Hadoop Performance Evaluation

Praktikum für Fortgeschrittene

Name: Tien Duc Dinh
Betreuer: Olga Mordvinova, Julian Kunkel
Datum: 04-12-2007

Outline

1. Introduction
   - Motivation
   - Basic notations
1. HDFS Overview
   - Architecture
   - MapReduce
1. HDFS Performance
   - Test Scenarios
   - Write
   - Read
   - Comparison with local FS

What is Hadoop ?

Outline

- 1. **Introduction**
  - o **Motivation**
  - o **Basic notations**
- 2. **Overview**
  - o **Architectur**
  - o **MapReduce**
- 2. **Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

- Hadoop is an open-source, Java-based programming framework
  - Apache project
- supports the processing of large data sets in a distributed computing environment
- was inspired by Google MapReduce and Google File System (GFS)
- currently used by many famous IT enterprises, e.g. Google, Yahoo, IBM

Basic notations

Outline

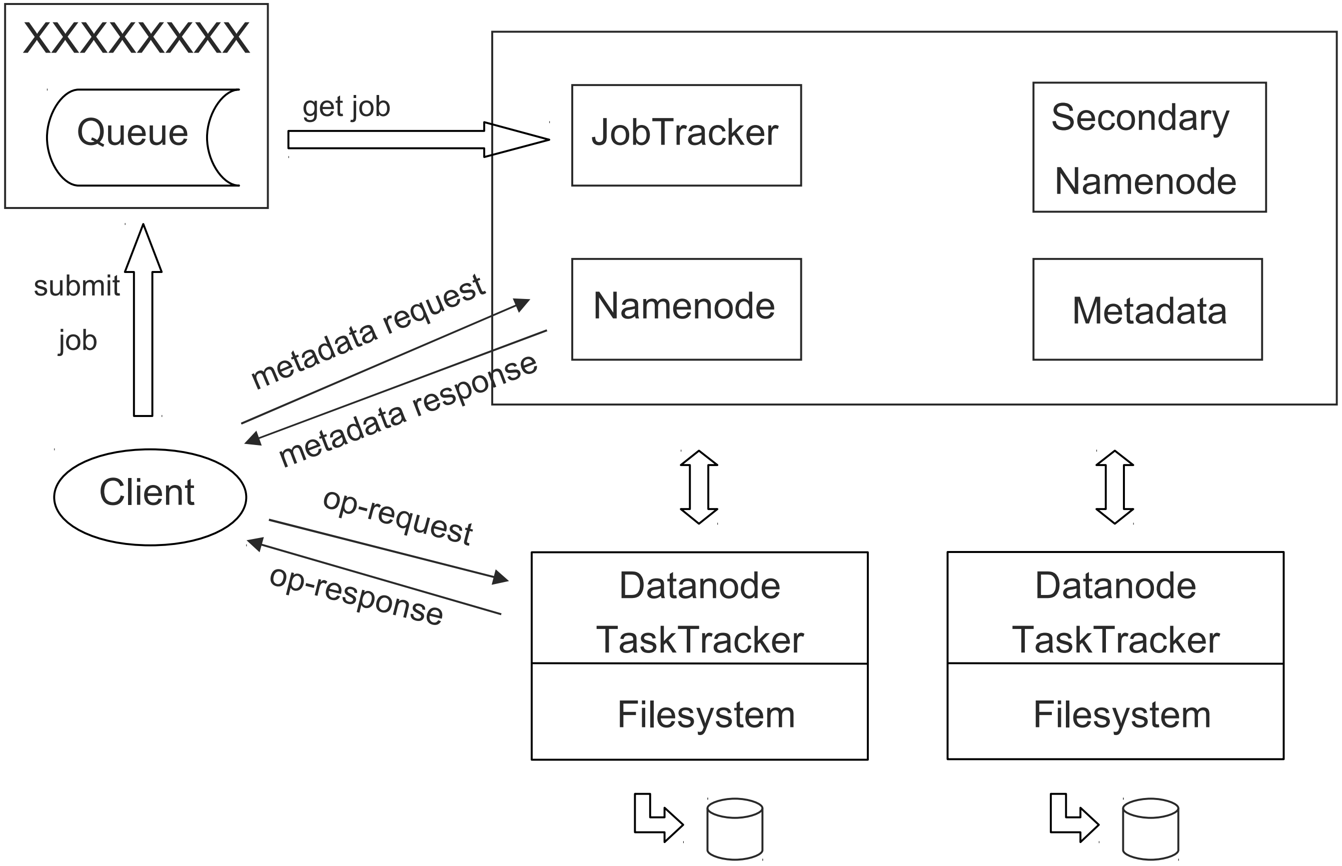
- **1. Introduction**
  - o **Motivation**
  - o **Basic notations**
- **2. Overview**
  - o **Architectur**
  - o **MapReduce**
- **2. Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

- HDFS = Hadoop Distributed File System
- Distributed file system
  - contains mechanisms for job scheduling/execution
  - for instance allows to move jobs to data
- Job/Task = MapReduce job/task
- Metadata
  - data, which consist of other data information
  - e.g. file name, block location
- Block
  - part of a logical file
  - contiguous data stored on one server
  - 64 MB default
  - configurable

HDFS Overview

Outline

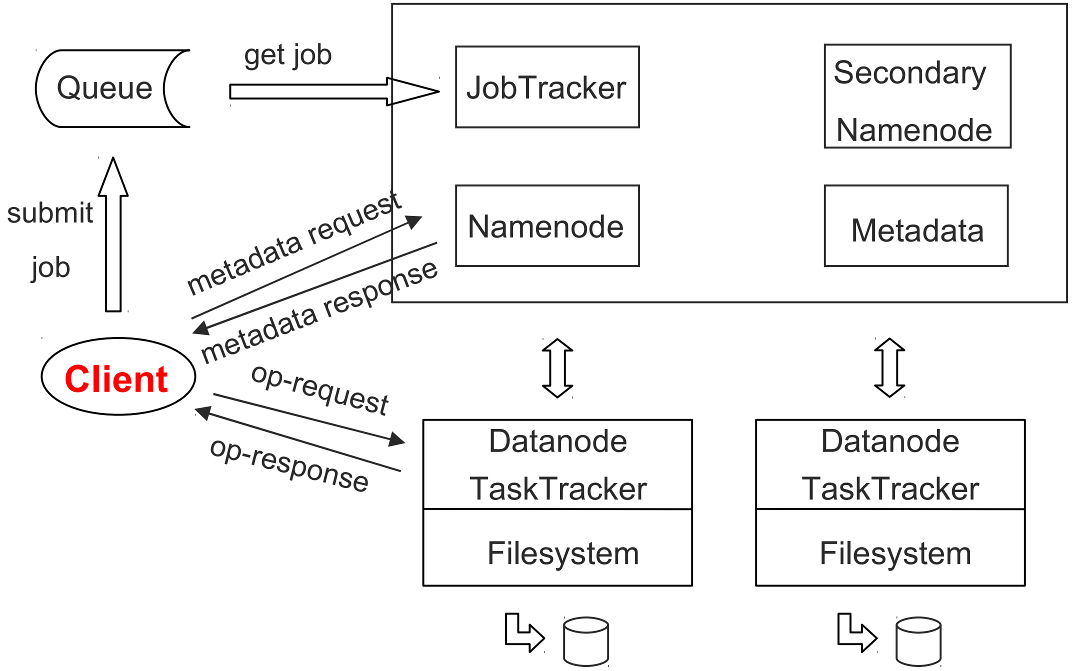
- **1. Introduction**
  - o **Motivation**
  - o **Basic notations**
- **2. Overview**
  - o **Architectur**
  - o **MapReduce**
- **2. Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**



Client

Outline

1. Introduction
  - Motivation
  - Basic notations
2. Overview
  - Architectur
  - MapReduce
2. Performance
  - Test scenarios
  - Write
  - Read
  - Comparison with local FS



- is an api of a HDFS application

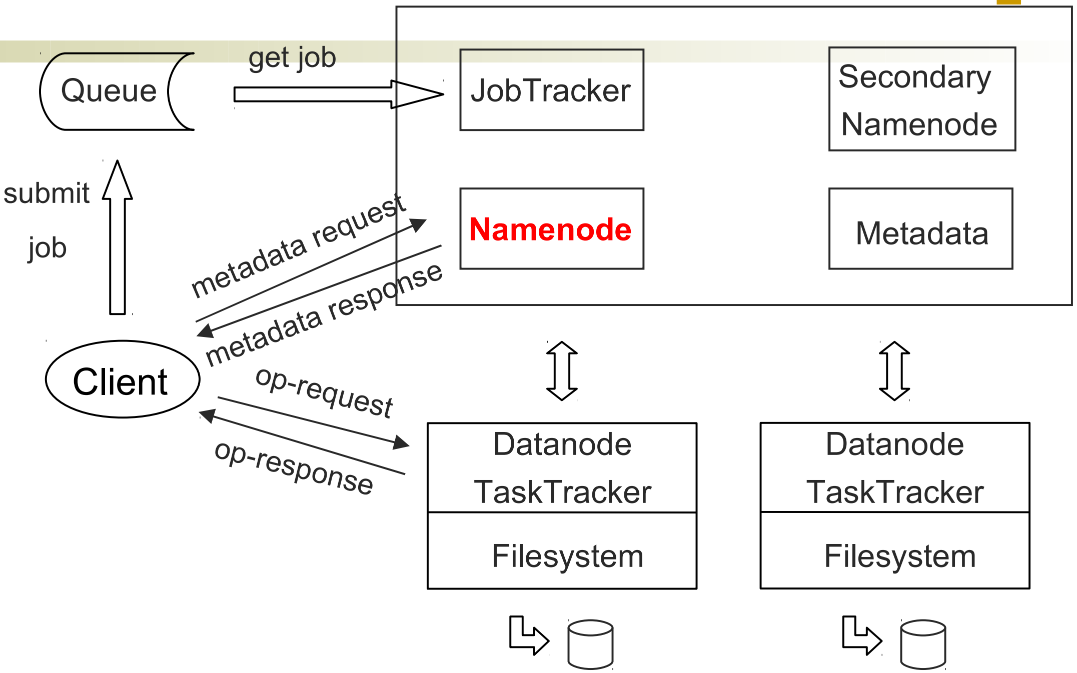
- communicates with the Namenode because of metadata and directly runs the operation on Datanodes

- if it’s a MapReduce operation, client creates an job and send it into the queue. JobTracker handles this queue

Namenode

Outline

- **1. Introduction**
  - **Motivation**
  - **Basic notations**
- **2. Overview**
  - **Architectur**
  - **MapReduce**
- **2. Performance**
  - **Test scenarios**
  - **Write**
  - **Read**
  - **Comparison with local FS**



- is the master server which manages all system metadata like the namespace, access control information, mapping from files to chunks and chunk locations executes file system namespace operations like opening, closing, renaming files and directories

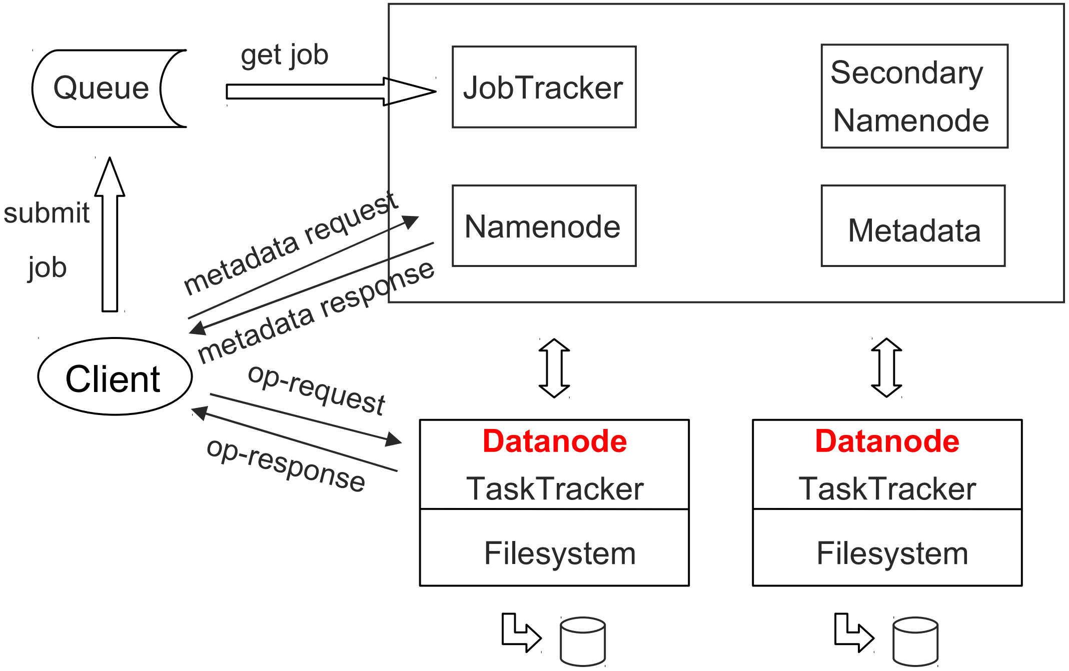
- gives instructions to the Datanodes to perform system operations, e.g. block creation, deletion and replication

- having only one Namenode simplifies the design

Datanode

Outline

- **1. Introduction**
  - **Motivation**
  - **Basic notations**
- **2. Overview**
  - **Architectur**
  - **MapReduce**
- **2. Performance**
  - **Test scenarios**
  - **Write**
  - **Read**
  - **Comparison with local FS**

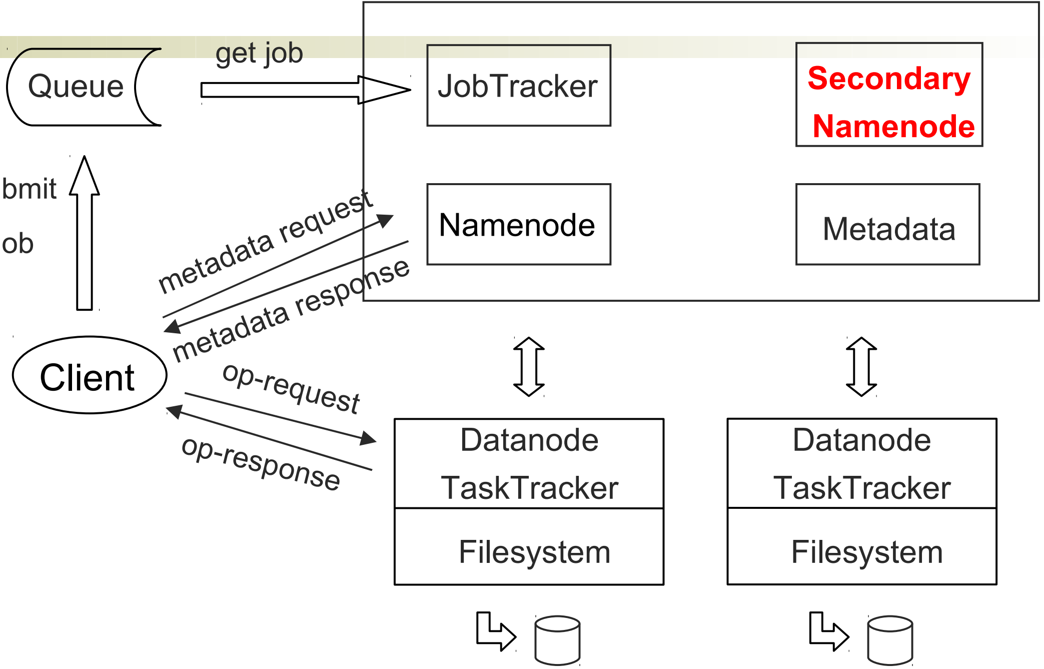


- one per node

- stores HDFS data in its local file system

- performs operations by clients and system operations upon instruction from the Namenode

Secondary Namenode



Outline

1. **Introduction**
   - o **Motivation**
   - o **Basic notations**
2. **Overview**
   - o **Architectur**
   - o **MapReduce**
2. **Performance**
   - o **Test scenarios**
   - o **Write**
   - o **Read**
   - o **Comparison with local FS**

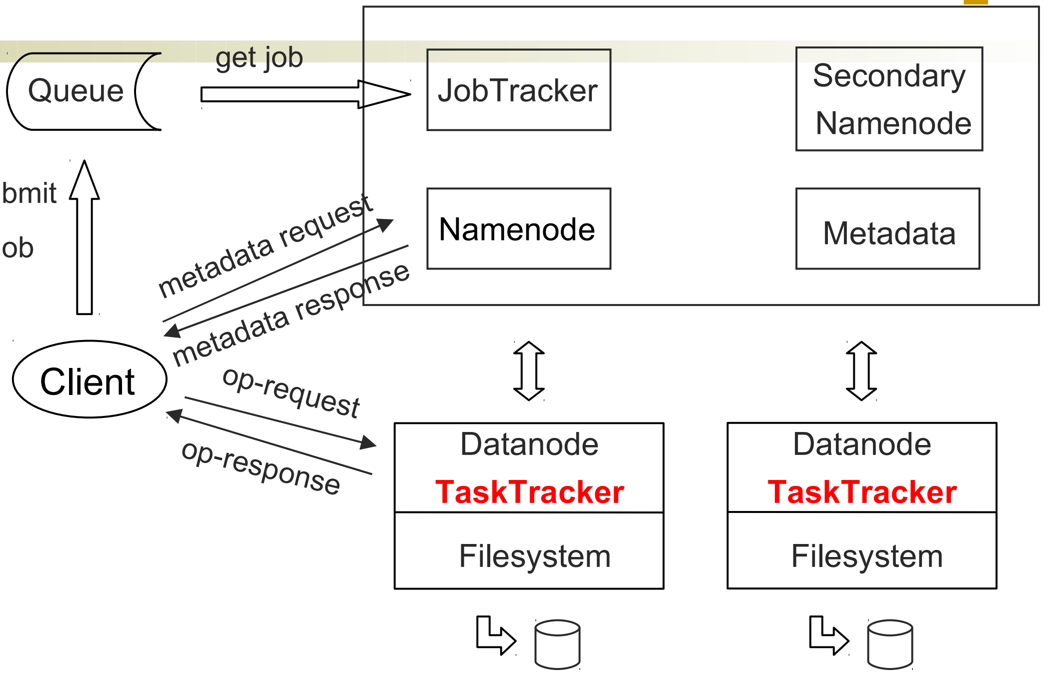
- modifications to the file system are stored as a log file by the Namenode

- while starting up, the Namenode reads the HDFS state from an image file (fsimage) and then applies modifications from the log file

- after the Namenode finished writing the new HDFS state to the image file, it empties the log file

- merges fsimage and the log file periodically and keeps the log size within a limit

TaskTracker



Outline

1. Introduction

- **Motivation**
- **Basic notations**

2. Overview

- **Architectur**
- **MapReduce**

2. Performance

- **Test scenarios**
- **Write**
- **Read**
- **Comparison with local FS**

- is a node in the cluster that accepts MapReduce tasks from the JobTracker

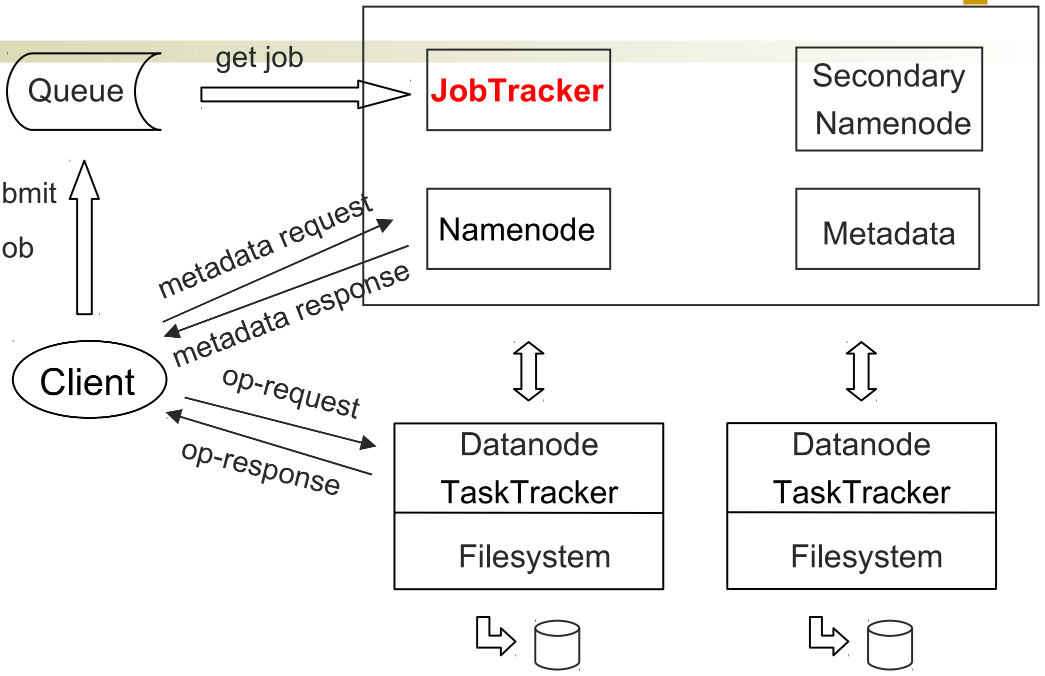
- is configured with a set of slots, these indicate the number of tasks that it can accept

- spawns a separate JVM processes to do the actual work, this helps to ensure that process failure does not take down the TaskTracker

- monitors the processes and reports their state to the JobTracker

- contacts to the JobTracker through heartbeat meassages

JobTracker (1)



Outline

1. **Introduction**
   - **Motivation**
   - **Basic notations**
2. **Overview**
   - **Architectur**
   - **MapReduce**
2. **Performance**
   - **Test scenarios**
   - **Write**
   - **Read**
   - **Comparison with local FS**

- is the MapReduce master

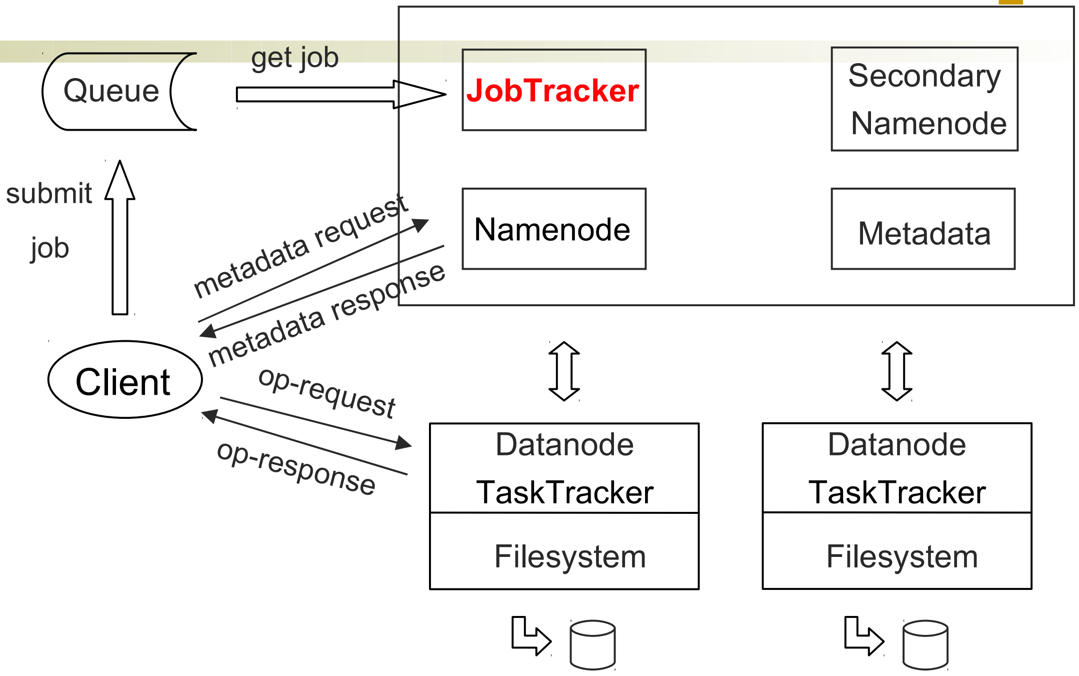
- runs normally on a separate node

- uses a queue for the IO scheduling

- talks to the NameNode to determine the location of the data

- submits the work to the chosen TaskTracker nodes and monitors them through heartbeat meassages in a time interval

JobTracker (2)



Outline

- **1. Introduction**
  - o **Motivation**
  - o **Basic notations**
- **2. Overview**
  - o **Architectur**
  - o **MapReduce**
- **2. Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

- if a task is failed, it may resubmitted elsewhere

- when the work is completed, the JobTracker updates its status

- Client applications can poll the JobTracker for information

- 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 map/reduce job

MapReduce (1)

Outline

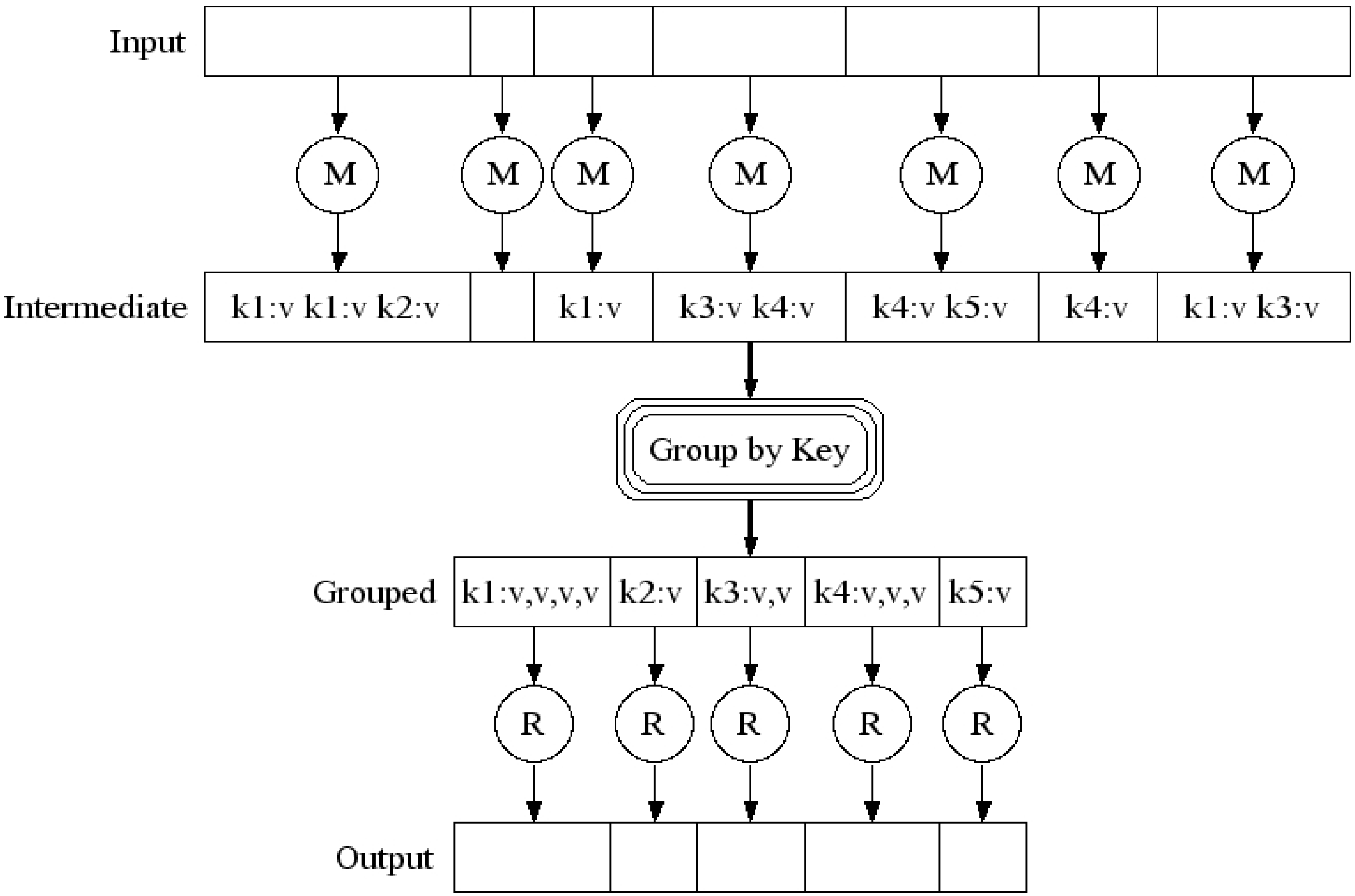
- 1. Introduction
  - Motivation
  - Basic notations
- 2. Overview
  - Architectur
  - MapReduce
- 2. Performance
  - Test scenarios
  - Write
  - Read
  - Comparison with local FS

- Is a programming model and an associated implementation for processing and generating large data sets
- Its functions map and reduce are supplied by the user
- Map
  - process a key/value pair to generate a set of intermediate key/value pairs
  - group together all intermediate values with the same key and pass them to the Reducer
- Reduce
  - XXXXXXXXXXXXXXX

MapReduce (2)

Outline

- 1. **Introduction**
  - o **Motivation**
  - o **Basic notations**
- 2. **Overview**
  - o **Architectur**
  - o **MapReduce**
- 2. **Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**



Input

M M M M M M M

Intermediate

k1:v k1:v k2:v | | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v

Group by Key

Grouped

k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v

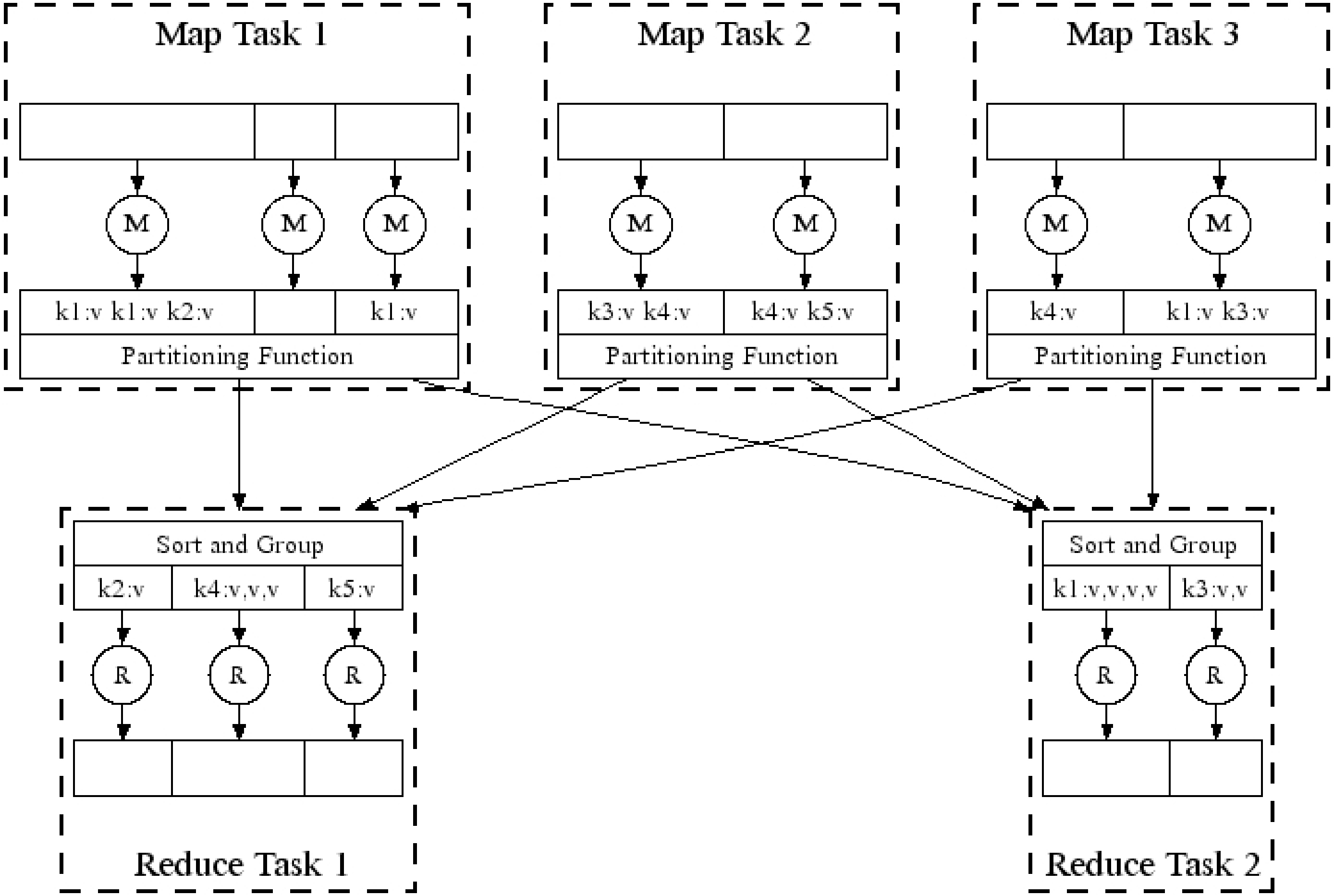
R R R R R

Output

MapReduce (3)

Outline

- 1. Introduction
  - o Motivation
  - o Basic notations
- 2. Overview
  - o Architectur
  - o MapReduce
- 2. Performance
  - o Test scenarios
  - o Write
  - o Read
  - o Comparison with local FS



Example: Word count occurences (1)

Outline

- 1. **Introduction**
  - o **Motivation**
  - o **Basic notations**
- 2. **Overview**
  - o **Architectur**
  - ▶ o **MapReduce**
- 2. **Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

```
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));
```

Example: Word count occurences (2)

Outline

- **1. Introduction**
  - o **Motivation**
  - o **Basic notations**
- **2. Overview**
  - o **Architectur**
  - o **MapReduce**
- **2. Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

- the folder “data” contains 2 files a and b with the following contents:
  - a: Hello World Bye World
  - b: Hello Hadoop Goodbye Hadoop
- the following command will solve this problem

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

- the output looks like

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

Example: Word count occurences (3)

- with MapReduce and e.g. with 2 map and reduce tasks we have for:

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

Map

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

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

Reduce

Outline

1. **Introduction**
   - o **Motivation**
   - o **Basic notations**
2. **Overview**
   - o **Architectur**
   - o **MapReduce**
2. **Performance**
   - o **Test scenarios**
   - o **Write**
   - o **Read**
   - o **Comparison with local FS**

Practise with HDFS Streaming

Outline

- **1. Introduction**
  - o **Motivation**
  - o **Basic notations**
- **2. Overview**
  - o **Architectur**
  - o **MapReduce**
- **2. Performance**
  - o **Test scenarios**
  - o **Write**
  - o **Read**
  - o **Comparison with local FS**

- copy the folder “data” onto the HDFS

```
> hadoop-0.18.3/bin/hadoop fs -put data /
```

- create and run the job with our defined 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”
```

- with 2 reduce tasks we will end up with 2 reduce output files

```
> 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
```

Test scenarios

Outline

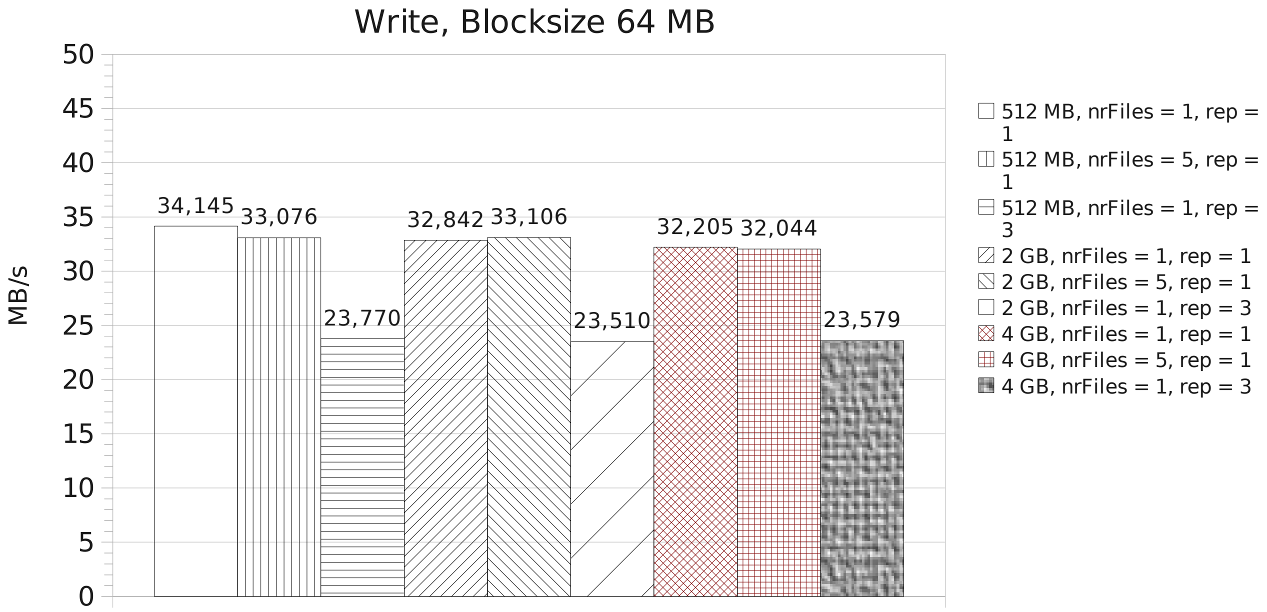
- 1. Introduction
  - o Motivation
  - o Basic notations
- 2. Overview
  - o Architectur
  - o MapReduce
- 2. Performance
  - o Test scenarios
  - o Write
  - o Read
  - o Comparison with local FS

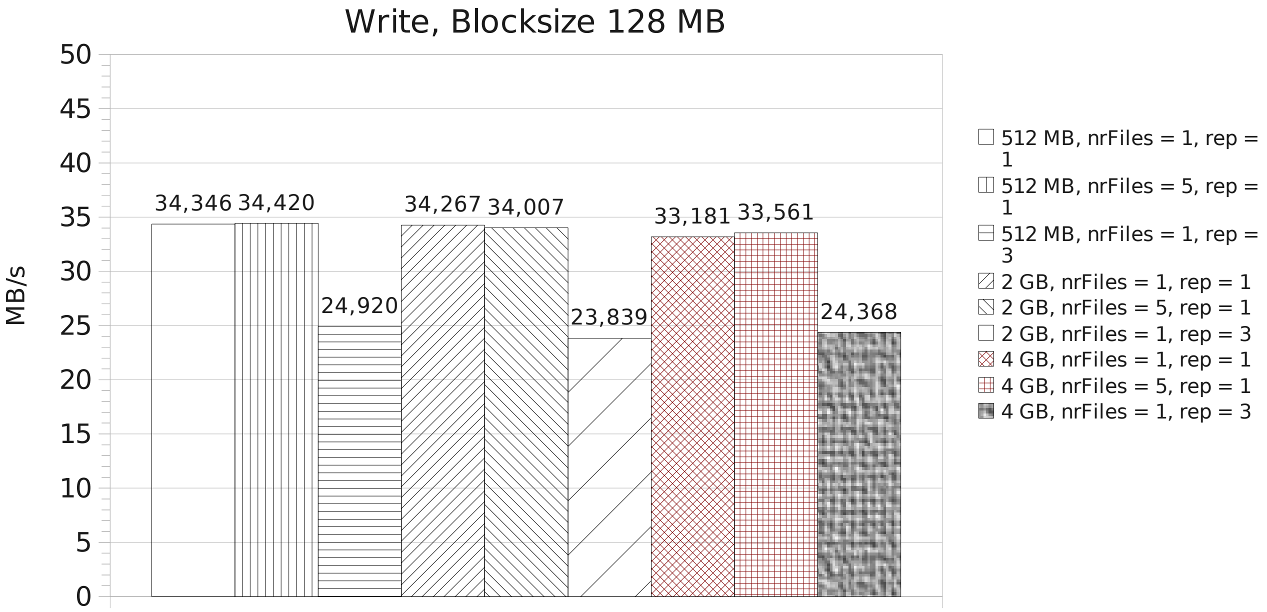
- write/read 512 MB with blocksize 64/128 MB
- write/read 2 GB with blocksize 64/128 MB
- write/read 4 GB with blocksize 64/128 MB

Write

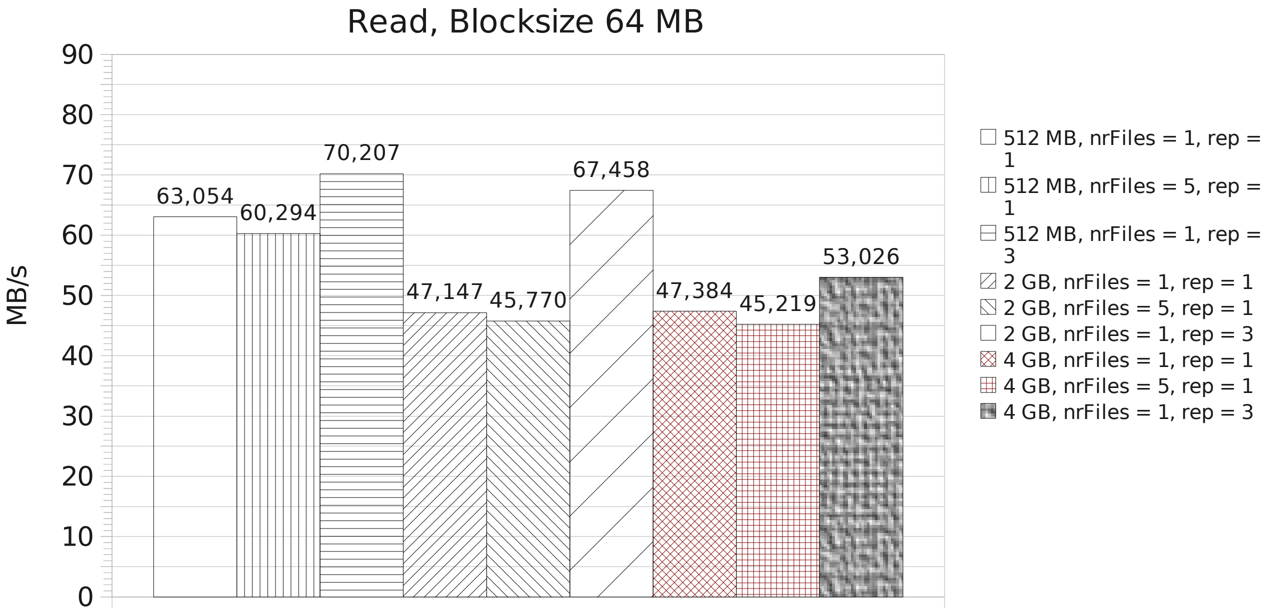
Outline

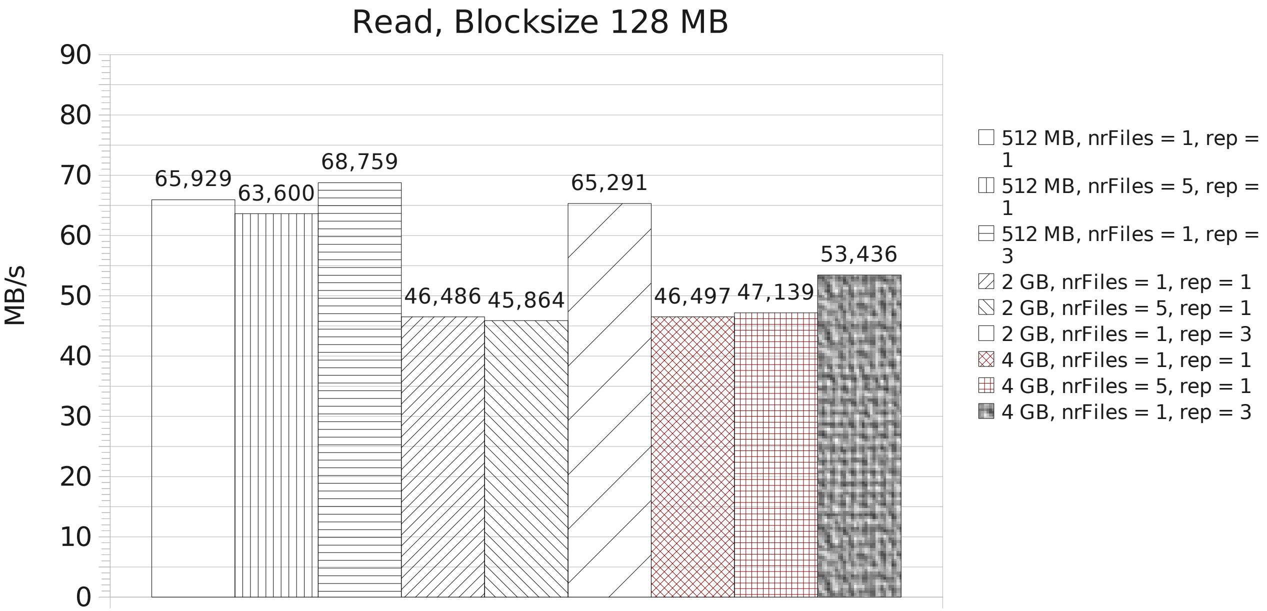
1. **Introduction**
   - o **Motivation**
   - o **Basic notations**
2. **Overview**
   - o **Architectur**
   - o **MapReduce**
2. **Performance**
   - o **Test scenarios**
   - o **Write**
   - o **Read**
   - o **Comparison with local FS**





Read





Outline

1. **Introduction**
   - o **Motivation**
   - o **Basic notations**
2. **Overview**
   - o **Architectur**
   - o **MapReduce**
2. **Performance**
   - o **Test scenarios**
   - o **Write**
   - o **Read**
   - o **Comparison with local FS**

Comparison (1)

Outline

1. Introduction
   - Motivation
   - Basic notations
2. Overview
   - Architectur
   - MapReduce
2. Performance
   - Test scenarios
   - Write
   - Read
   - Comparison with local FS

- compare the HDFS with local FS performance (nrFiles = 1, rep = 1, Blocksize = 64 MB)
- test on the cluster with 9 nodes, each node has 1 GB RAM

HDFS	512 MB	4 GB
write	34.145	32.205
read	63.054	47.384

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

compare	512 MB	4 GB
write	-28,6%	-25,3%
read	-86,3%	-11.8%

Comparison (2)

Outline

- 1. Introduction
  - o Motivation
  - o Basic notations
- 2. Overview
  - o Architectur
  - o MapReduce
- 2. Performance
  - o Test scenarios
  - o Write
  - o Read
  - o Comparison with local FS

- 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
- 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
- HDFS performance losing because of the HDFS management and maybe Java IO overhead

Summary

Outline

1. Introduction
   - Motivation
   - Basic notations
2. Overview
   - Architectur
   - MapReduce
2. Performance
   - Test scenarios
   - Write
   - Read
   - Comparison with local FS

- Hadoop Architecture
- MapReduceJava
- I/O Performance is not too bad

References

Outline

1. **Introduction**
   - o **Motivation**
   - o **Basic notations**
2. **Overview**
   - o **Architectur**
   - o **MapReduce**
2. **Performance**
   - o **Test scenarios**
   - o **Write**
   - o **Read**
   - o **Comparison with local FS**

- http://labs.google.com/papers/mapreduce.html
- http://hadoop.apache.org/core/docs/current/hdfs_design.html
- http://hadoop.apache.org/core/docs/current/cluster_setup.html
- http://hadoop.apache.org/core/docs/current/quickstart.html
- http://wiki.apache.org/hadoop/JobTracker
- http://wiki.apache.org/hadoop/TaskTracker
- http://wiki.apache.org/hadoop/PoweredBy

[End]

Outline

1. Introduction
   - Motivation
   - Basic notations
2. Overview
   - Architectur
   - MapReduce
2. Performance
   - Test scenarios
   - Write
   - Read
   - Comparison with local FS

Danke für Eure Aufmerksamkeit !