator appears within a query as a list with generator var, following by a number of output vars equal to the number of generator's tuple fields. A generator with two output fields can be composed in two ways, depending on if you have defined an array that holds the variables you wish to output the generated data to:

```
(<- [?a ?b] (generator :> ?a ?b))
```

or

```
(def output-variables ["?a" "?b"])
   (<- output-variables (generator :>> output-variables))
```

There are 3 types of generators:

1. **Clojure sequences:** These are simplest form of generators and are ideal for testing. For example:

```
(def generator-seq [["a" 1]  ["b" 2]])
    (?<- (stdout) [?a ?b] (generator-seq :> ?a ?b))
```

2. **Existing Queries, defined by <-:** Queries are decomposable. Very complex workflows can be decomposed into multiple subqueries. For example, re-using generator-seq from above:

```
(let [subquery (<- [?a ?b] (generator-seq ?a ?b))]
   (?<- (stdout) [?also-a ?also-b](subquery ?also-a ?also-b)))
```

3. **Cascading Taps:** These process data from a wide range of input sources into tuple format. The hfs-textline function, located in cascalog.api, accepts a path to a file or a directory containing text files and returns a Cascading tap that produces a 1-tuple for each line of text. These source text files can be terabytes in size; Cascalog will take care of parallelization. All that need to be thought are about 1-tuples generated by the tap. For example:

```
(let [text-tap (hfs-textline "/some/textfile.txt")]
   (?<- (stdout) [?textline] (text-tap ?textline)))
```

- **Operations:** Implicit relations that take in input variables defined elsewhere and either act as a function that binds new output variables or a filter that capture or release tuples based on the truthiness of their return values. Examples include (split ?two-chars :> ?first-char ?last-char) and (< ?val 5).

- **Aggregators:** While operations act on one tuple at a time, aggregators act on sequences of tuples. Examples include count, sum, min, max, etc.

A predicate has a name, list of input variables and list of output variables. Predicates in all the above queries are:

- (< ?age 30)

< is a Clojure function and since no output variables are specified, this predicate acts as a filter and filters out records where ?age is less than 30.

- (age ?person ?age)

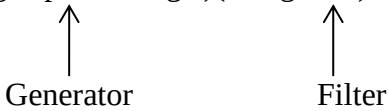
It is an aggregator because age predicate refers to a tap which emits variables ?person and ?age

- (* 2 ?age :> ?a2)

Considering some examples of predicates below:

Example 1:

(?<- (stdout) [?person] (age ?person ?age)(< ?age 30)

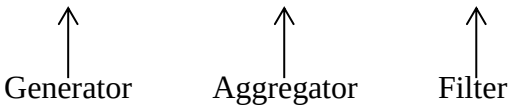


Example 2:

(?<- (stdout) [?person] (full-name ?person ?name) (extract-first-name ?name :> "Leon"))

Example 3:

(?<- (stdout) [?age] (age ?person ?age) (c/count ?count) (> ?count 5))



5.4.2 Predicate operators

Cascalog includes two basic operators for specification of predicate input and output variables :< and :> . And the equivalent versions of these are :<< and :>> which allows specification of sequences of input and output variables.

:< and :>

The :< predicate operator treats the variables on its right as input to the function on its left, as here:
(some-op :< ?a :> ?b)
Similarly, the :> operator, located after the initial function and input vars (if any exist), marks variables to its right as outputs.

Thus, input variables are separated from output variables using keyword :> .
If no keyword :> is specified then:
- Variables are considered as input variables for operations.
- Variables are considered as output variables for generators and aggregators.

For ex: 2 and ?age represent input variables and ?a2 represents output variables in (* 2 ?age :> ?a2)

(> ?age 60 :> ?adult) In this predicate ">" is considered as a function which binds new variable ?adult as boolean variable which says whether ?age is greater than 60. Cascalog is purely declarative. Ordering of predicates doesn't matter.

Considering one example code below :

```
(def digits [[1 2][3 4]])
      (<- [?a ?b ?c](digits ?a ?b)(+ ?a ?b :> ?c))
```

The + function receives ?a and ?b and generates ?c. A more explicit form of this predicate would be:

(+ :< ?a ?b :> ?c)

The :< ushers the variables to its right into the function to its left. This can be left in almost all queries because Cascalog is smart enough to infer where :< should go; at the break between the first function and the following dynamic variables. :> is necessary, as there is no other way to distinguish between inputs and outputs.

:<< and :>>

The :<< and :>> are used to denote sequences of input and output variables. These are equivalent to :< and :> discussed above.
Considering an example, which is equivalent to the code, discussed above:

```
(def my-vars ["?a" "?b"])
      (<- [?a ?b ?c](digits :>> my-vars)(+ :<< my-vars :> ?c))
```

my-vars is bound to a single sequence of Cascalog vars. If we want to use a sequence of vars, we have to explicitly place the :<< or :>>. Cascalog can't infer the difference between :< and :<<.

:#>

The predicate operator, :#>, allows the user to pull specific tuple fields out of a generator by referencing position.

5.4.3 Queries using different cascalog features

Below sections explains complex Cascalog queries which makes use of different Cascalog features: [6]

Non-nullable variables

Cascalog has a feature called non-nullable variables, which handles null variables. Variables that are prefixed with "?" are non-nullable variables and variables prefixed with "!" are nullable variables. Below 2 queries shows the effect of non-nullable variables:

```
user=> (?<- (stdout) [?person ?city] (location ?person _ _ ?city))
```

In the above query Cascalog inserts a null check to filter out any records if a non-nullable variable ?city is binded to null.

```
user=> (?<- (stdout) [?person !city] (location ?person _ _ !city))
```

This query produces some null values in result set.

Variables and constant substitution

Variables are symbols that begin with ? or !. Symbol _ is used to ignore the variable when we don't care about the value of the output variable. In all other cases it is evaluated and inserted as a constant within a query. This feature is called "constant substitution". For example

```
(* 4 ?v :> 100 )
```

Using a constant as output variable acts as a filter on the results of the function. In the above example 4 and 100 are constants. Output variable 100 acts as a filter to keep only the values of ?v which produces 100 when multiplied by 4. Strings, numbers, other primitives and any objects that have Hadoop serializers registered can be used as constants.

Aggregators

As mentioned earlier, aggregators are a type of predicates and operate on series of tuples. Let's consider below example that finds number of people less than 30 years old.

```
(?<- (stdout) [?count] (age _ ?a) (< ?a 30) (c/count ?count))
```

In this example there is a single value about all of the records. Aggregation can also be done over partition of records.

```
(?<- (stdout) [?person ?count] (follows ?person _) (c/count ?count))
```

In this example ?person is included in the output variable. So Cascalog will partition the records by ?person and apply c/count aggregator within each partition.

Multiple aggregators can also be used.

```
(?<- (stdout) [?country ?avg]
   (location ?person ?country _ _) (age ?person ?age)
   (c/count ?count) (c/sum ?age :> ?sum)
   (div ?sum ?count :> ?avg))
```

This query gets the average age of people living in a country. In the above example we have 2 aggregators c/count and c/sum. "div" operation is applied after both aggregators run, because it depends on the aggregator output variables.

Custom operations

Queries can use custom operations like

defmapop: Defines a custom operation which adds fields to a tuple. Expects a single tuple to be returned.

deffilterop: Defines a custom operation which only keeps tuples for which this operation returns true.

defmapcatop: Defines a custom operation which creates multiple tuples for a given input.

Defaggregateop: Defines an aggregator.

```
user=> (defmapcatop split [sentence]
       (seq (.split sentence "\\s+")))

user=> (?<- (stdout) [?word ?count] (sentence ?s)
            (split ?s :> ?word) (c/count ?count))
```

The above query gets the number of times a word appears in the set of sentences. But in the above query the same word will be counted differently if it appears as different combinations of uppercase and lowercase letters.

Below query is a fix for the above.

```
user=> (defn lowercase [w] (.toLowerCase w))

user=> (?<- (stdout) [?word ?count]
            (sentence ?s) (split ?s :> ?word1)
            (lowercase ?word1 :> ?word) (c/count ?count))
```

Here we see the regular Clojure functions can be used as operations. When no output variables are given, Clojure functions acts as filters. If output variables are given, then it acts as a map.

Subqueries

Complex queries can use subqueries. Let form can be used to define subqueries. Subqueries can be defined using <- , the query definition operator. For example

```
user=> (let [many-follows (<- [?person] (follows ?person _)
                              (c/count ?c) (> ?c 2))]
         (?<- (stdout) [?person1 ?person2] (many-follows ?person1)
          (many-follows ?person2) (follows ?person1 ?person2)))
```

In this example, subquery *many-follows* is defined using the *let* form. *many-follows* subquery can be used within the query executed in the body of the *let* form.

Cascalog can use multiple subqueries which produce multiple outputs. In the below example both subqueries many-follows and active-follows are defined without executing them. The query execution operator ?- is used to bind each query to a tap. ?- executes both queries in tandem.

```
user=> (let [many-follows (<- [?person] (follows ?person _)
                              (c/count ?c) (> ?c 2))
                         active-follows (<- [?p1 ?p2] (many-follows
                          ?p1)(many-follows ?p2) (follows ?p1 ?p2))]
            (?- (stdout) many-follows (stdout) active-follows))
```

Duplicate elimination

If there are no aggregators in the query, then Cascalog tries to insert a reduce step to remove duplicates from the output. This behaviour can be controlled using *:distinct* predicate. Comparing the below queries:

```
user=> (?<- (stdout) [?a] (age _ ?a))
user=> (?<- (stdout) [?a] (age _ ?a) (:distinct false))
```

The second query will have duplicates in the output. The use case of this functionality is when the subquery needs to do some preprocessing on the input source.

Sorting

Cascalog has *:sort* and *:reverse* predicates that control the order in which tuples arrive at the aggregator. By default, tuples arrive at the aggregator in some arbitrary order. For example, to find the youngest person each person follows:

```
user=> (defbufferop first-tuple [tuples] (take 1 tuples))
user=> (?<- (stdout) [?person ?youngest] (follows ?person ?p2)
              (age ?p2 ?age) (:sort ?age) (first-tuple ?p2 :>
             ?youngest))
```

To find the oldest person that each person follows, *:reverse* is used together with *:sort* predicate:

```
user=> (?<- (stdout) [?person ?oldest] (follows ?person ?p2)
              (age ?p2 ?age) (:sort ?age) (:reverse true)
              (first-tuple ?p2 :> ?oldest))
```

Implicit equality constraints

Considering some examples which are self-explanatory for this feature :

```
user=> (?<- (stdout) [?n] (integer ?n) (* ?n ?n :> ?n))
```

integer is one of the dataset mentioned in the earlier section 5.4 of this chapter. It contains a set of numbers. The above query outputs all the numbers that are equal to themselves when squared. Here Cascalog sees that ?n is rebinded and automatically filters out remaining ?n where output of the predicate * is not equal to input.

```
user=> (?<- (stdout) [?n] (num-pair ?n ?n))
```

In this example Cascalog detects whether both numbers are same in *num-pair* dataset.

```
user=> (?<- (stdout) [?n1 ?n2](num-pair ?n1 ?n2) (* 2 ?n1 :> ?n2))
```

This example outputs pair of numbers where second number is twice the first number.

Outer joins

Multiple sources of data can be joined in Cascalog using the same variable name in the multiple sources of data. For example given two datasets "*age*" and "*gender*" the age and gender of a person can be retrieved using below query:

```
user=> (?<- (stdout) [?person ?age ?gender]
               (age ?person ?age) (gender ?person ?gender))
```

There is an inner join in the above example. The result consists of only people appearing in both datasets.

```
user=> (?<- (stdout) [?person !!age !!gender]
             (age ?person !!age) (gender ?person !!gender))
```

This is an example query for full outer join. This consists of null values for people with non-existence ages or gender. Cascalog's outer joins can be triggered by variables that begin with "!!". These are called "ungrounding variables". A predicate that contains an ungrounding variable is called an "unground predicate" and a predicate that does not contain a grounding variable is called a "ground predicate".

Joining together two unground predicates results in a full outer join, while joining a ground predicate to an unground predicate results in a left join.

```
user=> (?<- (stdout) [?person1 !!person2]
             (person ?person1) (follows ?person1 !!person2))
```

The above query is an example of left join. It gets the *follows* relationships for each person. And if no *follows* relationship exists for a person, it produces null values.

5.5 How Cascalog executes a query

Cascalog query is defined by a list of output vars which are constrained by a list of predicates. Predicates can be considered as facts about data. Ordering of predicates doesn't matter.

Consider below example to understand how Cascalog executes a query. This query outputs words from a sentence that occur more than 5 times.

```
(<- [?word]
        (sentence ?sentence)
        (split ?sentence :> ?word)
        (c/count ?count)
        (>  ?count 5))
```

Cascalog queries are executed in 3 steps:

1. Pre-aggregation
2. Aggregation
3. Post-aggregation

In step 1 Cascalog joins all aggregators while applying as many functions and filters as it can in the process.

In the above example query, Cascalog starts with the generator sentence. Since all the input vars are satisfied for a split, it then takes up the split which produces another var ?word. Now (> ?count 5) cannot be applied because ?count var is not satisfied. Since there are no operations to apply, it moves to aggregation phase.

In step 2 Cascalog partitions the tuples by any output variables that have already been satisfied. If no variables are satisfied, then it makes a single global partition containing all tuples. Then executes aggregators of the query on each partition.

In the above example the only output variable ?word is already satisfied, Cascalog applies the count aggregator to create a ?count var for every value of ?word.

In step 3 Cascalog executes the remaining functions/filters which are dependent on aggregator output.

In the above example, now ?count variable is satisfied, so predicate (> ?count 5) can now be applied.

5.6 Running Cascalog

Local mode –Cascalog can be run from REPL on our local machine. In this case Hadoop runs in local mode which means it is completely in process.

Running on a production cluster –Cascalog queries can be run on external single node or multinode Hadoop cluster.
Sample data needs to be copied onto the cluster. Next, run *lein uberjar* to create a jar containing the program with all its dependencies. To run the query on a cluster and output the results in text format to */tmp/results*, run: (assuming sample data is copied to */tmp/follows* and *tmp/action*)

```
hadoop jar [prjectname-standalone.jar] /tmp/follows /tmp/action
/tmp/results
```

Since *uberjar* has Clojure within it, queries can be run from REPL using below commands:

```
hadoop jar [prjectname-standalone.jar] clojure.lang.Repl
user=> (use 'cascalog-demo.demo) (use 'cascalog.api)
Followed by queries on REPL .
```

All the custom operations have to be compiled into *uberjar* before running on production cluster. But this is not the case in Hadoop local mode, where REPL is still great for development of custom operations.

5.7 Cascalog users and their opinions: [8]

- Factual – It was launched in October 2009 is an open data platform which is relying more and more on the Hadoop stack of technologies. It aggregates and processes growing sets of data. Factual is using Cascalog to run machine learning algorithms on billions of web pages and user contributed data to aggregate factual data present in multiple sources. Factual says Cascalog has allowed easy abstraction from details of data sources (with taps, as in cascading).They also benefit from the ad-hoc nature of Cascalog when doing things such as generating statistics across our datasets, verifying map-reduce job outputs, tracing the history of data through our processing pipeline and running experimental data manipulation and transformations.

  Cascalog -We're also benefiting from the availability of Clojure in Cascalog. Clojure is a natural fit when doing custom data manipulations and it's also quite useful to use the REPL to experiment. Being able to "call out" to pure Clojure from our Cascalog queries has been a big win.

- Harvard school of health –It uses Cascalog for processing large large data sets of sequencing data. They need approaches that scale to increasing amounts of data also such approaches should facilitate rapid iterations of coding and testing for algorithm development work. They say they need to be as efficient as possible, because any development code could potentially become part of processing pipelines.

  Cascalog -It made coding for Hadoop much easier. It allows them to focus on the queries and data interpretation. It additionally increases the understandability of the code, which is essential for reproducibility and transparency.

- Intent Media –It helps retailers recognize and react to the unique value of each site visitor by providing predictive analytics. They analyse terabytes of data efficiently using Cascalog to help retailers make smart, real-time choices about who sees what and how to adapt their site to best realize the full value of each visitor.

Cascalog –It is an increasingly core component of our backend modelling pipeline comprising data aggregation, pre-processing and feature extraction.

- Lumosity -It is pioneered in understanding and enhancement of human brain to give each person the power to unlock their full potential. Data analysis is an important part of their business, whether it's to conduct new scientific studies to learn more about the human brain or analyse user behaviour.

Cascalog -It allows our Research & Development team to efficiently analyse our database of human cognitive performance – the largest in the world with over 450 million data points - to gain new insights on cognitive training.

- Twitter –It uses Cascalog. A batch workflow written using Clojure and Cascalog updates a variety of !ElephantDB views a few times a day. These views include time series aggregations, influence analysis, follower distribution analysis and more. Additionally, the dataset greater than 40TB is vertically partitioned in a few different ways to allow for efficient querying later on using Cascalog. Twitter says Cascalog's conciseness and great expressive capabilities greatly reduce the complexity in their batch processing.

Cascalog –It is also used for ad-hoc querying and exploratory work, taking advantage of the ease of defining and running queries from the REPL. When a major event happens, we extract relevant tweets from the master datastore to a local computer where they can be analysed in a quick iterative fashion.

- Yieldbot's -Cascalog forms the core of intent modelling and matching technology stack. Publisher's data is fed through a batch workflow at regular intervals and performs a wide array of task such as predictive modelling, text processing and metrics aggregation.

Cascalog and Clojure allow us to develop, deploy, explore and iterate on our workflows with extreme speed and minimal effort.

- REDD Metrics –It uses Cascalog at the heart of their large-scale deforestation monitoring system, currently housed at the Center for Global Development in Washington. They process hundreds of gigabytes of NASA satellite data down into concrete predictions on the likelihood that some piece of land will be deforested in the next month.

Cascalog –It allows us to generate timeseries and perform analysis at a scale unimaginable with current "state of the art" practices.

- uSwitch –It uses high-level data to make business decisions and drill down to the microscopic-level to enable a personalised experience to each of their customers. Cascalog sits at the heart of their modular data pipeline transforming immutable event data to clean and extract customer features for the rest of the business. Furthermore, the logical and functional nature of Cascalog enables their small data team to build simple, composable data processing workflow on scale.

5.8 Why Cascalog?

There are a number of different technologies that use Hadoop as their processing/execution environment for querying and processing big data. Then comes the question why Cascalog?

There are two categories of work we may do with data: querying and transforming. Some tools are best suited for querying like SQL and few others are best suited for transforming. Cascalog is the best approach for both for a number of reasons. Firstly, Cascalog is a Clojure DSL. So manipulating data with Clojure is easy compared to other general-purpose languages (like Java, Python, Ruby etc.). Since it is on JVM it can leverage on any existing library. Its language is homoiconic, which means the source code is made of same data structures as that we use for data. And also same functions that are used for manipulating data can be used for manipulating source code at compile time.

In languages like Pig [45] and Hive [29] in order to make complex manipulations of data, User Defined Functions (UDF) is to be written which is a great way to extend basic functionality. But in Pig and Hive, UDF have to be written in different language because, the basic PigLatin of Pig and SQL of Hive have only handful of functions and lack basic control structures. Pig and Hive allow writing UDF in different languages like Java, Python or few others. But this requires developers to switch between programming paradigms. We start with a language like PigLatin and SQL and then end up writing, compiling and bundling UDF in another different language. Unlike these languages Cascalog is Clojure DSL, so main language is Clojure, functions are written in Clojure, data is represented as Clojure data types and runtime is JVM –all the available libraries in the JVM ecosystem can be used, no switch between the languages is required, no additional compilation is required, no additional installation burden.

Why another query language like Cascalog for Hadoop? Because existing tools cause too much accidental complexity.
Accidental complexity –complexity caused by the tool used to solve a problem rather than a problem itself. [11] Distinct query languages cause accidental complexity. Accidental complexity caused by other Hadoop tools like Pig and Hive is: query language is different than the programming language as explained above.
Table 2 shows the comparison when the query language is different from programming language.

Other tools	**Cascalog**
Friction when embedding custom operations.	Custom operations defined just like any other functions.
Interlacing queries with regular application logic is unnatural.	Interlacing queries with regular application logic is trivial.

Generating queries dynamically is difficult.	Generating queries dynamically is easy and idiomatic.

Table 2: Cascalog versus other tools [11]

With Cascalog, query logic can be customized –including text parsing – using the full power of Clojure. So whenever Cascalog query needs to manipulate data or text in some non-trivial fashion, a full programming language is available. There isn't an artificial barrier between programming language and query language.

Pig is one option for extract transform and load of log files and on big data but it is by no means the only option available. Others include Fluentd, Flume, Scalding and Cascading. Most of the other tools also have or use Hadoop as the processing/ execution environment. Like Pig, Cascalog takes care of building the low-level Hadoop job artefacts but offers a higher-level data abstraction. However, Cascalog uses Cascading for managing its interface with Hadoop. Cascalog has the ability to abstract and ability to compose in higher levels compared to other tools. It is based on logic programming and rule interface, we can and it is encouraged to write sub-queries and reuse them. So complex processing can be written by composing sub-queries. Finally most important feature REPL (Read, Eval, Print Loop) enables to explore data in a more interactive way. One more reason worth using Cascalog is it is suitable for either batch process or a real-time streaming solution.

Cascading describes itself on its home page as: Cascading is an application framework for Java developers to quickly and easily develop robust Data Analytics and Data Management applications on Apache Hadoop.
It is also worth emphasising the Full power of Clojure always-available differentiator: a Cascalog program is a Clojure program and the former can use the functions and facilities of the latter in a very natural way. Contrast this flexibility with Pig Latin and its user-defined functions where the API between the former and latter is defined and constrained.

5.9 Advantages of Cascalog

- Using Cascalog to write Hadoop jobs in Clojure - Even though only plain Clojure can be used, Cascalog gives the ability to write MapReduce jobs in a fast and concise way.
- The big bonus that comes from using Cascalog is the ease of testing.
- Cascalog stands higher on the abstraction ladder implementing Datalog, a truly declarative language on top of existing Cascading library.
- Unlike many other libraries and frameworks that increase the abstraction level while dropping the ability to go one level deeper, Cascalog gives the ability to actually resort to writing Cascading when required.
- There is an increase in expressiveness and performance of Cascalog compared to other Hadoop tools without compromising its simplicity or flexibility.
- Cascalog has added benefits to being Clojure DSL like –Excellent module system, Interactive REPL; make use of any Clojure functions in queries.
- Can build queries in an expressive and composable way as one would expect with a Clojure library and get scalability for free.

Chapter 6
Experimenting with Cascalog on sensor data

In today's world there are millions of devices and soon may be billions that are connected to the internet - cars, medical equipment, cell phones, buildings, meters, power grids, automobiles with built-in sensors and many more. These connected devices comprise the Internet of Things. [19] The Internet of Things is generating an unfathomable amount of sensor data -product manufacturers need to manage and analyse the data to build better products, predict failures to reduce costs and to improve customer satisfaction. According to Gartner, there will be nearly 26 billion devices on the Internet of Things by 2020. [19] Sensors are used in different sectors namely environmental monitoring, infrastructure management, industrial applications, energy management, medical and healthcare systems, building and home automation, transport systems and large scale deployments. Sensors are being used for monitoring movements, measuring the environment and many other purposes without the direct human interventions. Sensors are the future of distributed data. [20]

6.1 Challenges with sensor data

There are differences between sensor based data sources and traditional database sources, which make standard query processors poorly suited for querying sensor data as explained in section 2.5.
Sensors produce data continuously at regular intervals, because they use push model. They deliver data in streams. Sensor data should be processed or queried in real time as the data arrives, because it would be very expensive to store raw sensor data on disk, or mostly all the sensor streams represent real world events like traffic accidents, CCTV footages in banks and few others which need to be responded in time.
Sensors do not deliver data in reliable rates, data is often garbled. Limited processor and battery resources are the constraints.

In section 6.2 we see how sensor data is queried using Cascalog, which uses Hadoop as processing/execution environment.

6.2 Analysing set of Cascalog queries

In this experiment we use sensor data provided by Siemens. Data consists of events and measurements and they both correlate. Three different datasets which we call Dataset1, Dataset2 and Dataset3 are used in the experiment. These three datasets have a different amount of sensors, a different amount of tuples for the measurement table and event table accordingly and a different time range in each case. Comparing the sizes of these datasets - Dataset1<Dataset2<Dataset3. Below is a detailed description of how the three datasets look like.

Dataset1

Sample of measurements table:

Timestamp	assembly	sensor	value
12.05.2010 00:00:00	GasTurbine2103/01	TC1	251.3

This table has 6 sensors with a time range of 1 month and 25.921 tuples.

Sample of events table:

Timestamp	assembly	category	eventtext	downtime	tag	process	state
11.05.2010 23:57:50	GasTurbine2103/01	Start Inhibit	Flame On	03:37:43	sti 1033	FALSCH	No Data

This table has 2093 tuples with a time range of 1 month.

Dataset2

Sample of measurements table:

Timestamp	assembly	sensor	value
2005-07-06 16:27:38	Turbine2	Static Pressure	1.000

This table has 420.001 tuples.

Sample of events table:

Timestamp	assembly	category	eventtext	downtime	tag	process	state
2012-09-01 10:20:00	Turbine1	Event from the CU	Event642	-	5	f	5373

This table has 182974 tuples.

Dataset3

This dataset consists of timestamp and the values of different sensors during that timestamp. This dataset is the largest of the three datasets.

This section describes a few important queries on sensor data. First query is explained in detail and all the rest are the shorthand of concrete implementations.

Listing 4.1: Receive all sensor events with “Warning” category

```
(ns Cascalogproject.core
    (:use [Cascalog.api]
      [cascalog.more-taps :only (hfs-delimited)])
(:gen-class))
(def info
     (hfs-delimited  "/Users/Data/Messages.txt"
                      :delimiter ";"
                      :outfields ["?timestamp" "?assembly"
"?category" "?eventtext" "?downtime"  "?tag" "?process" "?state"]
                      :skip-header? false))

(def info-tap
 (<- [?timestamp ?assembly ?category ]
     ((select-fields info ["?timestamp" "?assembly" "?category"])
?timestamp ?assembly ?category )))

(?<- (stdout)
    [?timestamp ?assembly ?category ]
    (info-tap :> ?timestamp ?assembly ?category )
    (clojure.string/trim ?category :> ?trimmed-category)
    (= ?trimmed-category "Warning"))
```

This query is run on Dataset1. First we need to include the dependencies on Cascalog, Clojure and any other essential dependencies in *project.clj* file. And then import a number of namespaces from these libraries into our script or REPL.

Cascading provides a number of taps -sources of data or sinks to send data to, including one for CSV and other delimited data formats. Cascalog also has some very nice wrappers for several of these taps, but not for the CSV one.
hfs-delimited in the query above is used to read delimited file with a delimiter like quote characters, comma or any other.

The default separator is a tab character, so the standard *hfs-delimited* tap with no options would produce one tuple for each line of text:
(hfs-delimited "/path/to/file")
;; makes textlines

The "*:delimiter*" option allows you to change this:

```
(hfs-delimited "/pathto/data"
         :delimiter ";")
```

;; produces 8-tuples, all strings

The problem of the header line getting in the way can be solved using *:skip-header?*

```
(hfs-delimited "/pathto/data"
               :delimiter ","
               :skip-header? true)
```

;; produces 8-tuples of strings.

Next, if we include a vector of classes with the *:classes* keyword, the tap will do class conversions on the fields:

```
(hfs-delimited "/pathto/data"
              :delimiter ","
             :classes [String String String Float Float Float Integer
                       Float]
             :skip-header? true)
```

;; produces 8-tuples with the above classes -- numbers are parsed properly, strings stay strings.

The ability to select out specific fields by name can be done using *:outfields*

```
(def info-tap
 (<- [?timestamp ?assembly ?category ]
     ((select-fields  info  ["?timestamp"  "?assembly"  "?category"])
      ?timestamp ?assembly ?category )))
```

;; returns 3-tuples

And the last section is the Cascalog query which retrieves all the events with "Warning" category together with the time of their occurrence and assembly causing it. The query structure is explained in the previous chapter in section 5.4.1. Clojure.string/trim is a Clojure API to remove whitespaces from both ends of a String.

Listing 4.2: Retrieve all the reasons for start failure and also time it occurred

```
(?<- (stdout)
    [?timestamp  ?category ?eventtext ]
    (info-tap :> ?timestamp  ?category ?eventtext)
    (clojure.string/trim ?category :> ?trimmed-category)
    (= ?trimmed-category "Start Failure"))
```

This query is run on Dataset1. The query includes ?eventtext in the output vars which gives the reason for start failure together with the time of their occurrence.

Listing 4.3: Union of "Start Inhibit" and "Start Failure" events

```
(let [subquery1 (<- [?timestamp  ?category ]
(info-tap :> ?timestamp  ?category )
(clojure.string/trim ?category :> ?trimmed-category)
(= ?trimmed-category "Start Failure"))
subquery2 (<- [?timestamp  ?category ]
(info-tap :> ?timestamp  ?category )
(clojure.string/trim ?category :> ?trimmed-category)
(= ?trimmed-category "Start Inhibit")) ]
(?- (stdout) (union query1 query2)))
```

This query is run on Dataset1. *subquery1* retrieves events with "Start Failure" category and *subquery2* retrieves events with "Start Inhibit" category. And then using union of Cascalog.api the tuples from the subqueries are merged together into a single subquery. Uniqueness of tuples is handled by union api. *subquery1*, *subquery2* and the query using union are bound in *let*. *let* is a Clojure special form which evaluates the expression in a lexical context.

Listing 4.4: Retrieve all the events where sensor values are greater than 800.0

```
(?<- (stdout)[?timestamp ?assembly ?value ]
     (info-tap :> ?timestamp ?assembly ?value )
     (> ?value 800.0))
```

This query is run on Dataset1. This is similar to previous queries except Clojure.string/trim not being used, since ?value is a float value and string trimming cannot be done on it. And > of Clojure.core checks if the numbers are in monotonically decreasing order. Also in *info* function *:classes* should be used in order to parse float and strings correctly.

Listing 4.5: Count of events where sensor values reached greater than 800.0

```
(?<- (stdout)[ ?count ?value]
     (info-tap :> ?value )
     (> ?value 800.0)(c/count ?count))
```

This query is run on Dataset1. Cascalog provides various helper operations in Cascalog.ops namespace. We need to include (:require [cascalog.ops :as c]) within namespace declaration. Count is one such built in operation present in Cascalog.ops. Cascalog makes a partition the tuples by any output variables that are satisfied, in this only one variable ?value. Applying the filter to filter out ?value greater than 800.0 and then making a count of them. Similar queries can be made to find the number of times for start failure categories.

Listing 4.6: Highest value of sensor

```
(?<- (stdout) [?timestamp-out ?value-out]
     (info-tap ?timestamp ?value)
     (:sort ?value) (:reverse true)
     (c/limit [1] ?timestamp ?value :> ?timestamp-out ?value-out))
```

This query is run on Dataset1. *:sort* and *:reverse* predicates are used to control the order in which the tuples arrive at an aggregator. By default aggregators receive tuples in some arbitrary order. Using only *:sort* predicate produces results in increasing order of values. Together with *:reverse* predicate the order is reversed means descending order of output. *limit* is a built in operator of Cascalog used to return the top-n tuples. In above query 1 is used with limit. So only the highest value will be displayed.

Also *first-n* can be used as below. *first-n* accepts a generator and a number `n` and returns a subquery that produces the first n elements from the supplied generator.

```
(def info-tap
    (<- [ ?timestamp ?value ]
    ((select-fields info [ "?timestamp" "?value"]) ?timestamp ?value
      )))
(?<- (stdout) [?timestamp ?value]  (info-tap ?timestamp ?value)
      ((c/first-n info-tap 1 :sort["?value"] :reverse true)?timestamp
      ?value))
```

Other method mentioned below produces top 4 sensor values.

```
(defbufferop first-tuples [tuples](take 4 tuples))

(?<- (stdout) [?timestamp-out ?smallest](info-tap ?timestamp ?value)
      (:sort ?value)(:reverse true)(first-tuples ?timestamp ?value :>
      ?timestamp-out ?smallest))
```

Similar to above queries, lowest value of sensor can be obtained just by removing *:reverse* predicate with :sort. It is also a must to include (:require [cascalog.ops :as c]) within the namespace declaration since limit, first-n are present in that.

Listing 4.7: Sensor values between timestamp 12:05:2010 10:00:00 and 12:05:2010 11:00:00

```
(ns Cascalogproject.core
(:use [cascalog.api]
       [cascalog.more-taps :only (hfs-delimited)])
(:require [clj-time.core :as t])
(:require [clj-time.format :as f])
(:require [cascalog.ops :as c])
(:require [clj-time.coerce :as ct])
(:gen-class))

(def info
     (hfs-delimited  "/Volumes/user/burner.txt"
                      :delimiter ";"
                      :outfields ["?timestamp" "?assembly" "?sensor"
                        "?value"]
                      :classes[String String String Float]
                      :skip-header? false))

(def info-tap
 (<- [?timestamp ?value]
     ((select-fields info ["?timestamp" "?value"]) ?timestamp
      ?value)))
```

```
  (def datefrom "12:05:2010 10:00:00")
  (def custom-formatter (f/formatter "dd:MM:yyyy HH:mm:ss"))
  (def start-value (ct/to-long (f/parse custom-formatter datefrom)))

  (def dateto "12:05:2010 11:00:00")
  (def end-value (ct/to-long (f/parse custom-formatter dateto)))

  (defn convert-to-long [a]
      (ct/to-long (f/parse custom-formatter a)))

 (?<- (stdout)[?timestamp ?value](info-tap ?timestamp ?value)
   (convert-to-long ?timestamp :> ?converted)
   (>= ?converted start-value)
   (<= ?converted end-value))
```

This query is run on Dataset1. To deal with timestamps *clj-time,* a date and time library for Clojure is used. Dependency for *clj-time* library should be included in the *project.clj* file. And then all the necessary namespaces need to be imported from the library in the namespace declaration section. Function i*nfo* is the source of data and also uses appropriate options. *datefrom* and *dateto* are the start and end timestamps, which represents a range and are created using d*ef*. If we need to parse or print date-times, we use *clj-time.format:*

```
(:require [clj-time.format :as f])
```

Parsing and printing are controlled by formatters. We can either use one of the built in ISO8601 formatters or define our own formatter. As seen in code snippet above, we are defining our own *custom-formatter,* which uses date-time format "dd:MM:yyyy HH:mm:ss" in which mm is minutes, MM is months, ss is seconds.
to-long is used to convert a Joda DateTime to and from a Java long, so that comparing dates would become easy. And in the query we check for every timestamp from the datasource, if it lies between the *start-value* and *end-value*. The tuples satisfying those filters are displayed with the sensor values.

Listing 4.8: What is the highest sensor value between specific time period

```
(?<- (stdout)[?timestamp-out ?value-out](info-tap ?timestamp ?value)
     (convert-to-long ?timestamp :> ?converted)
     (:sort ?value) (:reverse true)
     (> ?converted start-value)
     (< ?converted end-value)
 (c/limit [1] ?timestamp ?value :> ?timestamp-out ?value-out))
```

This query is run on Dataset1. Same as the query in Listing 4.7, but sorting is performed on sensor values here to find the highest value between given timestamps.

Listing 4.9: Retrieve all sensor values for the last 4 years from now

```
(def multi-parser (f/formatter (t/default-time-zone)  "dd:MM:YYYY
                  HH:mm:ss" "YYYY-MM-dd HH:mm:ss"))
      (def currenttimestamp
             (f/unparse multi-parser (f/parse multi-
      parser  (l/format-local-time (l/local-now)  :mysql))) )
      (def custom-formatter (f/formatter "dd:MM:yyyy HH:mm:ss"))
      (def subtractedtime
```

```
       (l/format-local-time  (t/minus  (f/parse custom-formatter
      currenttimestamp) (t/months 48) ):mysql))
      (def required-timestamp (f/unparse multi-parser (f/parse multi-
      parser subtractedtime)))

(defn convert-to-long [a]
      (ct/to-long (f/parse custom-formatter a)))

(?<- (stdout)[?timestamp ?value](info-tap ?timestamp ?value)
     (convert-to-long ?timestamp :> ?converted)
     (> ?converted required-timestamp))
```

This query is run on Dataset1. *multi-parser* and *minus* are used in addition to previous queries. *multi-parser* is present in namespace *clj-time.format*, which is used to parse dates in multiple formats and format dates in just one format. *minus* is present in namespace *clj-time.core* which subtracts some amount of time from the given time. In the above code snippet 48 months to say 4 years is subtracted from the current timestamp to get the *required-timestamp*.

Neither Hadoop nor Cascalog are built to work on smaller data sets, it is important to conduct the experiment on data of big sizes. The experiment is extended to work on sensor data bigger in size compared to Dataset1. Below are some complex queries, which run on Dataset2 and Dataset3.

Listing 4.10: Collect all messages related to potential start problems, concerning some time period in the past.

```
(let [query1 (<- [?timestamp  ?category_description ?eventtext]
    (info-tap :> ?timestamp  ?category_description ?eventtext)
    (convert-to-long ?timestamp :> ?converted-timestamp)
    (>= ?converted-timestamp needed-timestamp)
    (= ?eventtext "Start initiated"))
    query2 (<- [?timestamp  ?category_description ?eventtext]
    (info-tap :> ?timestamp  ?category_description ?eventtext)
    (convert-to-long ?timestamp :> ?converted-timestamp)
    (>= ?converted-timestamp needed-timestamp)
    (= ?category_description "Event from the CU which indicates
automatic prevention from a start attempt due to some fault"))
    query3 (<- [?timestamp  ?category_description ?eventtext]
    (info-tap :> ?timestamp  ?category_description ?eventtext)
    (convert-to-long ?timestamp :> ?converted-timestamp)
    (>= ?converted-timestamp needed-timestamp)
    (= ?category_description "Event from the CU  which
indicates  automatic prevention from a start attempt")) ]
 (?- (stdout) (union query1 query2 query3)))
```

This query is run on Dataset2.

Listing 4.11: Get a sorted list of event text and their number of occurences.

```
(defn global-sort [sq fields]
 (let [out-fields (get-out-fields sq)
       new-out-fields (v/gen-nullable-vars (count out-fields))]
   (<- new-out-fields
      (sq :>> out-fields)
      (:sort :<< fields)(:reverse true)
      ((IdentityBuffer.) :<< out-fields :>> new-out-fields))))
```

```
 (?- (stdout)
   (global-sort (<- [?count ?eventtext] (info-tap ?eventtext)
(c/count ?count))["?count"]))
```

This query is run on Dataset2. Simple counting and sorting as in some previous examples would not work in this case. *:sort* actually means "when aggregating, perform a secondary sort by the provided parameters within each separate reduce task." Every Cascalog query can have zero or one aggregations. In above example query, there is one aggregator, but it's introduced by `*count*` operator, which itself is then the source of the ?count variable. So sorting by ?count becomes a paradox, asking Cascalog to sort by ?count before it exists, which produces an error message. The *:sort* sorts items before they go into the buffer. To sort on the way out is done using global sort like in above query which uses *IdentityBuffer* which produces desired output.

Listing 4.12: Identify the control unit that causes most startup problems.

```
(defn comparethis [c1 c2]
 (if (< c1 c2) "Turbine2" "Turbine1" ))

(let [value1 (<- [?count1 ?category_description]
                 (info-tap ?assembly ?category_description)
                 (= ?category_description "Event from the CU  which
indicates  automatic prevention from a start attempt")
                 (= ?assembly "Turbine1")(c/count ?count1))
     value2 (<- [?count2 ?category_description]
                 (info-tap ?assembly ?category_description)
                 (= ?category_description "Event from the CU  which
indicates  automatic prevention from a start attempt")
                 (= ?assembly "Turbine2")(c/count ?count2)) ]

(?<- (stdout) [?final-value]
       (value1 :> ?count1 ?category_description)
       (value2 :> ?count2 ?category_description)
(comparethis :< ?count1 ?count2 :> ?final-value ) ) )
```

This query is run on Dataset2. In this example of Listing 4.12, numbers of start problems of two control units Turbine1 and Turbine2 are compared to see which has the max start problems. In this query, 2 subqueries are formulated *value1* and *value2* which has information of count and category description. Then these 2 subqueries are used as generators to access values of ?count1 and ?count2, further on ?count1 and ?count2 are compared to find the control unit.

Listing 4.13: Find events when a specific sensor previously had a peak

```
(defn convert-to-long [a]
      (ct/to-long (f/parse custom-formatter a)))

(def info-tap
 (<- [?timestamp  ?BTT367 ]
     ((select-fields info ["?timestamp"  "?BTT367"])
?timestamp  ?BTT367)))

(defn convert-to-float [a]
  (try
     (if (not= a " ") (read-string a))
```

```
  (catch Exception e (do nil))))

(?<- (stdout) [?timestamp-out ?highest-value](info-tap ?timestamp
?BTT367)
     (convert-to-float ?BTT367 :> ?converted-BTT367 )
     (convert-to-long ?timestamp :> ?converted-timestamp)
     (>= ?converted-timestamp start-value)
     (<= ?converted-timestamp end-value)
     (:sort ?converted-BTT367)(:reverse true)
     (c/limit [1] ?timestamp ?converted-BTT367 :> ?timestamp-out
?highest-value))
```

This query is run on Dataset3.

Chapter 7

Performance Evaluation of Cascalog queries

As seen in chapter 6, we have a set of Cascalog queries on sensor data. Benchmarking is done on these Cascalog queries. Benchmarking is used to measure performance using a specific indicator. In our experiment this specific indicator is execution time, which we compare between the single node Hadoop and multinode Hadoop cluster. Also further on, we increase the data size to see effects on execution time. Some test cases are created with queries that differ in amount of data or complexity of queries.

7.1 Performance evaluation methods

In order to get the execution time of Cascalog queries, time expression of Clojure can be used. This looks like *(time (Cascalog query))*. This evaluates the expression and prints the time it takes after a single run.

Considering an example:

```
(time  (?<- (stdout)
        [?timestamp ?assembly ?category ]
       (info-tap :> ?timestamp ?assembly ?category )
       (clojure.string/trim ?category :> ?trimmed-category)
       (= ?trimmed-category "Start Inhibit")) )
```

The results of which looks like:
Elapsed time: 610.221708 msecs (on 1st run)
Elapsed time: 787.81836 msecs (on 2nd run)
Elapsed time: 680.919887 msecs (on 3rd run)

Here we see there are variations in elapsed time from one run to another. There are many factors that cause variations in run time between runs in most computations. For example: [16]

- Computers do not simply execute one process at a time. They continually switch from one process to another. So it depends on what other processes are running.
- The state of L1, L2, etc. caches in the CPU memory systems, because the access patterns to the caches depend not just on the references made by the program we are trying to measure, but on those of other processes executing concurrently.
- If files are being accessed on a mechanical disk, then the execution time depends on the head position.
- If multiple nodes are involved, what other traffic is on the network.
- The number of users sharing the system, the network traffic and timing of disk operations –these factors influence the exact scheduling of processor resources for one program.
- For JVM processes, as Clojure/Java is, whether or not the JIT compiler runs on our code, and if so, how the execution that occurred so far up to that point affects its "choices" in what kind of optimized code to produce.

So all the above history like thread handling, operating system or hardware issues can cause variations in execution time. Since the results from the above example are in milliseconds, it can be ignored. But in order to be little more accurate, running the experiment multiple times is preferred instead of reporting a single value. So statistics like min, max, median, arithmetic mean, 10^{th} percentile etc. can be used. In order to have statistically rigorous results, Cascalog queries have to be run number of times starting from 5 to 10 and then variance of results should be observed. More is the variance, more number of times query has to be run in order to be confident that the results characterize the range of run times that are likely to occur.

The above option would be a little complicated. Other option is Criterium. Criterium is a benchmarking library for Clojure. It measures the computation time of an expression. It is designed to address some of the pitfalls of benchmarking and benchmarking on JVM in particular. It includes [17]:

- Multiple evaluations are statistically processed.
- Purging of gc before testing, to isolate timings from gc state prior to testing.
- A final forced GC after testing to estimate impact of cleanup on the timing results.
- Inclusion of a warm-up period, designed to allow the JIT compiler to optimise its code.

Considering an example:

```
(use 'criterium.core)

(bench (?<- (stdout)
      [?timestamp ?assembly ?category ]
      (info-tap :> ?timestamp ?assembly ?category )
      (clojure.string/trim ?category :> ?trimmed-category)
      (= ?trimmed-category "Warning")))
```

Result looks like:
Evaluation count: 240 in 60 samples of 4 calls.
Execution time mean: 265.359848 ms
Execution time std-deviation: 25.544031 ms
Execution time lower quantile: 229.851248 ms (2.5%)
Execution time upper quantile: 310.110448 ms (97.5%)
Overhead used: 2.708614 ns

It means Criterium made 240 timed test runs in 4 batches of 60 samples each. And takes the mean execution time of those entire 240 test runs (instead of reporting on single value, multiple invocations are made to address the pitfall of benchmarking). "lower quartile" means the 2.5% of the 240 samples (0.025 * 240 = 6) that had the lowest time that is 229.8 ms. Overhead (in this case negligible small) is the time Criterium takes in the samples. The main focus in this result is on execution time mean. In this experiment benchmarking is done on all the queries using execution time mean outputted by Criterium.

7.2 Hardware specification

How fast does program X run on machine Y? -For such questions it is also important to consider processor performance of machine on which program X is run.

In our experiment Cascalog queries are run and performance tests are conducted on the server with hardware specifications as below:

- Transtec Calleo 552
- 2U Quad Opteron
- 2 x AMD 8-Core Processors
- 5 x 2TB disk space
- 2x GB Ethernet

7.3 Experimental Results

Queries described in chapter 6 as Listing are considered as test cases here. Listing number corresponds to those queries. Datasets explained in chapter 6 that is Dataset1, Dataset2 and Dataset3 are used here. Executing Cascalog queries on Dataset1 of sensor data on single node and multinode Hadoop cluster produces the following results.

Listing 4.1: Receive all sensor events with "Warning" category

Single node Hadoop:	Evaluation count: 180 in 60 samples of 3 calls. Execution time mean: 374.846440 ms Execution time std-deviation: 2.563211 ms Execution time lower quantile: 370.776542 ms (2.5%) Execution time upper quantile: 379.101664 ms (97.5%) Overhead used: 3.800577 ns
Multinode Hadoop:	Evaluation count : 180 in 60 samples of 3 calls. Execution time mean: 368.122151 ms Execution time std-deviation: 4.094212 ms (2.5%) Execution time lower quantile: 361.358928 ms (97.5%) Overhead used: 6.921949 ns

Listing 4.2: Receive all events with "Start Failure " problems and also the reasons for it

Single node Hadoop:	Evaluation count : 180 in 60 samples of 3 calls. Execution time mean: 369.652707 ms Execution time std-deviation: 3.058232 ms Execution time lower quantile: 357.260020 ms (2.5%) Execution time upper quantile: 367.880197 ms (97.5%) Overhead used : 3.812353 ns
Multinode Hadoop :	Evaluation count : 180 in 60 samples of 3 calls. Execution time mean: 362.918414 ms Execution time std-deviation: 3.357051 ms Execution time lower quantile: 365.137062 ms (2.5%) Execution time upper quantile: 376.331265 ms (97.5%) Overhead used: 3.825462 ns

Analyzing a bit about the above results, which also applies for queries below. As mentioned in the beginning of this chapter the main focus of results is on execution

time mean. And in some cases it could be interesting when the execution time std-deviation is too large. We see from the results that Execution time mean of multinode Hadoop is less than the single node Hadoop, which means Multinode Hadoop is faster in processing than single node Hadoop. But still the difference between single and multinode is not much in terms of execution time mean. This could be because our multinode Hadoop has two Datanodes running on a single machine. May be two datanodes on a single machine does not prove much. Multiple datanodes running on one server will compete for cores, bus and memory access, even if not for spindles.

Listing 4.3: Union of "Start Inhibit" and "Start Failure" events

```
Single node Hadoop:  Evaluation count: 60 in 60 samples of 1 calls.
                     Execution time mean: 15.447780 sec
                     Execution time std-deviation: 1.090036 sec
                     Execution time lower quantile: 14.907752 sec (2.5%)
                     Execution time upper quantile: 19.695728 sec (97.5%)
                     Overhead used: 3.777536 ns

Multinode Hadoop:    Evaluation count : 60 in 60 samples of 1 calls.
                     Execution time mean: 14.447780 sec
                     Execution time std-deviation: 1.000036 sec
                     Execution time lower quantile: 14.357752 sec (2.5%)
                     Execution time upper quantile: 19.505728 sec (97.5%)
                     Overhead used: 3.577536 ns
```

In the above query we see a noticeable amount of time is taken which is in seconds compared to milliseconds of previous queries. It is seen from some examples that unions, joins and few other features of Cascalog take considerable amount of time.

Listing 4.4: Retrieve all the events where sensor values are greater than 800.0

```
Single node Hadoop:   Evaluation count : 180 in 60 samples of 3 calls.
                      Execution time mean: 452.275013 ms
                     Execution time std-deviation: 4.961628 ms
                     Execution time lower quantile: 446.464824 ms (2.5%)
                     Execution time upper quantile: 459.080531 ms (97.5%)
                     Overhead used: 3.816389 ns

Multinode Hadoop:    Evaluation count : 180 in 60 samples of 3 calls.
                     Execution time mean: 462.707819 ms
                     Execution time std-deviation: 3.047105 ms
                     Execution time lower quantile: 457.096924 ms (2.5%)
                     Execution time upper quantile: 466.858320 ms (97.5%)
                     Overhead used: 3.781914 ns
```

In this result we see execution time mean of multinode Hadoop is little more than single node Hadoop. One thing that can be taken into consideration here is, there are variations in run time between computations because of many factors as mentioned in the beginning of this chapter. That means such results can also depend on which other processes are running on that machine at that point of time, to say it depends on machine load. Also since the results are in milliseconds such differences can be ignored.

Listing 4.5: Count of events where sensor values reached greater than 800.0

```
Single node Hadoop:  Evaluation count : 120 in 60 samples of 2 calls.
                     Execution time mean: 557.359011 ms
                     Execution time std-deviation: 7.434236 ms
                     Execution time lower quantile: 540.887860 ms (2.5%)
                     Execution time upper quantile: 563.301550 ms (97.5%)
                     Overhead used: 3.799133 ns

Multinode Hadoop:    Evaluation count : 120 in 60 samples of 2 calls.
                     Execution time mean: 551.987749 ms
                     Execution time std-deviation: 8.658237 ms
                     Execution time lower quantile: 546.434380 ms (2.5%)
                     Execution time upper quantile: 570.524491 ms (97.5%)
                     Overhead used: 3.795115 ns
```

Listing 4.6: Retrieve the event when highest value of sensor occurred

```
Single node Hadoop:  Evaluation count: 120 in 60 samples of 2 calls.
                     Execution time mean: 643.488997 ms
                     Execution time std-deviation: 13.276665 ms
                     Execution time lower quantile: 604.045185 ms (2.5%)
                     Execution time upper quantile: 657.561964 ms (97.5%)
                     Overhead used: 3.783847 ns

Multinode Hadoop:    Evaluation count : 120 in 60 samples of 2 calls.
                     Execution time mean: 600.405934 ms
                     Execution time std-deviation: 23.308440 ms
                     Execution time lower quantile: 572.196440 ms (2.5%)
                     Execution time upper quantile: 638.509888 ms (97.5%)
                     Overhead used: 3.836950 ns
```

Also overhead used in result means time taken by Criterium in samples. This is very negligible in all cases since it is in nanoseconds.

Listing 4.7: Sensor events between 12:05:2010 10:00:00 and 12:05:2010 11:00:00

```
Single node Hadoop:  Evaluation count: 120 in 60 samples of 2 calls.
                     Execution time mean: 600.980124 ms
                     Execution time std-deviation: 5.490431 ms
                     Execution time lower quantile: 591.699433 ms (2.5%)
                     Execution time upper quantile: 612.915271 ms (97.5%)
                     Overhead used: 4.404661 ns

Multinode Hadoop:    Evaluation count: 120 in 60 samples of 2 calls.
                     Execution time mean: 597.772996 ms
                     Execution time std-deviation: 5.589792 ms
                     Execution time lower quantile: 588.009814 ms (2.5%)
                     Execution time upper quantile: 607.226288 ms (97.5%)
                     Overhead used: 3.815941 ns
```

Listing 4.8: Highest sensor value between specific time period

```
Single node Hadoop:   Evaluation count: 120 in 60 samples of 2 calls.
                       Execution time mean: 702.486660 ms
                      Execution time std-deviation: 10.095460 ms
                      Execution time lower quantile: 691.313139 ms (2.5%)
                      Execution time upper quantile: 719.952195 ms (97.5%)
                      Overhead used: 3.801515 ns

Multinode Hadoop:     Evaluation count: 120 in 60 samples of 2 calls.
                      Execution time mean: 708.880906 ms
                      Execution time std-deviation: 6.922659 ms
                      Execution time lower quantile: 702.124179 ms (2.5%)
                      Execution time upper quantile: 723.876528 ms (97.5%)
                      Overhead used: 3.777129 ns
```

Listing 4.10: Collect all messages related to potential start problems, concerning some time period in the past.

```
Single node Hadoop:  Evaluation count: 60 in 60 samples of 1 calls.
                      Execution time mean: 23.429731 sec
                      Execution time std-deviation: 219.919412 ms
                       Execution time lower quantile: 23.035365 sec (2.5%)
                      Execution time upper quantile: 23.864622 sec (97.5%)
                      Overhead used: 6.920535 ns

Multinode Hadoop:    Evaluation count: 60 in 60 samples of 1 calls.
                      Execution time mean: 23.398958 sec
                      Execution time std-deviation: 201.265028 ms
                      Execution time lower quantile: 23.103671 sec (2.5%)
                       Execution time upper quantile: 23.760843 sec (97.5%)
                       Overhead used: 3.801878 ns
```

Listing 4.11: Get a sorted list of event text and number of occurrences of all these events.

```
Single node Hadoop: Evaluation count: 60 in 60 samples of 1 calls.
                      Execution time mean: 39.528952 sec
                      Execution time std-deviation: 543.850935 ms
                      Execution time lower quantile: 38.093032 sec (2.5%)
                      Execution time upper quantile: 40.118840 sec (97.5%)
                      Overhead used: 3.790229 ns

Multinode Hadoop:   Evaluation count : 60 in 60 samples of 1 calls.
                      Execution time mean: 39.528952 sec
                      Execution time std-deviation: 543.850935 ms
                      Execution time lower quantile: 38.093032 sec (2.5%)
                      Execution time upper quantile: 40.118840 sec (97.5%)
                      Overhead used: 3.790229 ns
```

Listing 4.12: Identify the control unit that causes most startup problems.

```
Single node Hadoop:  Evaluation count: 60 in 60 samples of 1 calls.
                     Execution time mean: 1.181832 min
                     Execution time std-deviation: 1.270972 sec
                     Execution time lower quantile: 1.128102 min (2.5%)
                     Execution time upper quantile: 1.191209 min (97.5%)
                     Overhead used: 3.778576 ns

Multinode Hadoop:    Evaluation count: 60 in 60 samples of 1 calls.
                     Execution time mean: 1.154474 min
                     Execution time std-deviation: 2.876200 sec
                     Execution time lower quantile: 1.127387 min (2.5%)
                     Execution time upper quantile: 1.290471 min (97.5%)
                     Overhead used: 6.935840 ns
```

In the above queries as of Listing 4.10, 4.11 and 4.12, we see a noticeable amount of execution time is taken which is in seconds and in the query as of Listing 4.12 it is in minutes. This could be because these queries are tested on bigger dataset, which is Dataset2 and also can depend on the complexity of the query, which uses some Cascalog features that consumes time.

After looking at all the above results it is evident that running multiple datanodes on a single machine does not provide very significant performance results. To say results from single node and multinode are nearly equal. By default, Hadoop is configured to run in a non-distributed mode, as a single java process. To have a look at this we use *top* command on terminal, which provides an ongoing look at processor activity in real time. The result of which indicated that the whole Hadoop job was running as a single process and only one cpu was utilized even though there were several of them. This is an evidence for the fact that there was no data distribution between cores of the machine, by having multiple datanodes installed on single machine. Below tables indicates the result of *top* Unix command:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27752	username	20	0	1867m	1.1g	12m	S	101	0.8	9.24.76	java
27423	username	20	0	1919m	332m	11m	S	1	0.3	0.21.51	java
28167	username	20	0	17572	1708	1052	R	1	0.0	0.04.68	top
---	---	---	---	---	---	---	---	---	---	---	---

Cpu0	0.3%us	0.3%sy	0.0%ni	99.3%id	0.0%wa	0.0%hi	0.0%si	0.0%st
Cpu1	---	---	---	---	---	---	---	---
Cpu2	---	---	---	---	---	---	---	---
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
Cpu18	100%us	0.0%sy	0.0%ni	0.0%id	0.0%wa	0.0%hi	0.0%si	0.0%st
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

Table 3: Results of *top* command

There are some ways used in the experiment to improve the execution time of the queries across the vertical stack which is shown in Fig 13.

a) Starting from the bottom, we can add more hardware nodes to our cluster. This is not being done in our experiment since the required hardware was not available. So we have set up multiple datanodes on single machine. In this experiment we check if there exists any other methods to improve performance on single machine rather than adding more hardware nodes and if they exist do all of them apply for Cascalog?

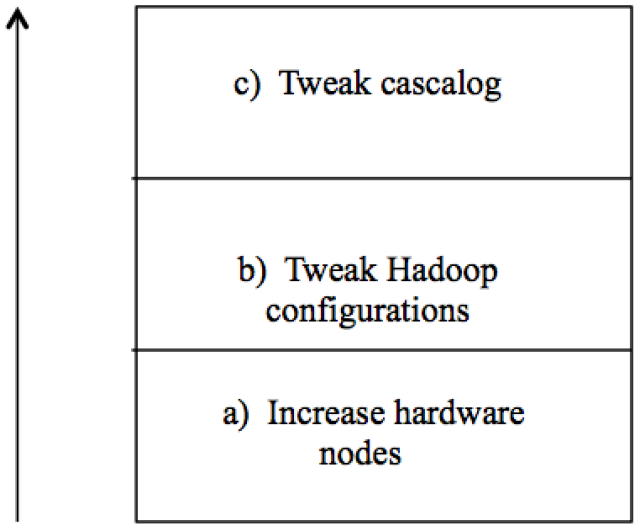


Fig 13: Stack of performance improvements methods

b) The next in the stack to improve performance would be trying to tweak Hadoop. To maximize MapReduce job performance some performance improvement strategies can be implemented. Configure memory settings, map and reduce slot numbers and job parameters to tune the cluster for optimal MapReduce performance. Below are some of the parameters affecting the performance:

1. Chunk size

Chunk size affects parallel processing and random disk I/O. A higher chunk size means less parallel processing because there are fewer map inputs and therefore fewer mappers. A lower chunk size improves parallelism, but results in higher random disk I/O during shuffle because there are more map outputs. *dfs.block.size* is the property in configuration file hdfs-site.xml of Hadoop which refers to the file system block size. The default value is 64MB as explained in section 3.4.1. When there is a small cluster and large data set then the default block size will create a large number of map tasks.

Example: Input data size = 10 GB
If dfs.block.size = 64 MB then the minimum no. of maps=(10*1024)/64 = 160 maps.
If dfs.block.size = 128 MB then minimum no. of maps=(10*1024)/128 = 80 maps.

Considering a query as of Listing 4.13 which is run on Dataset3 below:
When dfs.block.size is 64MB, the execution time of the query looks like below

```
Elapsed time: 811491.110925 msecs
```

(time expr) of Clojure is used. This is the query, which takes significant amount of time say 13.52 minutes that is 811491.110925 msecs.
When dfs.block.size is 128MB, the execution time of the query looks like below

```
Elapsed time: 868218.896312 msecs
```

The above two results indicate that still there is no significant improvement in performance.

2. Setting the Number of Mappers and Reducers on Each Node

mapred.child.java.opts:

Controls the amount of memory that is allocated to a task.

mapred.tasktracker.map.tasks.maximum:

Controls the maximum number of map tasks that are created simultaneously on Task Tracker.

mapred.tasktracker.reduce.tasks.maximum:

Controls the maximum number of map tasks that are created simultaneously on Task Tracker.

mapred.reduce.tasks:

It is the default number of reduce tasks per job.

The setting mapred.tasktracker.* related settings are related to maximum number of maps or reducers a tasktracker can run. For example if we set it to 4, it will basically mean that at any given point the tasktracker running on that machine will run the maximum of 4 maps or reducers. MapReduce parallelization can be increased by setting mapred.tasktracker.* values. All the above properties are set in configuration file mapred-site.xml of Hadoop.

The mapper number is usually not a performance bottleneck. We then try to increase the number of reducers and MapReduce parallelism using above-mentioned properties. Since we have set up Hadoop in non-distributed mode, it can't run more than one reducer. It can support the zero reducer case, too. This is normally not a problem, as most applications can work with one reducer, although on a cluster you would choose a larger number to take advantage of parallelism. The thing to watch out for is that even if we set the number of reducers to a value over one, it will silently ignore the setting and use a single reducer. Also to the reducers question, if we are using an expressive wrapper such as cascalog for MapReduce, then it also depends on what we are doing in it. If we are computing an operation such as a total count, or a global max for example, then the wrapper may by itself set the number of reducers to 1, since this is the only way to do a global sum/count. So this shows that we cannot use above mentioned mapred.* parameters to take advantage of parallelism since it is too application dependent.

In our experiment, we try to use these above-mentioned properties to increase parallelization. But most of the queries use the default number of reducers that is 1. So there was no significant difference in execution time of the queries.

c) The next up in the stack to improve performance would be to tweak our Cascalog query to get maximum performance. That is to optimize the cascalog queries - like using a buffer instead of many aggregators and avoiding any redundancies. In our experiment we had no such queries; all of them were already tuned to get maximum performance.

However multiple Cascalog queries can be submitted simultaneously to Hadoop. ??- is a Cascalog query operator that accepts any number of queries (defined by <-), executes them in parallel and returns a sequence of sequences of the results of each query's execution.

Considering an example below:

```
(def multiple-results
(??- (<- [?timestamp ?assembly ?category ]
     (info-tap2 :> ?timestamp ?assembly ?category )
     (clojure.string/trim ?category :> ?trimmed-category)
     (= ?trimmed-category "Warning"))
(<- [?timestamp ?assembly ?category ]
     (info-tap2 :> ?timestamp ?assembly ?category )
     (clojure.string/trim ?category :> ?trimmed-category)
     (= ?trimmed-category "Start Failure"))))
```

In the above example we see there are 2 Cascalog queries defined by <- operator, one query retrieves all the events with "Warning" category and other retrieve events with "Start Failure" category. These 2 queries are executed simultaneously since the *??-* operator is being used.

When we run above query on Hadoop we see some logs from Hadoop like below:

```
14/08/16 21:59:16 INFO flow.Flow: []  parallel execution is enabled: true
14/08/16 21:59:16 INFO flow.Flow: []  starting jobs: 2
14/08/16 21:59:16 INFO flow.Flow: []  allocating threads: 2
14/08/16  21:59:16  INFO  flow.FlowStep:  []  starting  step:  (1/2)  ...e3-4263-ae7c-
09c62d1a2d37
14/08/16  21:59:16  INFO  flow.FlowStep:  []  starting  step:  (2/2)  ...07-4a9b-ada1-
5cf88b914ec4
14/08/16 21:59:17 INFO mapred.FileInputFormat: Total input paths to process : 1
14/08/16 21:59:17 INFO mapred.FileInputFormat: Total input paths to process : 1
14/08/16 21:59:17 INFO flow.FlowStep: [] submitted hadoop job: job_201408162142_0001
14/08/16 21:59:17 INFO flow.FlowStep: [] submitted hadoop job: job_201408162142_0002
```

The above logs clearly explain that 2 Hadoop jobs have been started in 2 different threads. It is also seen that parallel execution is enabled to true and the 2 Hadoop jobs have unique Id's. This parallel execution permits multiple independent threads of execution, to better utilize the resources provided by modern processor architectures.
The results of *top* Unix command showed multiple cores are being utilized. This shows that Hadoop can run multiple Cascalog queries simultaneously. Since the machine on which experiment is conducted have multiple processor cores, our program can use threads to take advantage of additional cores to perform our processing. If we don't use threads at all, our program will only utilize a single processor core, which is still fine if that's all is needed. Threads cannot speed up execution of code. All they can do is increase the efficiency of the computer by using time that would otherwise be wasted.

Conclusion

In today's hyper-connected world, where data is growing so rapidly and the rise of unstructured data is accounting for 90% of digital data, which is beyond the processing capabilities of traditional databases. The time has come for enterprises to re-evaluate the approaches of data storage and management. Hadoop provides the new way of storing and processing data. No data is big with Hadoop. Scalability, cost-effectiveness and streamlined architectures of Hadoop are making it more and more attractive. There are number of technologies that work on top of Hadoop for querying and processing big data. One such tool is Cascalog.

The whole experiment is about formulating the queries in Cascalog, in turn understanding the Cascalog query structure, features of Cascalog and its simplicity. Together with Cascading and Clojure, Cascalog is highly impressive and provides a powerful ecosystem for manipulating and analysing large data sets.

In this experiment sensor data is queried using Cascalog and then some performance tests are conducted. Performances of Cascalog queries are compared, which are run on single node Hadoop and multinode Hadoop. Also different sizes of sensor data are used to compare the performance. Further on, complexity of queries is also increased to see the performance. The results of the experiment prove that setting up multiple datanodes on a single machine does not show significant performance improvements. The extended work of this thesis which can be done in the future could be, to use multiple hardware nodes and see the performance improvements.

Investing in learning Cascalog will pay rich dividends. The little down side of Cascalog is that the learning curve for a complete beginner is quite steep and whilst of great use, the limited documentation and tutorial available will make the hill harder to climb.

Bibliography

[1] http://hadoop.apache.org

[2] http://www.emc.com/leadership/digital-universe/index.htm

[3] http://www.statisticbrain.com/twitter-statistics/

[4] http://zephoria.com/social-media/top-15-valuable-facebook-statistics/

[5] www.scn.sap.com

[6] http://www.cascalog.org/articles/marz_intro_1.html

[7] http://www.nathanmarz.com

[8] http://cascalog.org/articles/users.html

[9] http://clojure.com/blog/2012/02/03/functional-relational-programming-with-cascalog.html

[10] https://github.com/nathanmarz/cascalog

[11] http://www.slideshare.net/nathanmarz/cascalog-at-hadoop-day

[12] http://leiningen.org

[13] http://www.cis.upenn.edu/~matuszek/Concise/Guides/Concise/Clojure.html

[14] http://www.michael-noll.com/tutorials/

[15] http://www.guruzon.com

[16] http://csapp.cs.cmu.edu/public/1e/public/ch9-preview.pdf

[17] https://github.com/hugoduncan/criterium

[18] http://blog.brunobonacci.com

[19] http://en.wikipedia.org/wiki/Internet_of_Things

[20] http://datasensinglab.com

[21] http://www.sas.com/resources/asset/five-big-data-challenges-article.pdf

[22] http://www.mongodb.com/big-data-explained

[23] http://globalsp.ts.fujitsu.com/dmsp/Publications/public/br-bigdata-revolution.pdf

[24] http://radar.oreilly.com/2011/01/what-is-hadoop.html

[25] http://en.wikipedia.org/wiki/Apache_Hadoop

[26] http://clojure.org/getting_started

[27] http://www.mongodb.org

[28] http://cassandra.apache.org

[29] http://hive.apache.org

[30] www.optique-project.eu

[31] http://www.gartner.com/it-glossary/big-data/

[32] http://www.cbinsights.com/blog/big-data-report

[33] http://blogs.wsj.com/experts/2014/03/28/sectors-where-big-data-could-make-an-impact/

[34] http://www.informationweek.com/big-data/big-data-analytics/big-data-development-challenges-talent-cost-time/d/d-id/1105829?

[35] http://www.foreignaffairs.com/articles/139104/kenneth-neil-cukier-and-viktor-mayer-schoenberger/the-rise-of-big-data

[36]http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

[37] http://en.wikipedia.org/wiki/Relational_database_management_system

[38] http://dba.stackexchange.com/questions/13931/why-cant-relational-databases-meet-the-scales-of-big-data

[39] http://www.zdnet.com/blog/btl/big-data-vs-traditional-databases-can-you-reproduce-youtube-on-oracles-exadata/52053

[40] http://scaledb.com/pdfs/BigData.pdf

[41] http://zookeeper.apache.org/

[42] http://hbase.apache.org/

[43] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[44] http://en.wikipedia.org/wiki/MapReduce

[45] http://pig.apache.org/

[46] https://github.com/jeffsack/playground/blob/master/src/playground/test_data.clj