

Introduction to Map-Reduce

Vincent Leroy

Sources

- [Apache Hadoop](#)
- [Yahoo! Developer Network](#)
- [Hortonworks](#)
- [Cloudera](#)
- [Practical Problem Solving with Hadoop and Pig](#)
- Slides will be available at
<http://lig-membres.imag.fr/leroyv/>

« Big Data »

- Google, 2008
 - 20 PB/day
 - 180 GB/job (variable)
- Web index
 - 50B pages
 - 15PB
- Large Hadron Collider (LHC) @ CERN :
produces 15PB/year

Capacity of a (large) server

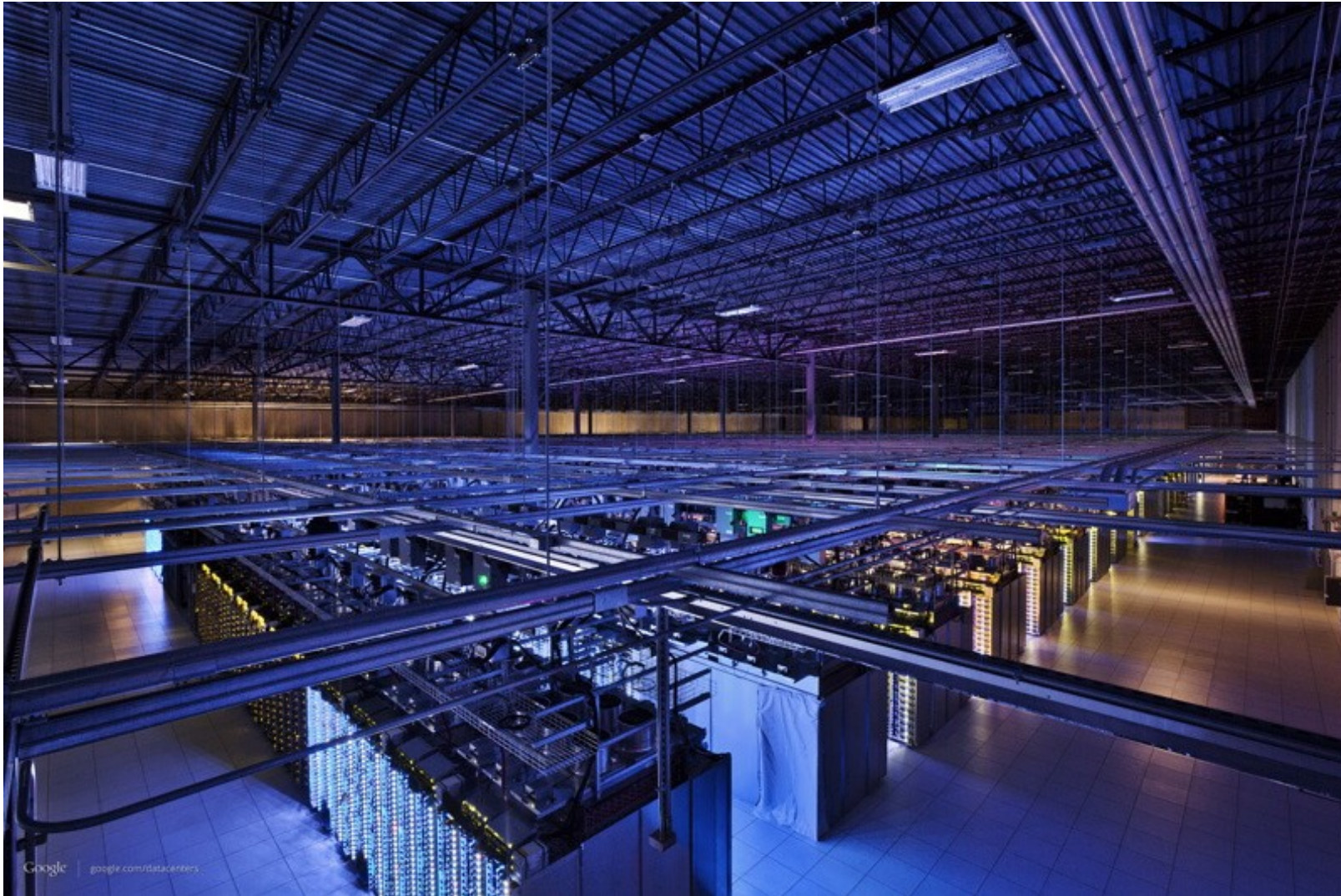
- RAM: 256 GB
- Hard drive capacity: 24TB
- Hard drive throughput: 100MB/s



Solution: Parallelism

- 1 server
 - 8 disks
 - Read the Web: 230 days
- Hadoop Cluster @ Yahoo
 - 4000 servers
 - 8 disks/server
 - Read the Web in parallel: 1h20

Data center Google



Pitfalls in parallelism

- Synchronization
 - Mutex, semaphores ...
- Programming models
 - Shared memory (multicores)
 - Message passing (MPI)
- Difficulties
 - Programming / debugging (deadlocks ...)
 - Optimizing
 - Elasticity (adding/removing machines)
 - Costly (experts)
 - Not reusable

Fault tolerance

- A server fails every few months
- 1000 servers ...
 - MTBF (mean time between failures) < 1 day
- A big job may take several days
 - There will be failures, this is **normal**
 - Computations should finish within a reasonable time
 - You cannot start over in case of failures
- Checkpointing, replication
 - Hard to code by hand

MAP-REDUCE MODEL

What are Map and Reduce ?

- 2 simple functions inspired from functional programming
 - `map (*2) [1,2,3] = [2,4,6]`
 - `reduce (+) [2,4,6] = 12`
- Generic
- Can be instantiated and combined to solve many different problems
- The developer provides the function applied

Map-Reduce: Keys and Values

- Map-Reduce processes key/value pairs
 - Map is applied independently on every key/value pair
 $map(key, value) \rightarrow list(key, value)$
 - Reduce is applied to **all** values associated with the same key
 $reduce(key, list(value)) \rightarrow list(key, value)$
 - The types of keys and values taken as input does not have to be the same as the output

Example: Counting frequency of words

- Input : A file of 2 lines
 - 1, "a b c aa b c"
 - 2, "a bb cc a cc b"
- Output
 - a, 3
 - b, 3
 - c, 2
 - aa, 1
 - bb, 1
 - cc, 2

Word frequency: Mapper

- Map processes a portion (line) of text
 - Split words
 - For each word, count one occurrence
 - Key not used in this example (line number)
- ```
Map(String line, Output output){
 foreach String word in line.split() {
 output.write(word, 1)
 }
}
```

# Word frequency: Reducer

- For each key, Reduce processes all the corresponding values
  - Add number of occurrences
- `Reduce(String word, List<Int> occurrences, Output output){`
  - `int count = 0`
  - `foreach int occ in occurrences {`
    - `count += occ`
  - `}`
  - `output.write(word,count)`
  - `}`

# Execution plan

Map

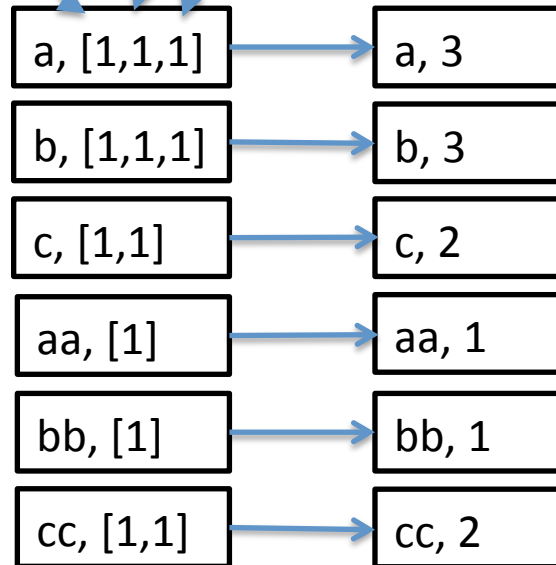
1, "a b c aa b c"

2, "a bb cc a cc b"

a, 1  
b, 1  
c, 1  
aa, 1  
b, 1  
c, 1

a, 1  
bb, 1  
cc, 1  
a, 1  
cc, 1  
b, 1

Reduce



# **HDFS : DISTRIBUTED FILE SYSTEM**



# Random vs Sequential disk access

- Example
  - DB 100M users
  - 100B/user
  - Alter 1% records
- Random access
  - Seek, read, write: 30mS
  - 1M users → 8h20
- Sequential access
  - Read ALL Write ALL
  - 2x 10GB @ 100MB/S → 3 minutes

→ It is often faster to read all and write all sequentially

# Distributed File System (HDFS)

- Goal
  - Fault tolerance (redundancy)
  - Performance (parallel access)
- Large files
  - Sequential reads
  - Sequential writes
- “in place” data processing
  - Data is stored on the machines that process it
    - Better usage of machines (no dedicated *filer*)
    - Less network bottlenecks (better performance)

# HDFS model

- Data organized in files and directories
  - mimics a standard file system
- Files divided in blocks (default: 64MB) spread on servers
- HDFS reports the data layout to the Map-Reduce framework
  - If possible, process data on the machines where it is already stored

# Fault tolerance

- File blocks replicated (default: 3) to tolerate failures
- Placement according to different parameters
  - Power supply
  - Network equipment
  - Diverse servers to increase the probability of having a “close” copy
- Checksum of data to detect corrupter blocks (also available in modern file systems)

# Master/Worker architecture

- *A master*, the NameNode
  - Manage the space of file names
  - Manages access rights
  - Supervise operations on files, blocks ...
  - Supervise the *health* of the file system (failures, load balance...)
- Many (1000s) slaves, the DataNodes
  - Store the data (blocks)
  - Perform read and write operations
  - Perform copies (replication, ordered by the NameNode)

# NameNode

- Stores the metadata of each file and block (*inode*)
  - File name, directory, blocks associated, position of these blocks, number of replicas ...
- Keeps all in main memory (RAM)
  - Limiting factor = number of files
  - 60M objects in 16GB

# DataNode

- Manage and monitor the state of blocks stored on the host file system (often Linux)
- Directly accessed by the clients
  - data never transit through the NameNode
- Send *heartbeats* to the NameNode to show that the server has not failed
- Report to the NameNode if blocks are corrupted

# Writing a file

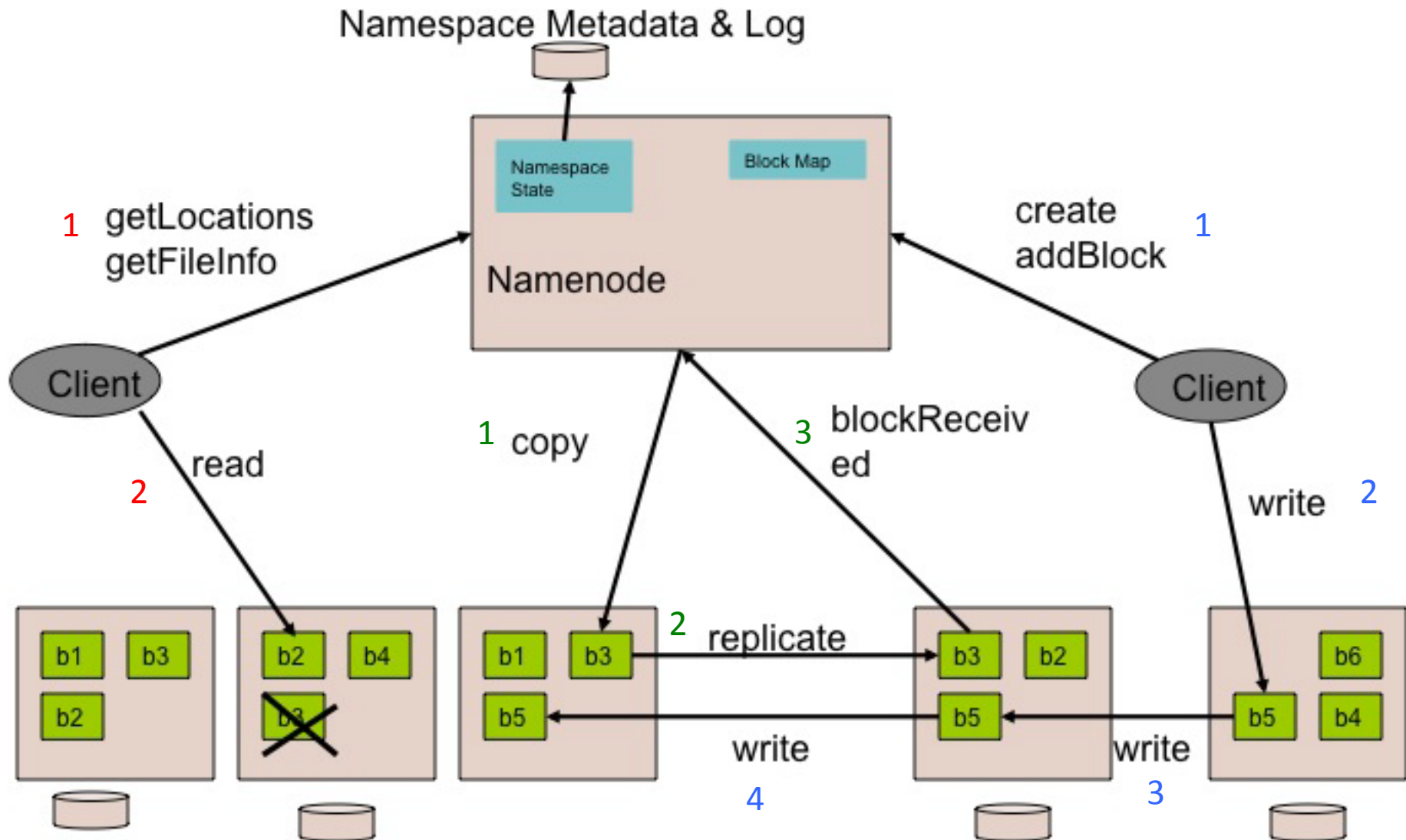
- The client sends a query to the NameNode to create a new file
- The NameNode checks
  - Client authorizations
  - File system conflicts (existing file ...)
- NameNode chooses DataNodes to store file and replicas
  - DataNodes “pipelined”
- Blocks are allocated on these DataNodes
- Stream of data sent to the first DataNode of the pipeline
- Each DataNode forwards the data received to the next DataNode in the pipeline



# Reading a file

- Client sends a request to the NameNode to read a file
- NameNode checks the file exists and builds a list of DataNodes containing the first blocks
- For each block, NameNode sends the address of the DataNodes hosting them
  - List ordered wrt. Proximity to the client
- Client connects to the closest DataNode containing the 1<sup>st</sup> block of the file
- Block read ends:
  - Close connection to the DataNode
  - New connection to the DataNode containing the next block
- When all blocks are read:
  - Query the NameNode to retrieve the following blocks

# HDFS Structure



# HDFS commands (directories)

- Create directory dir  
*\$ hadoop dfs -mkdir /dir*
- List HDFS content  
*\$ hadoop dfs -ls*
- Remove directory dir  
*\$ hadoop dfs -rmr /dir*

# HDFS commands (files)

- Copy local file toto.txt to HDFS dir/  
*\$ hadoop dfs -put toto.txt dir/toto.txt*
- Copy HDFS file to local disk  
*\$ hadoop dfs -get dir/toto.txt ./*
- Read file /dir/toto.txt  
*\$ hadoop dfs -cat /dir/toto.txt*
- Remove file /dir/toto.txt  
*\$ hadoop dfs -rm /dir/toto.txt*

# **APACHE HADOOP: MAPREDUCE FRAMEWORK**

# Objective of the Map-Reduce framework

- Provide a simple and generic programming model: map and reduce
- Deploy execution automatically
- Provide fault tolerance
- Scalability to thousands of machines
- Performance is important but not the priority
  - What's important is that jobs finish within reasonable time
  - If it's too slow, add servers!  
*Kill It With Iron (KIWI principle)*

# What does the developer do?

- Provide the functions performed by Map and Reduce (Java, C++)
  - Application dependent
- Defines the data types (keys / values)
  - If not standard (Text, IntWritable ...)
  - Functions for seralization
- That's all.

# Imports

```
import java.io.IOException ;
import java.util.* ;

import org.apache.hadoop.fs.Path ;
import org.apache.hadoop.io.IntWritable ;
import org.apache.hadoop.io.LongWritable ;
import org.apache.hadoop.io.Text ;
import org.apache.hadoop.mapreduce.Mapper ;
import org.apache.hadoop.mapreduce.Reducer ;
import org.apache.hadoop.mapreduce.JobContext ;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat ;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat ;
import org.apache.hadoop.mapreduce.Job ;
```

**Do not use the old mapred API!**



# Mapper

```
// input key type, input value type, output key type,
output value type
public class WordCountMapper extends Mapper<LongWritable,
Text, Text, IntWritable> {

 @Override
 protected void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException
 {
 for (String word : value.toString().split("\\s+")) {
 context.write(new Text(word), new IntWritable(1));
 }
 }
}
```

# Reducer

```
// input key type, input value type, output key type,
output value type
public class WordCountReducer extends Reducer<Text,
IntWritable, Text, LongWritable> {

 @Override
 protected void reduce(Text key, Iterable<IntWritable>
values, Context context) throws IOException,
InterruptedException {
 long sum = 0;
 for (IntWritable value : values) {
 sum += value.get();
 }
 context.write(key, new LongWritable(sum));
 }
}
```

# Main

```
public class WordCountMain {
 public static void main(String [] args) throws Exception {
 Configuration conf = new Configuration();
 String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
 Job job = Job.getInstance(conf, "word count");
 job.setJarByClass(WordCountMain.class);
 job.setMapOutputKeyClass(Text.class);
 job.setMapOutputValueClass(IntWritable.class);
 job.setOutputKeyClass(Text.class);
 job.setOutputValueClass(LongWritable.class);
 job.setMapperClass(WordCountMapper.class);
 job.setReducerClass(WordCountReducer.class);
 job.setInputFormatClass(TextInputFormat.class);
 job.setOutputFormatClass(TextOutputFormat.class);
 FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
 FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

# Terminology

- MapReduce program = job
- Jobs are submitted to the JobTracker
- A job is divided in several tasks
  - A Map is a task
  - A Reduce is a task
- Tasks are monitored by TaskTrackers
  - A slow task is called a straggler
- In MapReduce, barrier between map and reduce (shuffle & sort)
  - Need to wait for the slowest map before beginning to reduce

# Job execution

- *\$ hadoop jar wordcount.jar org.myorg.WordCount inputPath(HDFS) outputPath(HDFS)*
- Check parameters
  - Is there an output directory ?
  - Does it already exist ?
  - Is there an input directory ?
- Compute splits
- The job (MapReduce code), its configuration and splits are copied with a high replication
- Create an object to follow the progress as the tasks are created by the JobTracker
- For each split, create a Map
- Create default number of reducers

# Tasktracker

- TaskTracker sends a periodic signal to the JobTracker
  - Show that the node still functions
  - Tell whether the TaskTracker is ready to accept a new task
- A TaskTracker is responsible for a node
  - Fixed number of slots for map tasks
  - Fixed number of slots for reduce tasks
  - Tasks can be from different jobs
- Each task runs on its own JVM
  - Prevents a task crash to crash the TaskTracker as well

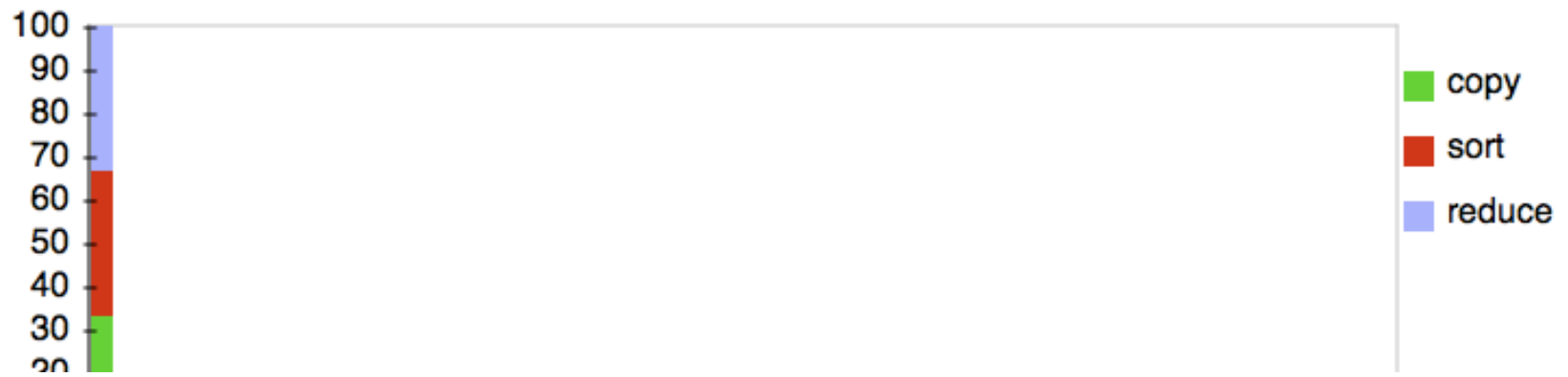
# Job Progress

- A Map task reports on its progress, i.e. amount of the split processed
- For a reduce task, 3 states
  - copy
  - sort
  - reduce
- Report sent to the TaskTracker
- Every 5 seconds, report forwarded to the JobTracker
- User can see the JobTracker state through Web interface

# Progress



Reduce Completion Graph - [close](#)





# End of Job

- Output of each reducer written to a file
- Job tracker notifies the client and writes a report for the job

14/10/28 11:54:25 INFO mapreduce.Job: Job job\_1413131666506\_0070 completed successfully

## Job Counters

Launched map tasks=392

Launched reduce tasks=88

Data-local map tasks=392

[...]

## Map-Reduce Framework

Map input records=622976332

Map output records=622952022

Reduce input groups=54858244

Reduce input records=622952022

Reduce output records=546559709

[...]

# Server failure during a job

- Bug in a task
  - task JVM crashes → TaskTracker JVM notified
  - task removed from its slot
- Task become unresponsive
  - timeout after 10 minutes
  - task removed from its slot
- Each task may be re-run up to N times (default 7) in case of crashes

# Combiner

- Potential problem of a Map function: many key/value pairs in the output
- These pairs are copied to the reducer over the network  
→ costly
- Combiner: mini-reducer executed at the output of a map to reduce the number of pairs
- Combiner input types = Combiner output types = Map output type
- Combiner may be used by Hadoop
  - The correctness of the program should not depend on it
- `conf.setCombiner(...)`

# Combiner

Map

1, "a b c aa b c"

a, 1  
b, 1  
c, 1  
aa, 1  
b, 1  
c, 1

a, 1  
b, 2  
c, 2  
aa, 1

2, "a bb cc a cc b"

a, 1  
bb, 1  
cc, 1  
a, 1  
cc, 1  
b, 1

a, 2  
bb, 1  
cc, 2  
b, 1

Combiner

Reduce

a, [1,2] → a, 3

b, [2,1] → b, 3

c, [2] → c, 2

aa, [1] → aa, 1

bb, [1] → bb, 1

cc, [2] → cc, 2