Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Praktikum für Fortgeschrittene

Hadoop Performance Evaluation

Name:	Tien Duc Dinh
Matrikelnummer:	2592222
Betreuer:	Olga Mordvinova, Julian Kunkel
Abgabe Datum:	March 10, 2009

1 Abstract

This article delivers results of a work whose goal is the Hadoop evaluation.

Hadoop is a framework which enables applications to work on petabytes of data on large clusters with thousand of nodes built of commodity hardware. It provides a distributed file system (HDFS) that stores data on the computed nodes, providing very high aggregate bandwidth across the cluster. In addition, Hadoop implements a parallel computational paradigm named MapReduce which divides the application into many small fragments of work, each of which may be executed or reexecuted on any node in the cluster.

To measure the performance we will set up a Hadoop cluster with many nodes and use the file TestDFSIO.java of the Hadoop version 0.18.3 which gives us the data throughput, average I/O rate and I/O rate standard deviation.

The HDFS writing performance scales well on both small and big data set. The average HDFS reading performance scales well on big data set where it is - however - lower than on the small data set.. The more nodes a writing/reading operation is run on, the faster its performance is. This work also draws a comparison between the HDFS and local filesystem performance.

Contents

2 Introduction

Hadoop is an open-source, Java-based programming framework that supports the processing of large data sets in a distributed computing environment. It was inspired by Google MapReduce and Google File System (GFS) papers [1].

Hadoop is now a top level Apache project, being built and used by a community of contributors from all over the world, since it's easy to install, configure and can be run on many platforms supporting Java. The Hadoop framework is currently used by major players including Google, Yahoo and IBM, [7] largely for applications involving search engines and advertising.

The major contributions of this work are a Hadoop performance evaluation on writing/reading and full understanding about the MapReduce concept as well as the distribution of processes.

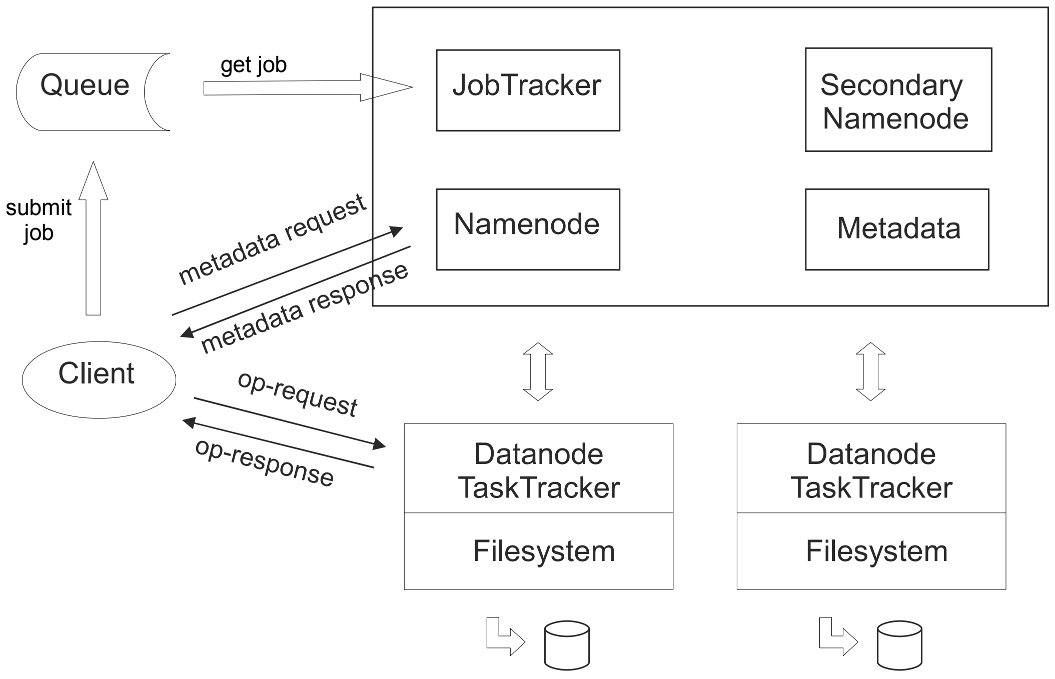
Section 3 describes the HDFS [1] [2] and MapReduce [1] concept with a small example. Section 4 describes the installation and configuration a HDFS cluster. Section 5 describes the test preparation towards our cluster enviroment. Section 6 has performance measurements based on many test cases and a comparision between the HDFS and local file system on the testing cluster.

[1]HDFS = Hadoop Distributed File System

3 HDFS Model

In this chapter we will go through the HDFS [1] architecture and MapReduce concept with a MapReduce example in order to gain full knowledge about them. [2]

3.1 HDFS Architecture



HDFS has a client/server architecture. A HDFS cluster consists of two masters: Namenode and JobTracker, multiple Datanodes and is acessed by many clients.

Client

A client is an api of applications. It communicates with the Namenode because of metadata and after receiving them, it directly runs operations on the Datanodes. If the operation is a MapReduce operation, the client creates a job and sends it to the queue. The JobTracker handles this queue.

[1]HDFS = Hadoop Distributed File System

Namenode

Namenode is the master server which maintains all file system metadata like the namespace, access control information, the mapping from files to blocks and the current location of blocks. Block locations are not stored on the Namenode permanently, it collects by asking Datanodes while starting up or when a new Datanode is connected to the cluster instead. The reason why the Namenode does not store block locations is, that the locations become easily changeable, e.g. a Datanode is failed or newly connected (these events occur quite frequently).
The NameNode executes file system namespace operations like opening, closing, renaming files and directories and gives instructions to the Datanodes to perform system operations, e.g. block creation, deletion, replication, etc. Normally it's about an operation (write/read/append) on the cluster. Basing on the system resources and the input filesize the Namenode decides which Datanodes the clients should connect and responds this information to the Client. Basically the Namenode shouldn't execute any operations on its node [2] to avoid becoming a bottelneck. Having only one Namenode simplifies the design and implementation of other complex operations like block placement, block replication, cluster rebalancing, etc.

Datanode

A Datanode, usually one per node, stores HDFS data in its local file system and runs client operations or performs system operations upon instruction from the Namenode.
HDFS is designed for processing huge data sets, so a data set is often very big and split into many blocks stored and replicated across the Datanodes. A Datanode normally has no knowledge about HDFS files. While starting up, it scans through the local file system and creates a list of HDFS data blocks conrresponding to each of these local files and sends this report to the Namenode.
A Datanode doesn't store all files in the same directory, it uses a heuristic to calculate which number of files is best for the local file system and creates subdirectories suitably.

Secondary Namenode

Modifications to the file system are stored as a log file by the Namenode. While starting up, the Namenode reads the HDFS state from an image file (fsimage) and then applies the modifications from the log file. After the Namenode finished writing the new HDFS state to the image file, it empties the log file. Because the Namenode merges fsimage and edits files only during start up, so the log file might become very big and the next restart might take longer. To avoid this problem the Secondary Namenode merges fsimage and the log file periodically to keep the log size within a limit.

TaskTracker

A TaskTracker is a node in the cluster that accepts MapReduce tasks from the JobTracker. Every TaskTracker is configured with a set of slots, these indicate the number of tasks that it can accept. The TaskTracker spawns a separate JVM processes to do the actual work, this helps to ensure that process failure does not take down the TaskTracker. The TaskTracker monitors these spawned processes, captures the outputs and exit codes. When the process finishes, suc-

[2] It's possible in Hadoop by the configuration

cessfully or not, the tracker notifies the JobTracker. The TaskTrackers also send out heartbeat messages to the JobTracker, usually every few minutes, to reassure that the JobTracker is still alive. These messages also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster work can be delegated to. [6]

JobTracker

The JobTracker is the MapReduce master which normally runs on a separate node. Hier is an overview how the JobTracker works.

1. Client applications submit jobs to the Job tracker through a queue.
2. The JobTracker talks to the NameNode to determine the location of the data.
3. The JobTracker locates TaskTracker nodes with available slots at or near the data to reduce network traffic on the main backbone network.
4. The JobTracker submits the work to the chosen TaskTracker nodes.
5. The TaskTracker nodes are monitored through heartbeat signals in a time interval. When a task fails, the JobTracker decides what to do then: it may resubmit the job elsewhere, mark that specific record as something to avoid or even blacklist the TaskTracker as unreliable.
6. When the work is completed, the JobTracker updates its status.
7. Client applications can poll the JobTracker for information.

The JobTracker is a single point of failure for the Map/Reduce infrastructure. If it goes down, all running jobs are lost. The fileystem remains live. There is currently no checkpointing or recovery within a single MapReduce job [5]

3.2 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Its functions like *map* and *reduce* are supplied by the user and depend on user's purposes.

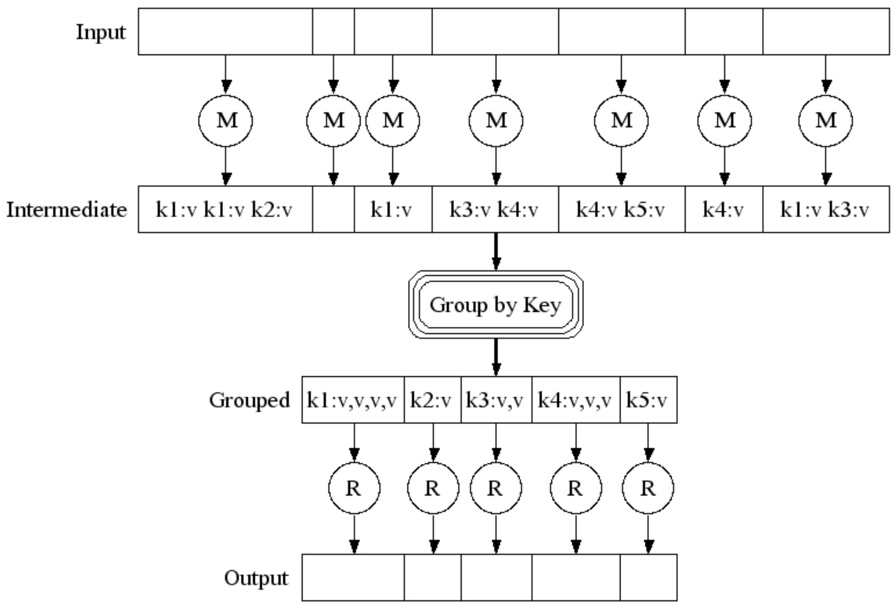
MapReduce has 2 main parts:

- Map: processes a key value pair to generate a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function.
- Reduce: accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values.

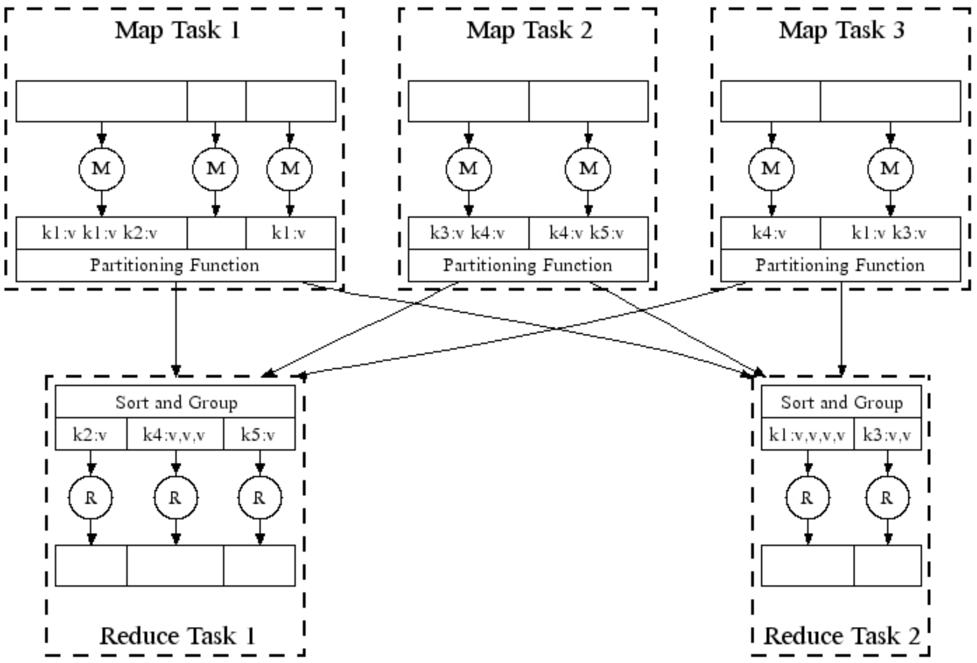
When an input data set is processed by a MapReduce job, depending on its size it will be split into many independent smaller split data sets, which are commited to the map tasks. After the map phase is completed the framework sorts the outputs of the map tasks and commits them to the reduced tasks. The MapReduce process can run serveral times instead of only once.

The advantage of MapReduce is that MapReduce's tasks can be run in a completely parallel manner for instance.

Execution:



Parallel execution on many nodes:



The above graphic shows that an input data set is split into three split sets which are committed to the map tasks and each map task runs on a different node. For example the map task "*1*" handles the 1st split data set containing 3 files and calls one map function for each file.

You might be asking yourself, what happens if the number of split data sets is not equal to the number of map tasks ?

Hadoop is implemented in a way, that ensures they are always equals, because each map task can only work with one split data set. The map task number can be configured in the file "*hadoop-site.xml*" but this is not the real map task number at the end. Basing on the map task configuration and other information about the data set, e.g. the data set size, etc. Hadoop will calculate how many parts the data set should be split. The HDFS cluster should always work correctly, even if we have a bad configuration for instance.

Now we take a look at the next example for more understanding about MapReduce.

Pseudo-Code example: Count word occurrences

```
map(String key, String value):
  // key: document name (usually key isn't used)
  // value: document contents
  for each word w in value:pair.
    EmitIntermediate(w, "1");
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

For example we have the folder "*data*" which contains two files a and b with the following contents.

```
a : Hello World Bye World
b : Hello Hadoop Goodbye Hadoop
```

The following UNIX-command will solve this problem:

```
perl -p -e 's/s+/n/g' data/* | sort | uniq -c
```

And the output looks like:

```
1 Bye
1 Goodbye
2 Hadoop
2 Hello
2 World
```

Let there be two map tasks and two reduced tasks:

Map:

Map 1	Map 2
Hello → <Hello,1>	Hello → <Hello,1>
World → <World,1>	Hadoop → <Hadoop,1>
Bye → <Bye,1>	Goodbye → <Goodbye,1>
World → <World,1>	Hadoop → <Hadoop,1>

G&S for Reduce 1	G&S for Reduce 2
Goodbye → <Goodbye,1>	Bye → <Bye,1>
Hadoop → <Hadoop,1,1>	Hello → <Hello,1,1>
	World → <World,1,1>

Reduce:

Reduce 1	Reduce 2
Goodbye → <Goodbye,1>	Bye → <Bye,1>
Hadoop → <Hadoop,2>	Hello → <Hello,1>
	World → <World,1>

Practise with HDFS Streaming

The Hadoop framework is written in Java, but MapReduce applications can be implemented in other programming languages. Therefore HDFS provides users an api "*hadoop streaming*" for manipulating jobs with any executing programs as a mapper/reducer.

With this above example we can use the next commands to perform a MapReduce job:

# Copy the folder "*data*" onto the HDFS
hadoop-0.18.3/bin/hadoop fs -put data /

# Create and run the job with our mapper/reducer
hadoop-0.18.3/bin/hadoop jar hadoop-0.18.3/contrib/streaming/hadoop-0.18.3-streaming.jar -input /data -output /out -mapper "perl -p -e 's/\s+/\n/g' " -reducer "uniq -c"

```
additionalConfSpec_:null
null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
packageJobJar: [/tmp/hadoop-dinh/hadoop-unjar31880/] [] /tmp/streamjob31881.jar tmpDir=null
09/02/08 15:48:15 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments.
Applications should implement Tool for the same.
09/02/08 15:48:15 INFO mapred.FileInputFormat: Total input paths to process : 2
09/02/08 15:48:15 INFO mapred.FileInputFormat: Total input paths to process : 2
09/02/08 15:48:16 INFO streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-dinh/mapred/local]
09/02/08 15:48:16 INFO streaming.StreamJob: Running job: job_200902081446_0002
09/02/08 15:48:16 INFO streaming.StreamJob: To kill this job, run:
09/02/08 15:48:16 INFO streaming.StreamJob: /home/dinh/v-0.18.3/hadoop-0.18.3/bin/../bin/hadoop job -
Dmapred.job.tracker=node04:44562 -kill job_200902081446_0002
09/02/08 15:48:16 INFO streaming.StreamJob: Tracking URL:
http://node04.pvscluster:50030/jobdetails.jsp?jobid=job_200902081446_0002
09/02/08 15:48:17 INFO streaming.StreamJob: map 0% reduce 0%
09/02/08 15:48:20 INFO streaming.StreamJob: map 100% reduce 0%
09/02/08 15:48:26 INFO streaming.StreamJob: map 100% reduce 50%
09/02/08 15:48:27 INFO streaming.StreamJob: map 100% reduce 100%
09/02/08 15:48:27 INFO streaming.StreamJob: Job complete: job_200902081446_0002
09/02/08 15:48:27 INFO streaming.StreamJob: Output: /out
```

With 2 reduced tasks we will end up with 2 reduced output files on the HDFS

hadoop-0.18.3/bin/hadoop fs -cat /out/part-00000

```
1 Goodbye
2 Hadoop
```

hadoop-0.18.3/bin/hadoop fs -cat /out/part-00001

```
1 Bye
2 Hello
2 World
```

We can have many reduced output files on the HDFS instead of one, because the HDFS application can handle it correctly and the MapReduce's process can run many times.

4 Installation and Configuration

This chapter shows how to install and configure Hadoop on a local as well as cluster server. [4, 3]

4.1 Pre-requisites

Supported Platforms

Hadoops supports GNU/Linux as as a development and production platform and only supports Win32 as a development platform.

Required Softwares

- Java VM 1.5.x or newer versions (I use the version 1.6.0 for my test).
- ssh and sshd (hadoop needs sshd to enable remote accesses).
- Windows users may install cygwin to have a linux enviroment for shell support.

4.2 On a single node

4.2.1 Installation

For installation we have to extract the hadoop release version. To extract it, we can use the following command:

tar -xvf hadoop-.tar.gz*

4.2.2 Configuration

I use in this article the version *hadoop-0.18.3* and it's the hadoop folder's name too, I just call it *HADOOP* for convenience. All the configuration files are located in the folder *HADOOP/conf/*

and we need to modify some of them:

HADOOP/conf/hadoop-env.sh

This is the file for the configuration of hadoop-specific enviroment variables, e.g. *JAVA_HOME*, *HADOOP_CLASSPATH*. For hadoop enviroment we only need to configure *JAVA_HOME*. Just change the path of *JAVA_HOME* in this file to the path that we installed java, for instance:

Change the line

# export JAVA_HOME=/usr/lib/j2sdk1.5-sun

to

export JAVA_HOME=/usr/lib/jvm/java-6-sun

HADOOP/conf/hadoop-site.xml

The properties defined in this file override the default properties from the *HADOOP/conf/hadoop-default.xml*. We need to define at least, where the data should be written to, the Namenode and JobTracker, for example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!– Put site-specific property overrides in this file. –>
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-${user.name}</value>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:10000</value>
</property>
<property>
   <name>mapred.job.tracker</name>
   <value>localhost:10001</value>
</property>
<property>
   <name>dfs.replication</name>
   <value>1</value>
</property>
</configuration>
```

4.2.3 ssh access without password

If a password is needed when running "*ssh localhost*", we can run this script to overcome it:

ssh-keygen -t dsa -P "" -f ˜/.ssh/id_ dsa
cat ˜/.ssh/id_ dsa.pub >> ˜/.ssh/authorized_ keys

4.2.4 Format HDFS

HADOOP/bin/hadoop namenode -format

```
09/02/04 22:31:04 INFO dfs.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = vn/127.0.1.1
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 0.18.3
STARTUP_MSG: build = http://svn.apache.org/repos/asf/hadoop/core/branches/branch-0.18 -r 686010;
compiled by 'hadoopqa' on Thu Aug 14 19:48:33 UTC 2008
************************************************************/
09/02/04 22:31:05 INFO fs.FSNamesystem:
fsOwner=duc,duc,adm,dialout,cdrom,floppy,audio,dip,video,plugdev,scanner,lpadmin,
admin,netdev,powerdev,pulse-access,pulse-rt
09/02/04 22:31:05 INFO fs.FSNamesystem: supergroup=supergroup
09/02/04 22:31:05 INFO fs.FSNamesystem: isPermissionEnabled=true
09/02/04 22:31:05 INFO dfs.Storage: Image file of size 77 saved in 0 seconds.
09/02/04 22:31:05 INFO dfs.Storage: Storage directory /home/duc/
v-0.18.3/hadoop-0.18.3/tmp/dfs/name has been successfully formatted.
09/02/04 22:31:05 INFO dfs.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at vn/127.0.1.1
************************************************************/
starting namenode, logging to /home/duc/v-0.18.3/hadoop-0.18.3/bin/../logs/hadoop-duc-namenode-vn.out
localhost: starting datanode, logging to /home/duc/v-0.18.3/hadoop-0.18.3/bin/../logs/hadoop-duc-
datanode-vn.out
localhost: starting secondarynamenode, logging to /home/duc/v-0.18.3/hadoop-0.18.3/bin/../ logs/hadoop-
duc-secondarynamenode-vn.out
starting jobtracker, logging to /home/duc/v-0.18.3/hadoop-0.18.3/bin/../logs/hadoop-duc-jobtracker-vn.out
```

```
localhost: starting tasktracker, logging to /home/duc/v-0.18.3/hadoop-0.18.3/bin/../logs/hadoop-duc-tasktracker-vn.out
```

Note: This command doesn't delete everything in your "*hadoop.tmp.dir*", so this occurs a problem when we format newly again. To avoid this, we should delete manually.

4.2.5 Start HDFS and MapReduce daemons

HADOOP/bin/start-all.sh

If everything runs correctly, the "*jps*"command should deliver NameNode, DataNode, SecondaryNameNode, JobTracker and TaskTracker, for example:

```
18920 NameNode
19403 TaskTracker
19857 Jps
19280 JobTracker
19175 SecondaryNameNode
19021 DataNode
```

4.2.6 Stop HDFS and MapReduce daemons

HADOOP/bin/stop-all.sh

4.3 On a multi node

This section bases on the above section. To avoid text duplicates only new modifications are written here.

4.3.1 Installation

We need to install Hadoop on all cluster's nodes. 13

4.3.2 Configuration

The file */etc/hosts* contains a list of IP addresses and the hostnames they correspond to. We can add either hostnames or IP addresses to the *HADOOP/conf/masters* to identify which nodes are masters or slaves of the HDFS.

Typically one machine acts as the Namenode, another one as the Jobtracker (masters). The remaining nodes can act as Datanode and Tasktracker (slaves).

Note: A node configured as master shouldn't be configured as a slave as well because of the performance of the HDFS.

Here is an example on how to configure a cluster with two masters and three slaves:

masters	slaves
node01	node03
node02	node04
	node05

HADOOP/conf/hadoop-site.xml

In general all configurations are possible, but the best configuration is one separate node for Namenode and one for JobTracker, for example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!– Put site-specific property overrides in this file. –>
<configuration>
<property>
<name>hadoop.tmp.dir</name>
<value>/tmp/hadoop-$user.name</value>
</property>
<property>
<name>fs.default.name</name>
<value>hdfs://node01:54310</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>node02:44562</value>
</property>
</configuration>
```

4.3.3 Format HDFS

During my experiment I found out that Hadoop can only be formatted correctly if we format on the node configured as the Namenode (Namenode's node), for example:

ssh node_configured_as_Namenode "HADOOP/bin/hadoop namenode -format"

4.3.4 Start HDFS and MapReduce daemons

Start the Namenode with the script "start-dfs.sh" on the Namenode's node and do the same the JobTracker's with "start-mapred.sh" on the JobTracker's node, for example:

ssh node_configured_as_Namenode "HADOOP/bin/start-dfs.sh"
ssh node_configured_as_JobTracker "HADOOP/bin/start-mapred.sh"

If everything runs correctly, the following script will show us that *Namenode* and *Secondary Namenode* are up on the Namenode's node, *JobTracker*, *Secondary Namenode* on the JobTracker's node and *DataNode*, *TaskTracker* on other nodes. With this small script, we can control it on all nodes (e.g. #nodes = 3):

for i in 'seq 1 3';do echo node0$i; ssh node0$i jps; done

```
node01
16980 SecondaryNameNode
17051 Jps
16861 NameNode
node02
3148 SecondaryNameNode
3316 Jps
3209 JobTracker
node03
22106 TaskTracker
22035 DataNode
22159 Jps
```

4.3.5 Stop HDFS and MapReduce daemons

Run the scripts "stop-dfs.sh", "stop-mapred.sh" corresponding on the Namenode's and JobTracker's node or this script, for example.
for i in 'seq 1 5';do ssh node0$i "killall -9 java"; done

5 Test Preparation

This chapter will show us how to compile a Hadoop's java file via command lines and eclipse with the benchmark program TestDFSIO.java which gives us the writing/reading performance corresponding with the throughput, average I/O rate and I/O rate standard deviation. Alternatively we can use the already built tar-file hadoop--test.jar.*

5.1 Command line (step by step)

Step 1

# download and extract the hadoop tar file, e.g. hadoop-0.18.3
tar -xvf ~/hadoop-0.18.3.tar.gz

# create a new folder and move the hadoop folder in it
mkdir ~/hdfs
mv ~/hadoop-0.18.3 ~/hdfs
cd ~/hdfs

# copy the file TestDFSIO.java out
cp hadoop-0.18.3/src/test/org/apache/hadoop/fs/TestDFSIO.java .

# build the project
cd hadoop-0.18.3/;ant clean;ant
cp -rf hadoop-0.18.3/build/classes/org/ .

# create Manifest.txt
echo "Main-Class: org/apache/hadoop/fs/TestDFSIO" > Manifest.txt

# create makefile with the content

```
CLASSPATH=hadoop-0.18.3/hadoop-0.18.3-core.jar:hadoop-0.18.3/hadoop-0.18.3-examples.jar:hadoop-
0.18.3/hadoop-0.18.3-test.jar:hadoop-0.18.3/hadoop-0.18.3-tools.jar:hadoop-0.18.3/lib/commons-
cli-2.0-SNAPSHOT.jar:hadoop-0.18.3/lib/commons-codec-1.3.jar:hadoop-0.18.3/lib/commons-
httpclient-3.0.1.jar:hadoop-0.18.3/lib/commons-logging-1.0.4.jar:hadoop-0.18.3/lib/commons-logging-
api-1.0.4.jar:hadoop-0.18.3/lib/commons-net-1.4.1.jar:hadoop-0.18.3/lib/jets3t-0.6.0.jar:hadoop-
0.18.3/lib/jetty-5.1.4.jar:hadoop-0.18.3/lib/junit-3.8.1.jar:hadoop-0.18.3/lib/kfs-0.1.3.jar:hadoop-
0.18.3/lib/log4j-1.2.13.jar:hadoop-0.18.3/lib/oro-2.0.8.jar:hadoop-0.18.3/lib/servlet-api.jar:hadoop-
0.18.3/lib/slf4j-api-1.4.3.jar:hadoop-0.18.3/lib/slf4j-log4j12-1.4.3.jar:hadoop-0.18.3/lib/xmlenc-0.52.jar
```

```
all:
        javac TestDFSIO.java -classpath $ (CLASSPATH)
        mv -f *.class org/apache/hadoop/fs/
        rm -f TestDFSIO.jar
        jar cmf Manifest.txt TestDFSIO.jar org/
```

Step 2

# compile IOMapperBase.java and copy it to org/apache/hadooop/fs
javac hadoop-0.18.3/src/test/org/apache/hadoop/fs/IOMapperBase.java
cp hadoop-0.18.3/src/test/org/apache/hadoop/fs/IOMapperBase.class org/apache/hadoop/fs

# compile AccumulatingReducer.java and copy it to org/apache/hadooop/fs
jar xvf hadoop-0.18.3/lib/commons-logging-api-1.0.4.jar
javac hadoop-0.18.3/src/test/org/apache/hadoop/fs/AccumulatingReducer.java
cp hadoop-0.18.3/src/test/org/apache/hadoop/fs/AccumulatingReducer.class org/apache/hadoop/fs

Step 3

Now start the HDFS daemons 18 and now we are able to run the test

# create the TestDFSIO.jar file again because of the new required compiled files
make
# run an example test
hadoop-0.18.3/bin/hadoop jar TestDFSIO.jar -write -fileSize 128 -nrFiles 2

5.2 Eclipse

To build and work Hadoop within Eclipse, follow the next steps:

Step 1: Import project

File > Import > path_to_hadoop. [1]

Step 2: Import the source files

Right mouse on Project [2] > Build Path > Configure Build Path > Source > Link Source > Choose folders you want to import.

Step 3: Import the libraries

[1]In this article it is hadoop-0.18.3
[2]Normally it is MapReduceTools

Right mouse on Project > Build Path > Configure Build Path > Libraries > Add External JARs > Choose jar files you want to import (See the makefile content above for the jar files). [3]

5.3 Testing cluster information

Hardware im common

∗ Two Intel Xeon 2GHz CPUs

∗ Intel Server Board SE7500CW2

∗ 1 GB DDR-RAM

∗ 80GB IDE HDD

∗ CD-ROM Drive

∗ Floppy Disk Drive

∗ Two 100-MBit/s-Ethernet-Ports (out of use)

∗ Two 1-GBit/s-Ethernet-Port (one in use)

∗ 450 Watt Single Power Supply

Network performance results

∗ Measured with netcat TCP

∗ Between 2 nodes

∗ 118 MB/s (one processor with 100% utilized)

- Vmstat results:

procs		memory				swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
# Sender															
1	0	0	38956	114608	838776	0	0	0	0	4004	36170	1	42	57	0
# Receiver															
1	0	0	43232	113540	840048	0	0	0	0	4003	8008	1	32	67	0

∗ 117 MB/s with deactivated tx/rx TCP checksumming

- Vmstat results:

[3]See the make file content above for the required jar files !

procs		memory				swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
# Sender															
1	0	0	41668	113540	840056	0	0	0	0	4004	28655	2	98	0	0
# Receiver															
1	0	0	42072	114608	839248	0	0	0	0	3106	4286	1	15	84	0

∗ 200 MB/s TCP eine Node mit localhost

∗ Master: 393 MB/s mit localhost

- Vmstart results:

procs		memory				swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
1	0	0	8156	1672	425580	0	0	0	0	22969	137692	5	50	40	0

Special Hardware for Master-Node

∗ 80 GB IDE HDD for Home-Folders

∗ Second 1-GBit/s-Ethernet-Interface in use for external interconnection

Special Hardware for Compute-Nodes 01-05

∗ RAID-Controller Promise FastTrack TX2300

∗ RAID0 (Striping): Two 160 GB S-ATAII HDDs

∗ Performance results for RAID Controller
(http://ludwig9.informatik.uni-heidelberg.de/wiki/index.php/Cluster:Raid_Leistungsergebnisse)

5.4 Test process understanding

In this section we will go into the details of the benchmark program TestDFSIO.java in oder to understand how a client creates and submits a MapReduce job in the queue and how the job is processed on the HDFS.

hadoop-0.18.3/bin/hadoop jar TestDFSIO.jar
Usage: TestFDSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile resultFileName] [-bufferSize Bytes]

Step 1: Client creates control files

At first the client receives the input parameters and creates control files on the HDFS depending on the parameter *-nrFiles* (default = 1) with the function *createControlFile(fs, fileSize, nrFiles)* 38 . The names of the control files are default *in_file_test_io_x* (x = 0,1,..,N). They are unique and consist of the file size internally. According to these files the client is able to know about the tasks (map tasks) and input filesize.

Step 2: Client creates and submits a job

The client creates a job using the function *runIOTest(Class<? extends Mapper> mapperClass, Path outputDir)* 39 . Via this function the client collects attributes to create a job, for example: *setInputPaths*, *setOutputPath*, *setMapperClass*, *setReduceClass*, *setNumReduceTasks(1)*, etc.

The design is that the number of the map tasks are overwritten by the number of input files (-nrFiles) 40 and each map task performs the operation completely on one Datanode, which means the file will be written completely on one Datanode. The map function *"Mapper"* used here implements an I/O operation 39 as well as gathers the values of *tasks (number of map tasks), size (filesize), time (executing time), rate and sqrate* of each corresponding map tasks and sends them to the Reducer 40 . The reduced function *"Reducer"* counts all the immediate values and save a reduced output file named *"part-00000"* on the HDFS. The Function *analyzeResult* 42 handles this file and prints out the final values of data throughput, average IO rate and IO rate standard deviation.

After the Reducer receives the outputs of the Mapper, it sums the intermediate values, calculates *Data Throughput (mb/s), Average IO (mb/s), IO rate std deviation*, etc. and creates reduced ouput files on the HDFS according to the number of reduced tasks. We only want to have a single reduced output file on the HDFS consisting all the values we need. So this is the meaning why the developers code the number of reduced tasks equals 1. 39

Furthermore the client collects other attributes via files like *control files*, *hadoop-site.xml*, *hadoop-default.xml*, etc. to create *job.jar, job.xml and job.split*. Here are the meanings of these files:
job.jar includes binary java files for the test, in this case it is for the class TestDFSIO.java.
job.xml includes attributes for the test, e.g. mapred.map.tasks, mapred.reduce.tasks, dfs.block.size, etc.

job.split includes the path to the control files and the java file used for splitting the input file

These files are useful for creating the job. Then the client deletes these files and sends the job 40 into the HDFS queue.

Step 3: Master handles jobs via queue

JobTracker's and Namenode's daemons are threads running in the background on HDFS after we start it 18 . There is one Thread, namely JobInitThread 41 . This thread gets the job in sequence from the queue and handles it. According to the number of MapReduce tasks in the job the JobTracker contacts to the Namenode for the nodes on where it should start the MapReduce tasks. The JobTracker is intelligent to make the job working even the job configuration is bad. For example we have only m map tasks (configured in *"hadoop-site.xml"*), but the number of split data sets is n (n<m). Each map task can only work with one split data set. If the JobTracker starts all of m map tasks, there is "m-n" map tasks which do nothing. So it's wasted and can occur problems on the HDFS. To avoid it the JobTracker sets them equals as default.

Each split data set has usually many files. The map task will call (or create) for each file a Mapper and this Mapper handles this file. It's analog to the reduced task. How the Mapper and Reducer work is already described in *"step 2"*.

6 Test Results

The test was performed on the cluster with nine nodes with the configuration: two nodes for masters (one for Namenode, one for JobTracker) and the remaining nodes are Datanodes. Once again the command:

hadoop-0.18.3/bin/hadoop jar TestDFSIO.jar
Usage: TestFDSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile resultFileName] [-bufferSize Bytes]

Each MapReduce task will run on one Datanode and the task distribution is made by the JobTracker.

Test scenarios: The tests deliver the writing/reading performance with the small (512 MB) / big (2 and 4 GB) data set with the blocksize 64/128 MB

- *Write/Read 512 MB with blocksize 64/128 MB*
- *Write/Read 2 GB with blocksize 64/128 MB*
- *Write/Read 4 GB with blocksize 64/128 MB*

Abbreviation : Tx = test x (for example T1 = the 1st test)

*The measurand belongs to one (map-) tasks. That means if we have five tasks and the measurand is equal to "X" Mb/s we will have altogether 5*X MB/s.*

For all test scenarios the writing/reading performance is tested three times and a median value will be compared to the other to avoid outliers.

To know about the locations of blocks we can run the "fsck" tool on the Namenode, for instance:

hadoop-0.18.3/bin/hadoop fsck path_to_file -blocks -files -locations

6.1 Write

6.1.1 512 MB, Blocksize = 64 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	34.024	34.163	34.247	34.145
Average IO rate (mb/s)	34.024	34.163	34.247	34.145
IO rate std deviation	0.006	0.006	0.008	0.006

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	33.193	32.171	33.865	33.076
Average IO rate (mb/s)	33.242	32.230	33.908	33.127
IO rate std deviation	1.269	1.379	1.231	1.293

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	25.506	23.416	22.390	23.770
Average IO rate (mb/s)	25.506	23.416	22.390	23.771
IO rate std deviation	0.004	0.005	0.004	0.004

6.1.2 2 GB, Blocksize = 64 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	33.075	32.652	32.801	32.842
Average IO rate (mb/s)	33.075	32.652	32.801	32.842
IO rate std deviation	0.002	0.002	0.006	0.003

nrFiles = 5, Replication = 1 [1]

	T1	T2	T3	Mean
Throughput (mb/s)	32.847	32.696	32.839	33.106
Average IO rate (mb/s)	32.896	32.949	32.853	32.899
IO rate std deviation	1.248	0.947	0.677	0.957

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	23.376	24.014	23.140	23.510
Average IO rate (mb/s)	23.376	24.014	23.140	23.510
IO rate std deviation	0.004	0.004	0.005	0.004

6.1.3 4 GB, Blocksize = 64 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	32.141	33.776	30.699	32.205
Average IO rate (mb/s)	32.141	33.776	30.699	32.205
IO rate std deviation	0.006	0.007	0.009	0.007

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	32.428	31.859	31.846	32.044
Average IO rate (mb/s)	32.434	31.892	31.875	32.067
IO rate std deviation	0.446	1.009	0.979	0.811

nrFiles = 1, Replication = 3

[1] This test simulates writing a 10 GB file distributed on 5 data nodes.

	T1	T2	T3	Mean
Throughput (mb/s)	23.065	23.737	23.936	23.579
Average IO rate (mb/s)	23.065	23.737	23.936	23.579
IO rate std deviation	0.005	0.001	0.005	0.003

6.1.4 512 MB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	34.233	34.643	34.162	34.346
Average IO rate (mb/s)	34.233	34.643	34.162	34.346
IO rate std deviation	0.005	0.007	0.005	0.005

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	34.314	34.493	34.454	34.420
Average IO rate (mb/s)	34.361	34.584	34.517	34.487
IO rate std deviation	1.273	1.748	1.455	1.492

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	25.510	22.680	26.571	24.920
Average IO rate (mb/s)	25.510	22.680	26.571	24.920
IO rate std deviation	0.004	0.005	0.004	0.004

6.1.5 2 GB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	33.091	34.425	35.286	34.267
Average IO rate (mb/s)	33.091	34.425	35.286	34.267
IO rate std deviation	0.005	0.002	0.005	0.004

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	34.209	34.282	33.530	34.007
Average IO rate (mb/s)	34.217	34.300	33.533	34.016
IO rate std deviation	0.530	0.779	0.310	0.539

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	23.702	23.646	24.169	23.839
Average IO rate (mb/s)	23.702	23.646	24.169	23.839
IO rate std deviation	0.004	0.004	0.002	0.003

6.1.6 4 GB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	33.815	32.941	32.787	33.181
Average IO rate (mb/s)	33.815	32.941	32.787	33.181
IO rate std deviation	0.003	0.004	0.004	0.003

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	33.481	33.403	33.800	33.561
Average IO rate (mb/s)	33.485	33.421	33.817	33.574
IO rate std deviation	0.351	0.769	0.755	0.625

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	23.888	24.381	24.837	24.368
Average IO rate (mb/s)	23.888	24.381	24.837	24.368
IO rate std deviation	0.002	0.003	0.004	0.003

6.2 Read

6.2.1 512 MB, Blocksize = 64 MB

nrFiles = 1, Replication = 1 [2]

	T1	T2	T3	Mean
Throughput (mb/s)	63.499	62.102	63.563	63.054
Average IO rate (mb/s)	63.499	63.182	63.563	63.414
IO rate std deviation	0.007	0.008	0.006	0.007

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	60.027	60.353	60.504	60.294
Average IO rate (mb/s)	60.111	60.513	60.660	60.428
IO rate std deviation	2.232	3.148	3.087	2.822

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	78.757	65.775	66.090	70.207
Average IO rate (mb/s)	78.757	65.775	66.090	70.207
IO rate std deviation	0.012	0.013	0.011	0.012

6.2.2 2 GB, Blocksize = 64 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	46.452	47.492	47.499	47.147
Average IO rate (mb/s)	46.452	47.492	47.499	47.147
IO rate std deviation	0.010	0.009	0.020	0.013

[2] We can only read if the file exists on HDFS and for correctness the read summary must be equal to the existing file size.

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	45.985	45.740	45.585	45.770
Average IO rate (mb/s)	46.021	45.763	45.605	45.796
IO rate std deviation	1.293	1.026	0.964	1.094

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	67.965	67.983	66.428	67.458
Average IO rate (mb/s)	67.965	67.983	66.428	67.458
IO rate std deviation	0.009	0.012	0.012	0.011

6.2.3 4 GB, Blocksize = 64 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	47.412	47.605	47.137	47.384
Average IO rate (mb/s)	47.412	47.605	47.137	47.384
IO rate std deviation	0.005	0.009	0.009	0.007

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	45.126	45.522	45.009	45.219
Average IO rate (mb/s)	45.155	45.543	45.014	45.237
IO rate std deviation	1.131	0.991	0.437	0.853

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	51.620	54.213	53.245	53.026
Average IO rate (mb/s)	51.620	54.213	53.245	53.026
IO rate std deviation	0.005	0.007	0.010	0.007

6.2.4 512 MB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	64.573	66.261	66.954	65.929
Average IO rate (mb/s)	64.573	66.261	66.954	65.929
IO rate std deviation	0.013	0.007	0.014	0.011

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	66.471	63.228	61.102	63.600
Average IO rate (mb/s)	67.093	63.496	61.410	63.999
IO rate std deviation	6.503	4.237	4.228	4.989

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	63.928	73.764	68.586	68.759
Average IO rate (mb/s)	63.928	73.764	68.586	68.759
IO rate std deviation	0.006	0.0009	0.009	0.0159

6.2.5 2 GB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	45.410	45.569	48.480	46.486
Average IO rate (mb/s)	45.410	45.569	48.480	46.486
IO rate std deviation	0.008	0.010	0.007	0.008

nrFiles = 5, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	45.757	45.590	46.246	45.864
Average IO rate (mb/s)	45.786	45.592	46.254	45.877
IO rate std deviation	1.154	0.331	0.606	0.697

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	67.346	65.009	63.519	65.291
Average IO rate (mb/s)	67.346	65.009	63.519	65.291
IO rate std deviation	0.015	0.008	0.006	0.009

6.2.6 4 GB, Blocksize = 128 MB

nrFiles = 1, Replication = 1

	T1	T2	T3	Mean
Throughput (mb/s)	45.515	45.864	48.113	46.497
Average IO rate (mb/s)	45.515	45.864	48.113	46.497
IO rate std deviation	0.007	0.014	0.007	0.009

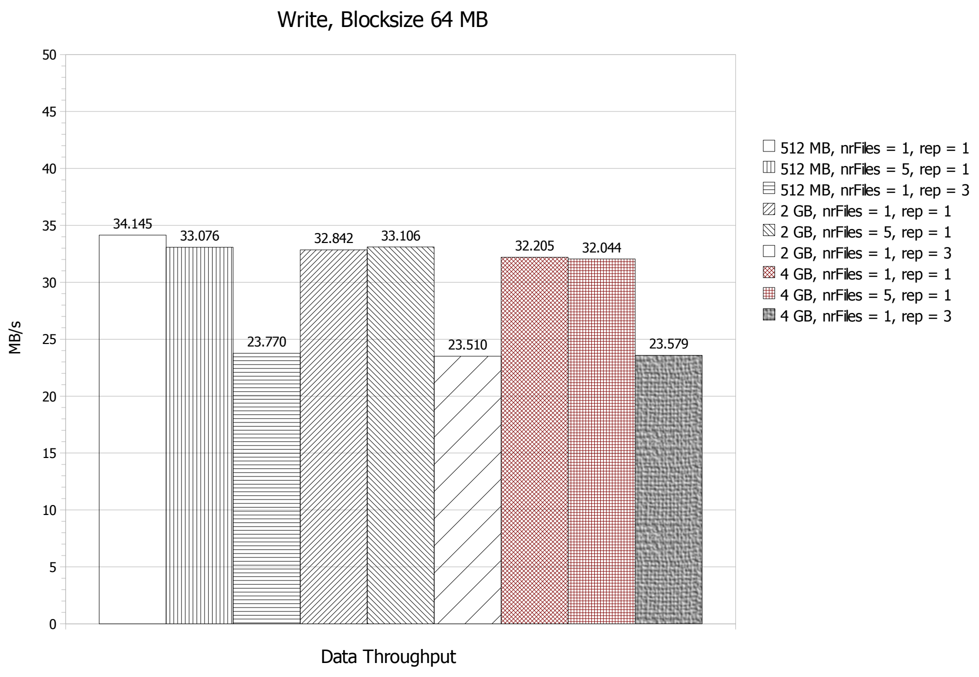
nrFiles = 5, Replication = 1

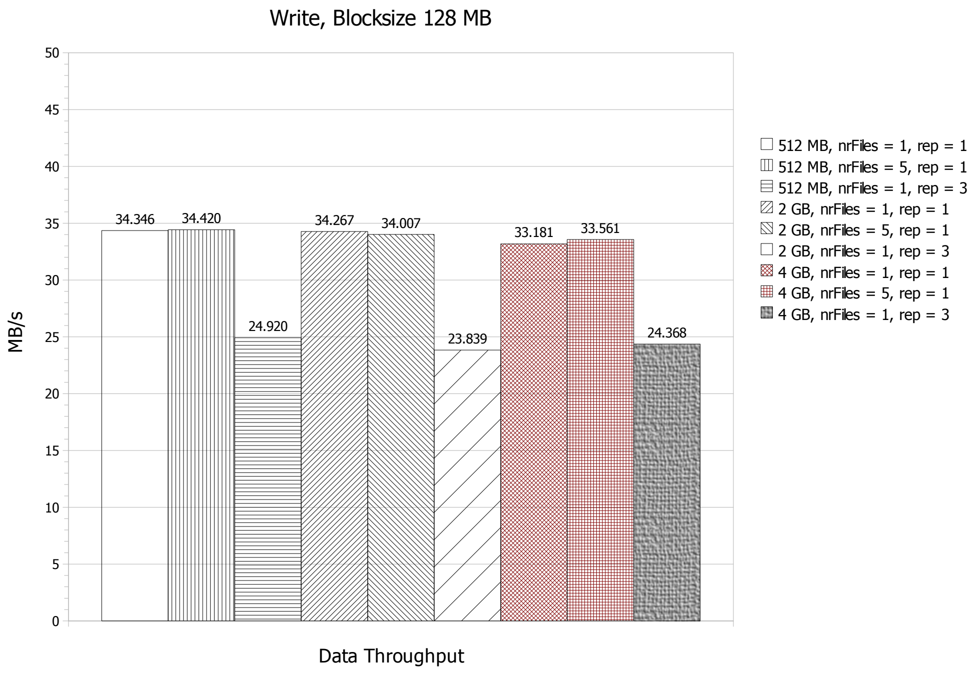
	T1	T2	T3	Mean
Throughput (mb/s)	46.938	46.980	47.500	47.139
Average IO rate (mb/s)	47.048	47.002	47.530	47.193
IO rate std deviation	2.255	1.030	1.174	1.486

nrFiles = 1, Replication = 3

	T1	T2	T3	Mean
Throughput (mb/s)	51.409	54.971	53.930	53.436
Average IO rate (mb/s)	51.409	54.971	53.930	53.436
IO rate std deviation	0.010	0.009	0.002	0.007

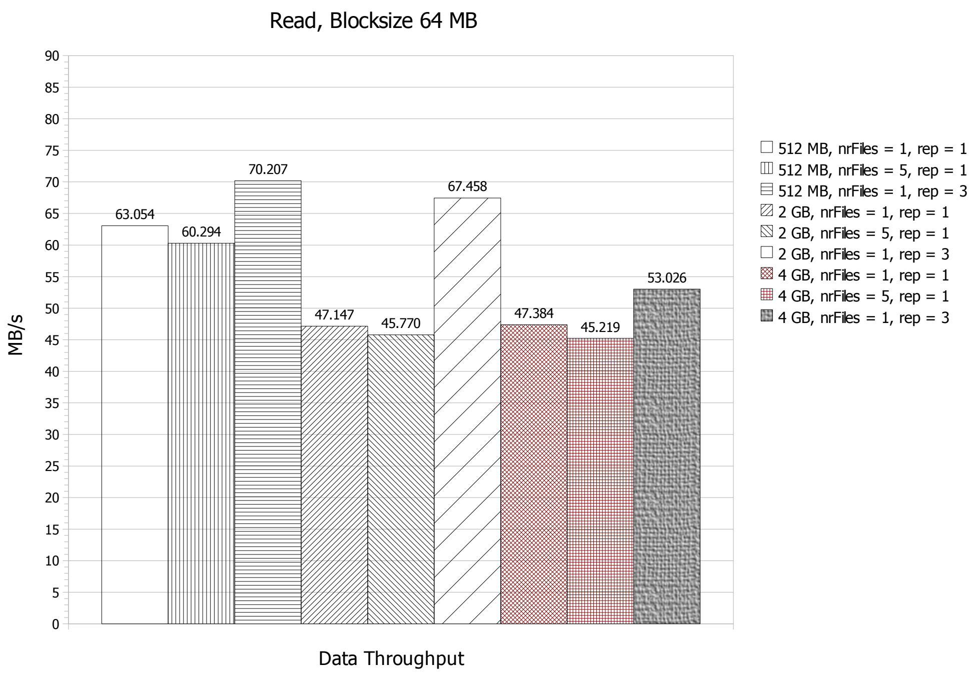
6.3 Write evaluation

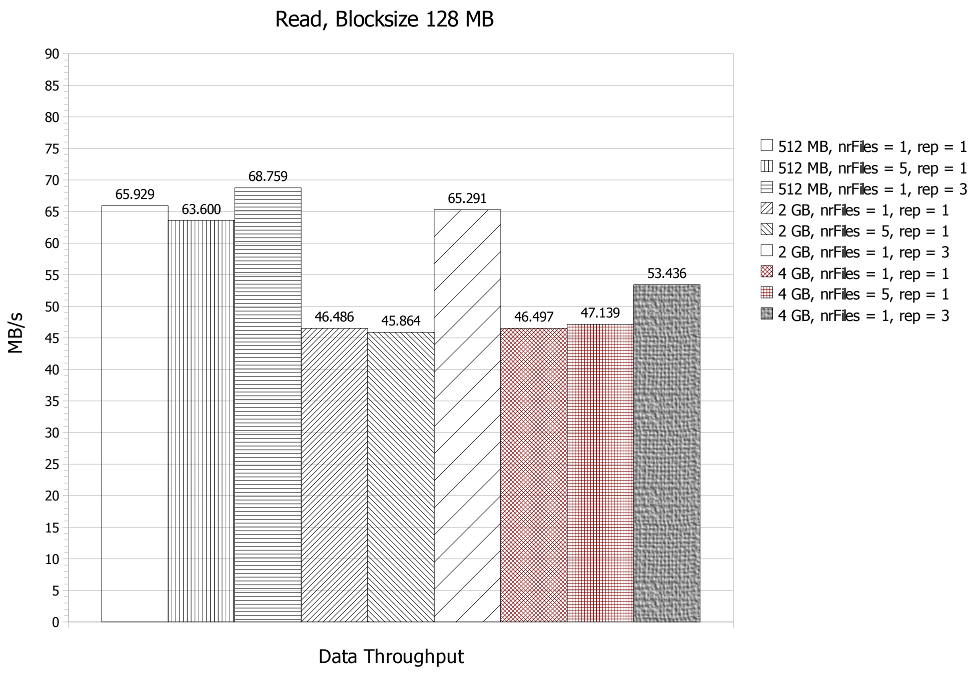




The writing performance with both blocksize 64 MB and 128 MB looks similar to each other. It scales very good with both the small as well as big data set. Writing with a replicated file logically produces a slower performance.

6.4 Read evaluation





The reading performance with both blocksize 64 MB and 128 MB looks similar to each other too and faster than the writing. The reading performance with small files (e.g. 512 MB) is faster than with the big data set (e.g. 2 and 4 GB). Reading with a replicated file logically produces a slower performance.

6.5 Comparison HDFS with local file system performance

The average writing/reading performance measured with the program "*dd*" for the small (512 MB) and big file (4 GB) on the testing cluster is, for instance:

Write X MB/s: dd if=/dev/zero of=test bs=1024k count=X
Read X MB/s: dd if=test of=/dev/null bs=1024k count=X

local FS	512 MB	4 GB
write	47.812	43.122
read	461.375	53.655

Compare with the HDFS performance (nrFiles = 1, rep = 1, Blocksize = 64 MB)

HDFS	512 MB	4 GB
write	34.145	32.205
read	63.054	47.384

Write

The HDFS writing performance is lower than the local file system , for the small and big data set it is circa -28,6% and -25,3%

Read

The HDFS reading performance is lower than the local file system , for the small and big data set it is circa -86,3% and -11.8%

Conclusion

The HDFS reading performance is much lower than the local FS for the small data set, because each node on the testing cluster has 1 GB Ram and the small data set (512 MB) is fit within the Ram. But HDFS is designed for huge data sets, so in this case the HDFS writing/reading performance is lower circa -25,3% / -11.8% than the local FS. Logically it must be lower, since the HDFS is a distributed file system above all over local file system on each node because of the HDFS management and maybe Java IO overhead.

The HDFS reading performance is fine with -11.8% and the significant property of HDFS is that its scalability is very good.

7 Conclusions

This chapter summarizes this article and its results.

Hadoop is a distributed file system and currently being used by many big companies like Yahoo, Goolge and IBM [7] for their applications. This article shows some techniques to work with Hadoop in a non-gui and gui enviroment.

Hadoop is designed for clients, which don't run on a Hadoop daemon itself. If a client performs a writing operation, normally this operation will run on the Namenode and it will split the input data set and spread the split parts across the Datanodes. Otherwise if we want to perform the writing operation for some reason on any Datanodes internally, this operation will only performed locally to avoid congestion on the network.

The benchmark program TestDFSIO.java use the MapReduce concept for writing/reading files. It sets map tasks equal to the value of *-nrFiles* and each map tasks will write/read the data on a Datanode completely. After each map task is done, it collects the values of *tasks (number of map tasks), size (filesize), time (executing time), rate and sqrate* sends them to the Reducer 40. The Reducer counts all the immediate values and saves a reduced output file named *"part-00000"* on the HDFS. Using this file the program computes data throughput, average IO rate and IO rate standard deviation.

The writing performance of Hadoop scales better than reading with small data sets. But it doesn't matter because Hadoop is designed for the batch processing on huge data sets. So in this case it's quite fine with the scalability. Furthermore the writing and reading performance are fast. If we write or read a data set with 20 GB distributed on 5 nodes, we will end up with circa 160 MB/s and 181 MB/s. The more data nodes we have, the faster it is.

In comparision with the local file system on the cluster the HDFS writing/reading performance is lower circa -25,3% / -11.8%. The loss of HDFS perfomance is caused by the HDFS management and maybe Java IO overhead.

Hadoop allows writing/reading parallely on all data nodes like other distributed file system. In addition, with MapReduce it is possible to perform MapReduce operations parallely and flexibly depending on user's purposes.

8 Code References

8.1 createControlFile

23

```
private static void createControlFile(FileSystem fs, int fileSize, int nrFiles) throws IOException {
    LOG.info("creating control file: "+fileSize+" mega bytes, "+nrFiles+" files");
    fs.delete(CONTROL_DIR, true);
    for(int i = 0; i < nrFiles; i++) {
        String name = getFileName(i);
        Path controlFile = new Path(CONTROL_DIR, "in_file_" + name);
        SequenceFile.Writer writer = null;
        try {
            writer = SequenceFile.createWriter(fs, fsConfig, controlFile, UTF8.class, LongWritable.class,
                    CompressionType.NONE);
            writer.append(new UTF8(name), new LongWritable(fileSize));
        } catch(Exception e) {
            throw new IOException(e.getLocalizedMessage());
        } finally {
            if (writer != null)
                writer.close();
            writer = null;
        }
    }
    LOG.info("created control files for: "+nrFiles+" files");
    }
```

8.2 writeTest

```
private static void writeTest(FileSystem fs) throws IOException {
    fs.delete(DATA_DIR, true);
    fs.delete(WRITE_DIR, true);
    runIOTest(WriteMapper.class, WRITE_DIR);
}
```

8.3 runIOTest

23

```
private static void runIOTest(Class<? extends Mapper> mapperClass, Path outputDir) throws IOException {
    JobConf job = new JobConf(fsConfig, TestDFSIO.class);
    FileInputFormat.setInputPaths(job, CONTROL_DIR);
    job.setInputFormat(SequenceFileInputFormat.class);
    job.setMapperClass(mapperClass);
    job.setReducerClass(AccumulatingReducer.class);
    FileOutputFormat.setOutputPath(job, outputDir);
    job.setOutputKeyClass(UTF8.class);
    job.setOutputValueClass(UTF8.class);
    job.setNumReduceTasks(1);
    JobClient.runJob(job);
}
```

8.4 doIO

23

Hier is the implementation of “write” but it’s similar to the read’s one.

```
public static class WriteMapper extends IOStatMapper {
    public WriteMapper() {
        super();
        for(int i=0; i ¡ bufferSize; i++)
            buffer[i] = (byte)(‘0’ + i % 50);
    }
    public Object doIO(Reporter reporter, String name, long totalSize) throws IOException {
        // create file
        totalSize *= MEGA;
        OutputStream out;
        out = fs.create(new Path(DATA_DIR, name), true, bufferSize);
        try {
            // write to the file
            long nrRemaining;
            for (nrRemaining = totalSize; nrRemaining > 0; nrRemaining -= bufferSize) {
                int curSize = (bufferSize < nrRemaining) ? bufferSize : (int)nrRemaining;
                out.write(buffer, 0, curSize);
                reporter.setStatus(”writing ” + name + “@” + (totalSize - nrRemaining) + ”/” + totalSize
                        + “ ::host = ” + hostName);
            }
        } finally {
            out.close();
        }
        return new Long(totalSize);
    }
}
```

8.5 collecStats

```
23
private abstract static class IOStatMapper extends IOMapperBase {
   IOStatMapper() {
      super(fsConfig);
   }
   void collectStats(OutputCollector<UTF8, UTF8> output, String name, long execTime, Object objSize)
                     throws IOException {
      long totalSize = ((Long)objSize).longValue();
      float ioRateMbSec = (float)totalSize * 1000 / (execTime * MEGA);
      LOG.info(”Number of bytes processed = ” + totalSize);
      LOG.info(“Exec time = ” + execTime);
      LOG.info(“IO rate = ” + ioRateMbSec);
      output.collect(new UTF8(“l:tasks”), new UTF8(String.valueOf(1)));
      output.collect(new UTF8(“l:size”), new UTF8(String.valueOf(totalSize)));
      output.collect(new UTF8(“l:time”), new UTF8(String.valueOf(execTime)));
      output.collect(new UTF8(“f:rate”), new UTF8(String.valueOf(ioRateMbSec*1000)));
      output.collect(new UTF8(“f:sqrate”), new UTF8(String.valueOf(ioRateMbSec*ioRateMbSec*1000)));
   }
}
```

8.6 -nrFiles overrides map tasks (JobInProcess.initTasks())

```
23
numMapTasks = splits.length;
maps = new TaskInProgress[numMapTasks];
```

8.7 submitJob (JobClient.submitJob())

```
24 JobStatus status = jobSubmitClient.submitJob(jobId);
if (status != null) {
return new NetworkedJob(status);
} else {
throw new IOException(“Could not launch job”);
}
```

8.8 offerService

```
public void offerService() throws InterruptedException {
    this.expireTrackersThread = new Thread(this.expireTrackers, “expireTrackers”);
    this.expireTrackersThread.start();
    this.retireJobsThread = new Thread(this.retireJobs, “retireJobs”);
    this.retireJobsThread.start();
    this.initJobsThread = new Thread(this.initJobs, “initJobs”);
    this.initJobsThread.start();
    expireLaunchingTaskThread.start();
    this.taskCommitThread = new TaskCommitQueue();
    this.taskCommitThread.start();
    if (completedJobStatusStore.isActive()) {
    completedJobsStoreThread = new Thread(completedJobStatusStore, ”completedjobsStore-housekeeper”);
    completedJobsStoreThread.start();
    }
    this.interTrackerServer.join();
    LOG.info(“Stopped interTrackerServer”);
}
```

8.9 JobInitThread

```
24  class JobInitThread implements Runnable {
    public JobInitThread() {
    }
    public void run() {
      JobInProgress job;
      while (true) {
        job = null;
        try {
          synchronized (jobInitQueue) {
            while (jobInitQueue.isEmpty()) {
              jobInitQueue.wait();
            }
            job = jobInitQueue.remove(0);
          }
          job.initTasks();
        } catch (InterruptedException t) {
          break;
        } catch (Throwable t) {
          LOG.error(“Job initialization failed:\n” + StringUtils.stringifyException(t));
          if (job != null) {
            job.kill();
          }
        }
      }
    }
}
```

8.10 analyzeResult

```
23 private static void analyzeResult( FileSystem fs, int testType, long execTime, String resFileName)
                                    throws IOException {
   Path reduceFile;
   if (testType == TEST_TYPE_WRITE)
      reduceFile = new Path(WRITE_DIR, “part-00000”);
   else
      reduceFile = new Path(READ_DIR, “part-00000”);
   DataInputStream in;
   in = new DataInputStream(fs.open(reduceFile));
   BufferedReader lines;
   lines = new BufferedReader(new InputStreamReader(in));
   long tasks = 0;
   long size = 0;
   long time = 0;
   float rate = 0;
   float sqrate = 0;
   String line;
   while((line = lines.readLine()) != null) {
      StringTokenizer tokens = new StringTokenizer(line, “ \t\n\r\f%”);
      String attr = tokens.nextToken();
      if (attr.endsWith(“:tasks”))
         tasks = Long.parseLong(tokens.nextToken());
      else if (attr.endsWith(“:size”))
         size = Long.parseLong(tokens.nextToken());
      else if (attr.endsWith(“:time”))
         time = Long.parseLong(tokens.nextToken());
      else if (attr.endsWith(“:rate”))
         rate = Float.parseFloat(tokens.nextToken());
      else if (attr.endsWith(“:sqrate”))
         sqrate = Float.parseFloat(tokens.nextToken());
   }
   double med = rate / 1000 / tasks;
   double stdDev = Math.sqrt(Math.abs(sqrate / 1000 / tasks - med*med));
   String resultLines[] = {
                              “—– TestDFSIO —– : ” + ((testType == TEST_TYPE_WRITE) ? “write” :
                                     (testType == TEST_TYPE_READ) ? “read” : “unknown”),
                              “ Date & time: ” + new Date(System.currentTimeMillis()),
                              “ Number of files: ” + tasks,
                              “ Total MBytes processed: ” + size/MEGA,
                              “ Throughput mb/sec: ” + size * 1000.0 / (time * MEGA),
                              “ Average IO rate mb/sec: ” + med,
                              “ IO rate std deviation: ” + stdDev,
                              “ Test exec time sec: ” + (float)execTime / 1000,
   };
   PrintStream res = new PrintStream(new FileOutputStream(new File(resFileName), true));
   for(int i = 0; i ¡ resultLines.length; i++) {
      LOG.info(resultLines[i]);
      res.println(resultLines[i]);
   }
}
```

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. `http://labs.google.com/papers/mapreduce.html`.

[2] Hadoop homepage. Hadoop architecture. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[3] Hadoop homepage. Setup hadoop on a cluster server. `http://hadoop.apache.org/core/docs/current/cluster_setup.html`.

[4] Hadoop homepage. Setup hadoop on a local server. `http://hadoop.apache.org/core/docs/current/quickstart.html`.

[5] Hadoop wiki. Jobtracker process. `http://wiki.apache.org/hadoop/JobTracker`.

[6] Hadoop wiki. Tasktracker process. `http://wiki.apache.org/hadoop/TaskTracker`.

[7] Hadoop wiki. Top companies using hadoop. `http://wiki.apache.org/hadoop/PoweredBy`.