

Hadoop 30088

Ubiquitous Hadoop

Course website:

<https://ucsc-extension.instructure.com/login/canvas>

Announcements

Lecture materials

Homework assignments

VM FAQs - on setting up and using your development environment

Messages

Quiz called “Express yourself” on course website

- **Used for groups and bridges**
- **Finish the quiz over the weekend**
- **There are no “wrong” answers**

Course overview

Lecture 1: Introduction to Hadoop

Lectures 2-3: Understanding the Map Reduce method

Lecture 4: Hadoop IO (including Flume, Kafka and Sqoop)

Lecture 5: Advanced Map Reduce

Lecture 6-7: SQL interfaces for Hadoop jobs - Hive and SparkSQL

Lecture 8: NoSQL databases for Hadoop - HBase and Cassandra

Lecture 9-10: Introduction to analytics

A Java-centric approach

10 lectures
VM running on VirtualBox
Spark Community Edition

Practice

- Practice exercises for each lecture
- *Practice exercises are not graded*

4 homework assignments

- Late policy: *no more than 1 week late*
- More than 1 week late? - *50% penalty*
- *No assignments accepted after the final class.*

Overview

Hadoop on a VM

- runs on VirtualBox
- Linux OS

Eclipse projects on VM

- practice exercises
- examples from class
- homework starting points

Open-source installed on VM

- Hadoop core and HDFS
- MapReduce and Spark engines
- Hive, Pig, Hue, Zookeeper, HBase...

VM development environment

Today's agenda

Overview of hardware

Overview of daemons

Deep dive into HDFS processing

First exposure to Map Reduce programming

Hadoop in context

Very large Hadoop setups

- Yahoo - 40,000 servers for production
 - 39 million jobs MONTHLY.
 - 100,000 transactions per second
- Facebook - realtime message processing
- Twitter - Exabyte of data in multiple datacenters and multiple clusters
- Ebay - processes all account data daily on Hadoop (2.4 billion accounts)
- Uber - uses Hadoop/spark for mappings, fraud detection, machine learning and data science
- LinkedIn - processes 1.3 TB message/day with Samza, Kafka and Databus

Typical Hadoop setup

40-100 machines with ~ 1 PB of data

Each machine costs \$8-10K, holds 10 TB of data

Use:

- Hadoop MapReduce (MR2) for large jobs
- Spark for smaller jobs

Ecosystem: MR2, Spark, Hive, Pig, HBase/Cassandra

Usually use a Vendor provided setup

Usually complemented with a relational database

Who *doesn't* use Hadoop: REALLY big data

Usage is low-profile and very secure

- satellite images
- central clearing houses
- market makers (HFT)
- VISA, MasterCard and Discover
- military theatre operations
- NSA

Characterized by deep load problems, speed of light issues

So, ... jobs?

Today: LinkedIn shows

- 13,000 jobs in the US
- 2,100 jobs in the bay area

Indeed is similar.

Big companies: IBM, Amazon, Twitter, Microsoft, LinkedIn, Yahoo, Facebook, Apple, Visa, Uber...

Cloud solutions: Altiscale, Amazon (AWS), Google Cloud, IBM BigData, Microsoft Azure, and Rackspace

Hadoop distributors and investors



What is Hadoop?

Many misconceptions about Hadoop

- Seems very complex
- Strange name - new and without meaning
 - **Hadoop is software** originally by Doug Cutting
 - Named after his son's stuffed elephant
- It doesn't help that people talk about a “cloud”.
 - Definition of a cloud:
 - <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>

What is Hadoop?

- Hadoop is a Java software package.
 - Download it onto a machine as a tarball
 - Extract it to any directory you want
 - Set up some configuration files
 - Start up a few Java processes called Hadoop's daemons
 - Hadoop's daemons run on immortal JVMs
 - Daemons live as long as the machine is up

2700 Java classes

- Hadoop jobs:
 - extending Hadoop classes
 - using Hadoop APIs
- we will cover 100+ Hadoop classes

300,000+ lines of code

800 configuration
properties

Hadoop core software

Hadoop's main components

MR 2

MapReduce-based
compute engine

Spark

MapReduce-based
compute engine

Hadoop core

HDFS

Hadoop Distributed
File system

YARN

Job Master

Analytic tools on Hadoop

MR2

- Machine learning - Mahout
- SQL interface - Hive
- Graphs - Giraph
- R - R on Hadoop

Spark

- Machine learning - MLLib
- SQL interface - Hive or SparkSQL
- Graphs - GraphX
- R - SparkR

Alternatives in analysis

Hadoop - it's about analysis

- data may not be big... to start - but repository can grow easily
- the data is easy to load
- can use almost any language to read and analyze the data
 - Java on Spark and MR2
 - Scala or Python on Spark
 - SQL on Hive and SparkSQL
 - Can use Python, Ruby, Perl, C/C++, Java or Scala via Hadoop streaming

terrible
name

We will learn about only part of the Ecosystem

- Intro to HDFS and YARN - what they do and how to use them
- MR2 - the more verbose, more transparent map-reduce engine
 - mapping and reducing
 - partitioning, combining
 - shuffle-sort: what it does
- Spark - the faster map-reduce engine
 - Introduction to MLlib for Spark
 - Introduction to Spark Streaming
- Hadoop IO - Sqoop/Flume/Kafka
- Hive - a SQL interface that secretly runs map-reduce jobs
- HBase - a NoSQL database that runs on its own cluster but uses HDFS

Never read the “news” about Hadoop.

An example of, well ...:

“Where once big data processing was practically synonymous with MapReduce," he said in an email, "you are now seeing frameworks like Spark, Storm, Giraph, and others providing alternatives that allow you to select the approach that is right for the analytic problem."

MapReduce is a paradigm - a method for parallel processing.

Spark does MapReduce computing, Storm is agnostic about the computing paradigms - it's a cluster management tool, and Giraph literally layers on top of Hadoop's original MapReduce engine, MR2.

----- so that happened...

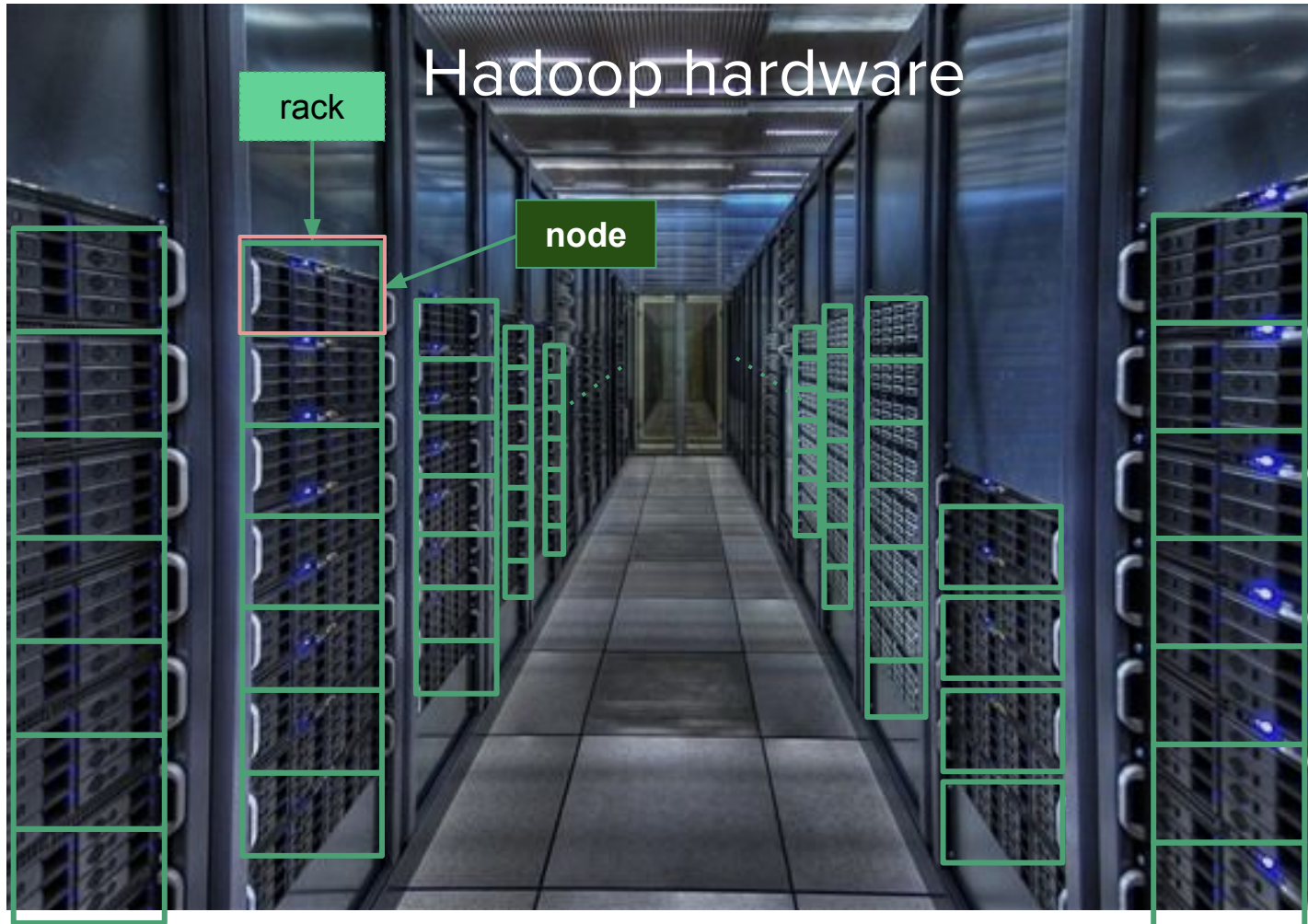
Anyway: what kinds of selections do you need to make in analysis?

Break

Hadoop hardware



Hadoop hardware



The Hadoop cluster

- **Commodity machines**

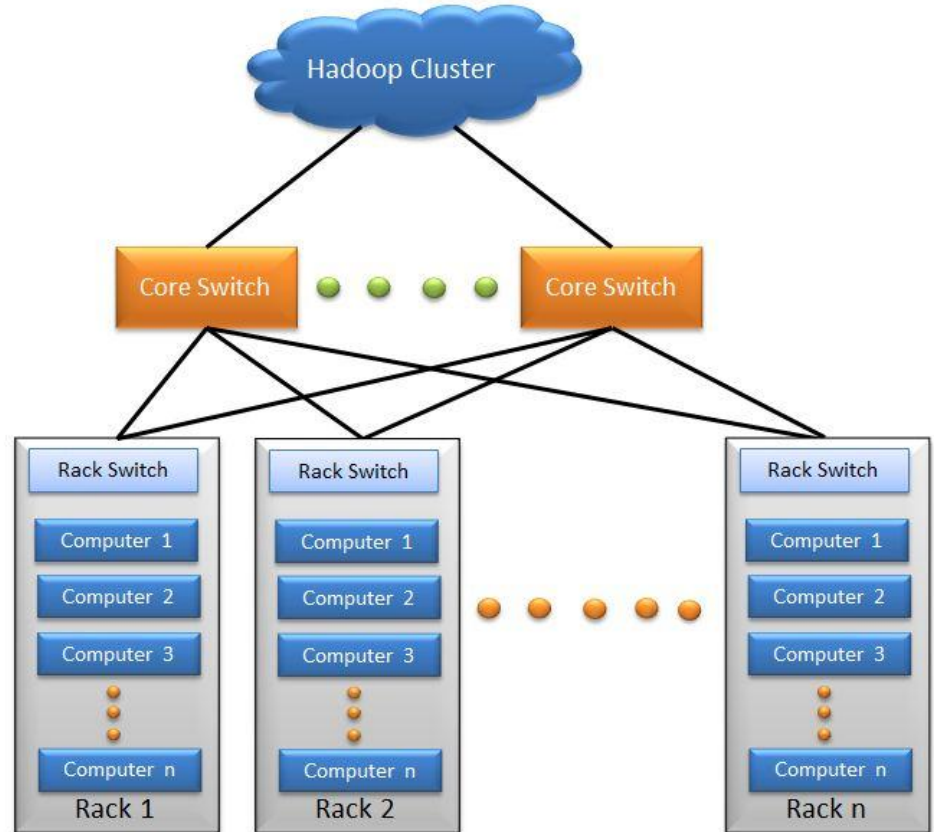
- Identical and plentiful
- ~\$8K per machine

- **40 - 50,000 nodes**

- may have many racks
- may have multiple clusters
- hierarchical switching

- **Network**

- 1 Gigabit Ethernet in-rack
- 10 Gigabit between-rack



10 - 50,000 machines

- 1 - 6,000 racks
- 1 - 50 clusters
- *average*: ~100 machines, 1 cluster

2 masters

- File System Master
- Job Master
- Each has a standby (HA)

10 - 49,998 slaves

- data storage
- compute tasks
- co-locate data and tasks

Hadoop hardware

Hadoop: Masters and Slaves

HDFS: Data storage

Master: NameNode

- 1 active, 1 standby

Slaves: DataNodes

- usually, 100s

YARN: Job Management

Master: ResourceManager

- 1 active, 1 standby

Slaves: NodeManagers

- usually, 100s

Data nodes

- **Processor:** dual hex/octo-core CPUs
- **Memory:** up to 1 TB ECC RAM
 - minimum 64GB
- **Storage:** 12-24 x 1-4 TB SATA disks
 - JBOD configuration



<http://www.supermicro.com/solutions/hadoop.cfm>

How much data can be stored?

Number of data nodes: 40 - 5,000

Storage per node:

between 12 and 24 disks at 1 - 4 TB per disk

⇒ 10-100 TB per node

Upshot: 400 Terabytes -> 500 Exabytes

Master nodes

- **Storage:** 4–6 1TB hard disks
 - 1 for the OS
 - 1 for Journals
 - 4 for the FS image [RAID 1+0] - NameNode
 - 1 for Apache ZooKeeper for ResourceManager HA
- **Processing:** 4 hex/octo-core CPUs
- **Memory:** 128 - 512GB of RAM

Master daemons have standby daemons

- reside on powerful servers
- monitor the health of their slaves
- are critical to the function of the cluster

Hadoop has daemons

Hadoop daemons are immortal JVMs

- pair up and share a machine
- the slave machines are called datanodes
- are easily replaced

YARN daemons

Resource Manager
Node Managers

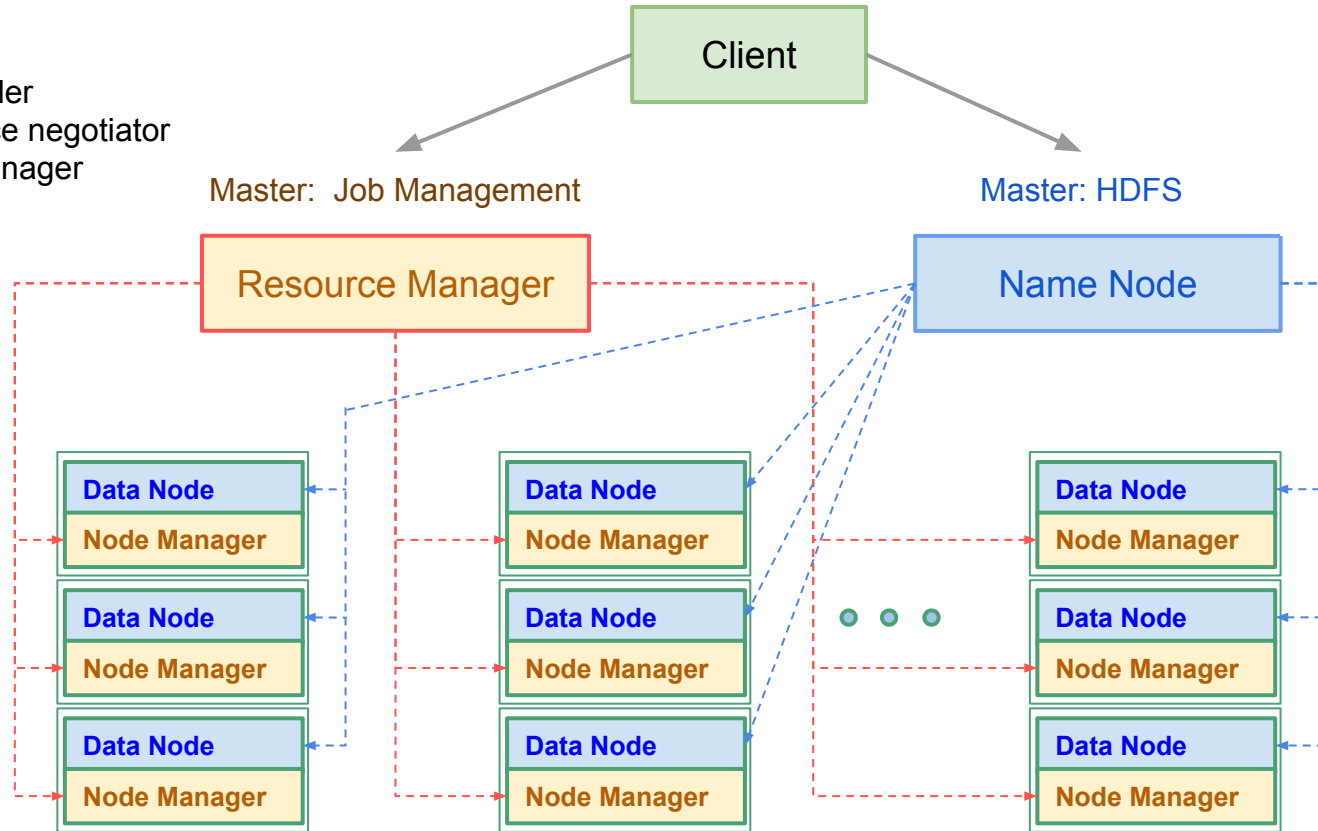
HDFS daemons

Name Node
Data Nodes

... and lesser daemonical creatures

Hadoop Nodes

- scheduler
- resource negotiator
- app manager



- File metadata
- disk health
- replication

Data nodes do all the computation



Always running:

- Data node daemon
 - manages file storage



- Node manager daemon
 - manages compute tasks



The importance of the namenode

Hadoop stores files using HDFS

- when a file is stored
 - broken into 128 M blocks
 - each block is unstructured bytes
 - map from file -> block: metadata
- datanodes store blocks of data
- the namenode stores the metadata

that's why there's a standby namenode



HDFS

basics

Exposes an API for the distributed file system (DFS)

Linux-like file system commands

- ls
- mkdir
- put
- copyFromLocal
- get
- copyToLocal

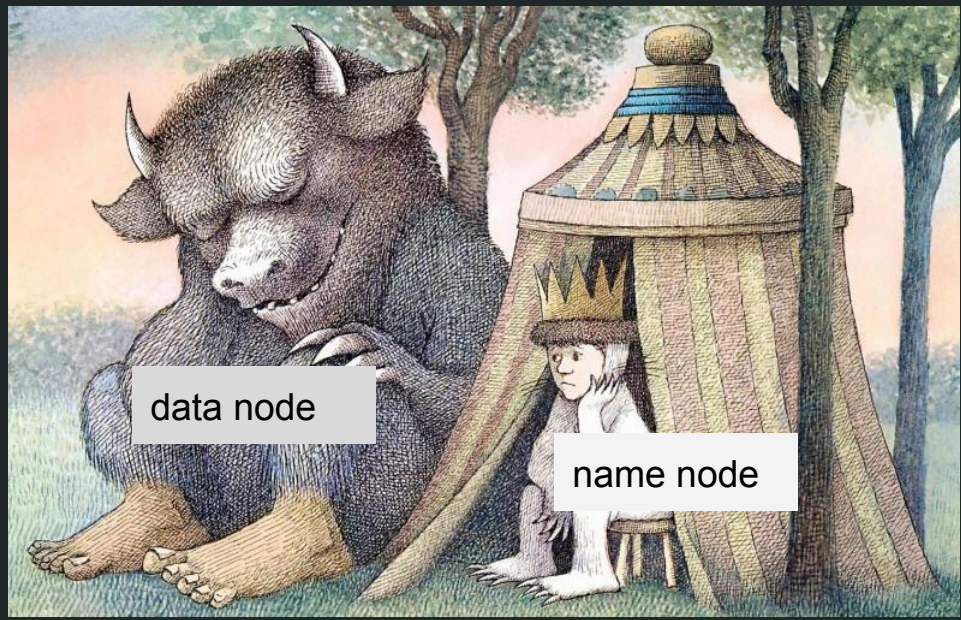
Provides client applications the locations of blocks in a file

Practice with HDFS

practice1UsingHDFS.pdf

HDFS

Deep dive



Hadoop Distributed File System

Let's talk about the weather

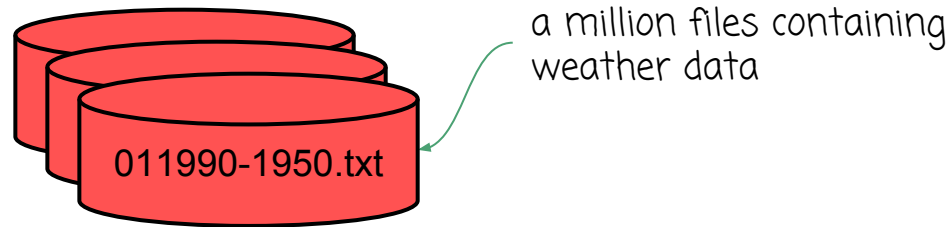
Example from Hadoop: The Definitive Guide

Tom White

2015

Typical example

- Load weather data onto the cluster
- Analyze the data
- Read the results

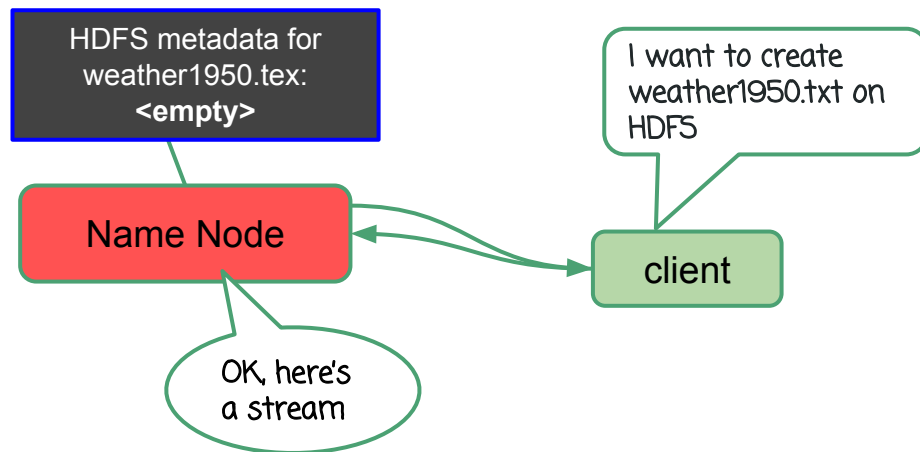


Question: What was the maximum temperature at each location

dataset: the weather dataset

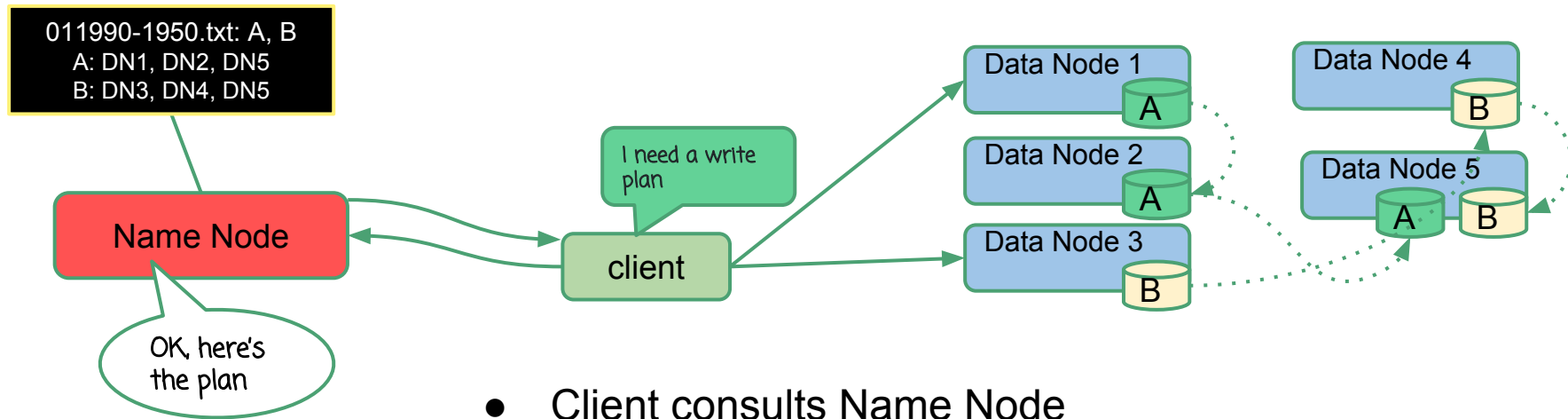
- Weather data
- From National Climate Data Center
 - <ftp://ftp.ncdc.noaa.gov/pub/data/noaa/>
- Sensors collect data every hour, many locations
- Some data goes back as far as 1901
- We have a million files

Loading data: initiating the write



- you type `"hadoop fs -put /home/emma/noaa/1950.txt weather1950.tex"`
- behind the scenes
 - the client requests a new HDFS file from the Name Node
 - the Name Node creates metadata for the file, returns an output stream

Loading data: getting the plan



- Client consults Name Node
- Client writes block directly to one Data Node
- Data Nodes replicate block
- Cycle repeats for the next block

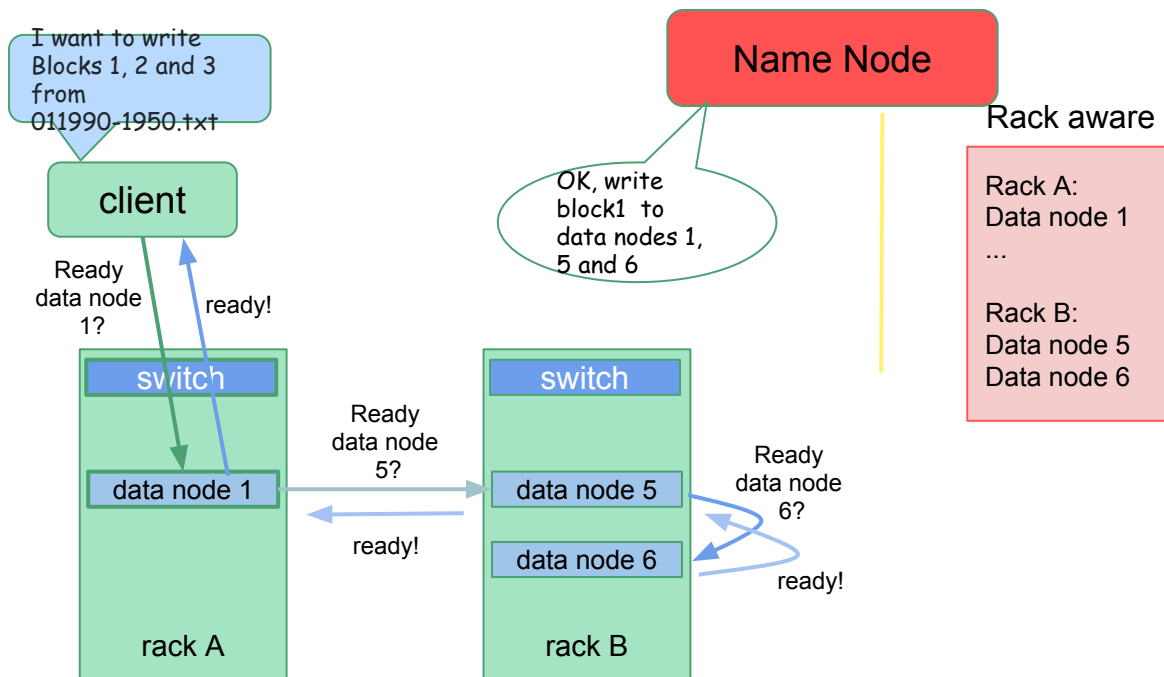
The write plan

Replication rule:

- NameNode picks two nodes on the same rack and one node on a different rack.

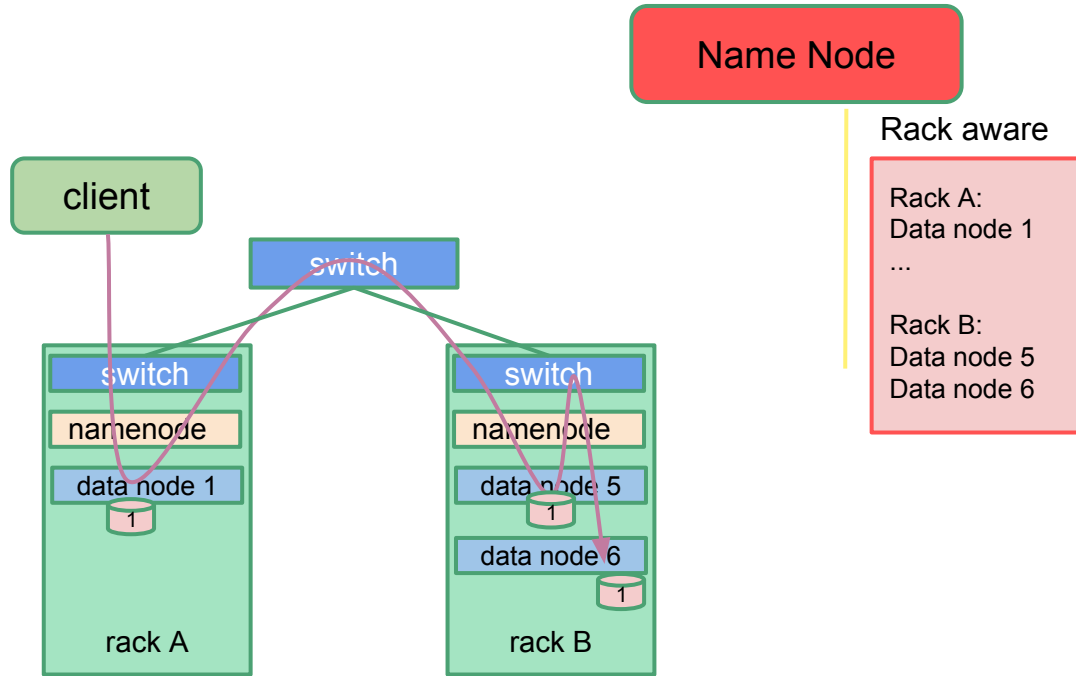
This provides:

- Data protection
- Data locality for M/R

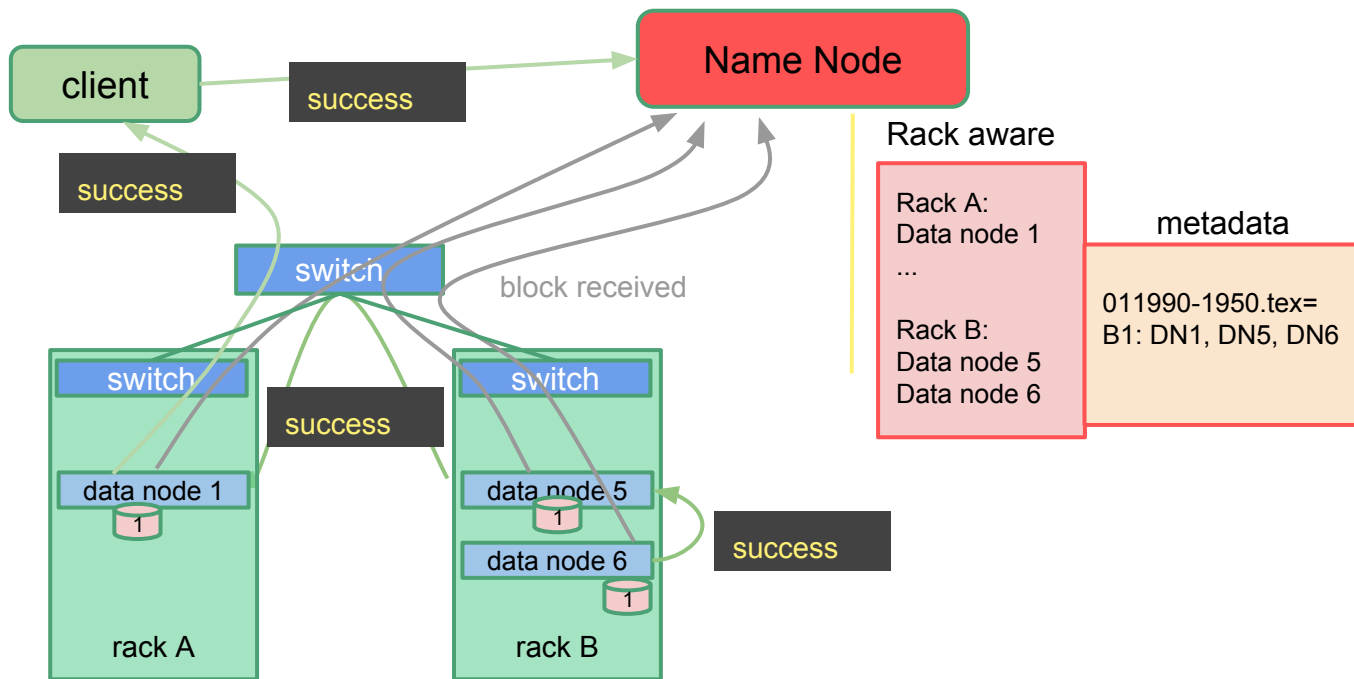


The writing pipeline for block1

- Data Nodes 1 and 5 pass data along as it is received.
- TCP 50010



Write success for block1

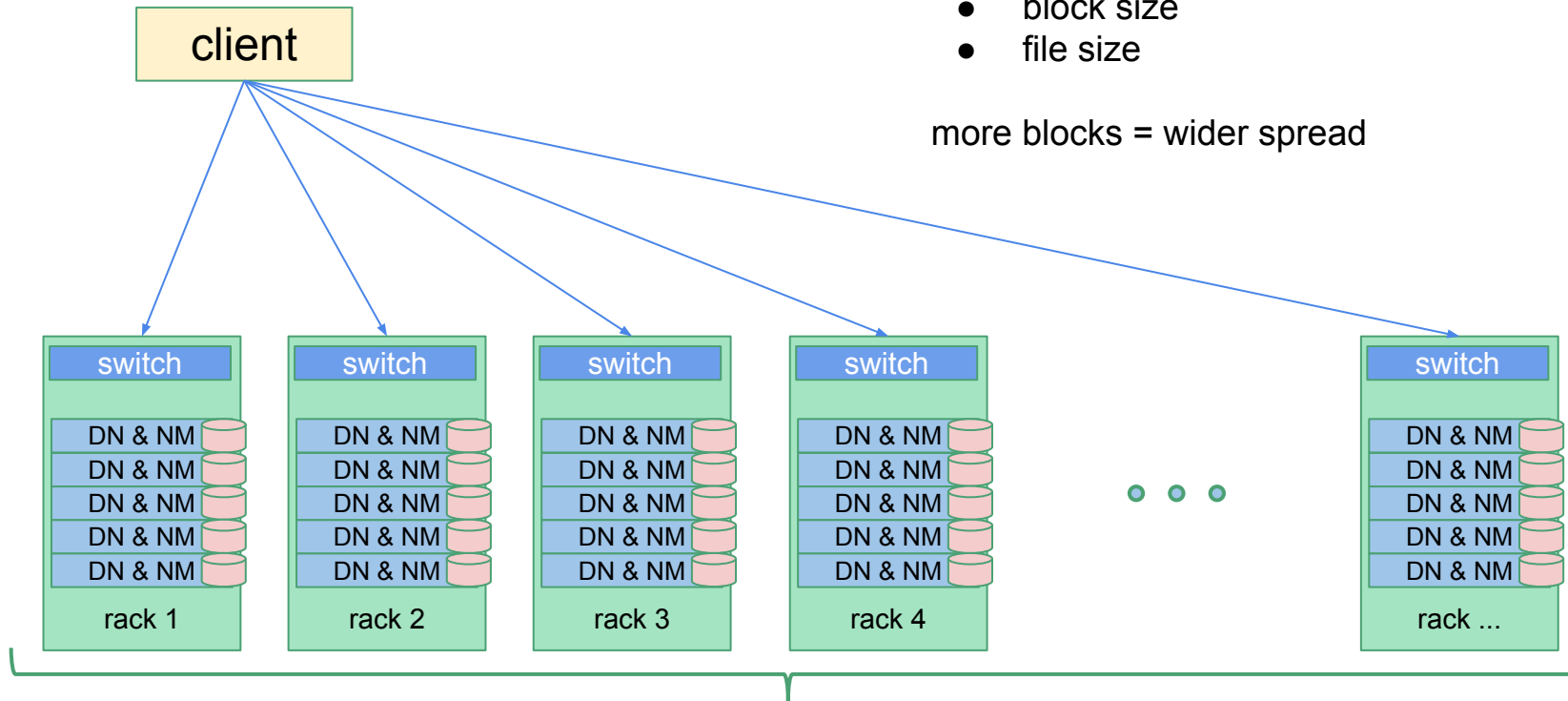


Loaded data spans the cluster

factors:

- block size
- file size

more blocks = wider spread



011990-1950.txt blocks are on multiple racks

Done!

011990-1950.txt is now distributed across the cluster

- All just by typing

```
$ hadoop fs -put 011990-1950.txt 011990-1950.txt
```

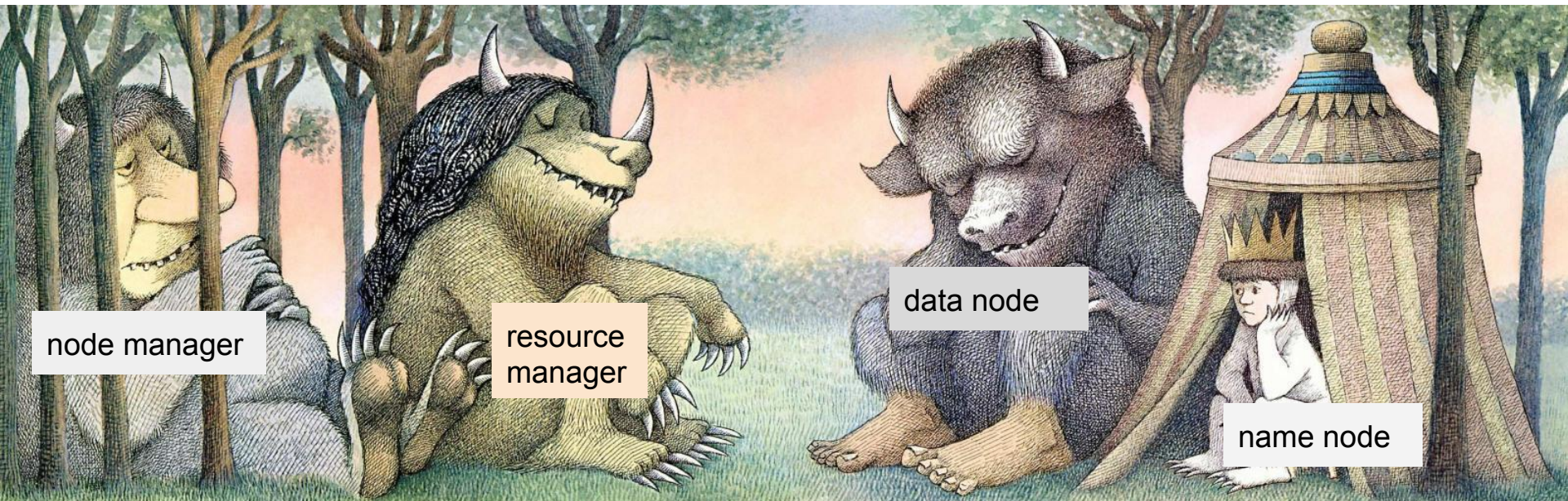
- We could specify a put on the whole directory:

```
$ hadoop fs -put weatherDir weatherDir
```

- But be careful with **put**...

Putting a large directory can take a very long time and use alot of space..

When the daemons are idle...



Next: Running Map-Reduce
on the NCDC weather data

Analyzing the weather data

- What we will do
 - check out the weather dataset
 - establish a baseline
 - look at MR dataflow
 - look at mapper and reducer code
 - look at the driver code - how to submit the job to a cluster

dataset: Now in HDFS directories by year



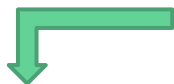
Index of /pub/data/noaa/1950/

Name	Size	Date Modified
[parent directory]		
011990-99999-1950.gz	11.5 kB	11/18/04, 4:00:00 PM
012650-99999-1950.gz	13.2 kB	11/18/04, 4:00:00 PM
013640-99999-1950.gz	11.3 kB	11/18/04, 4:00:00 PM
014150-99999-1950.gz	74.8 kB	11/18/04, 4:00:00 PM
014480-99999-1950.gz	12.7 kB	11/18/04, 4:00:00 PM
031350-99999-1950.gz	314 kB	11/18/04, 4:00:00 PM
033700-99999-1950.gz	100 kB	11/18/04, 4:00:00 PM
033770-99999-1950.gz	38.5 kB	11/18/04, 4:00:00 PM
034820-99999-1950.gz	114 kB	11/18/04, 4:00:00 PM
034873-99999-1950.gz	120 kB	11/18/04, 4:00:00 PM
035660-99999-1950.gz	34.7 kB	11/18/04, 4:00:00 PM
035671-99999-1950.gz	23.9 kB	11/18/04, 4:00:00 PM
035773-99999-1950.gz	72.7 kB	11/18/04, 4:00:00 PM
035810-99999-1950.gz	89.8 kB	11/18/04, 4:00:00 PM
035833-99999-1950.gz	149 kB	11/18/04, 4:00:00 PM


- data from 1901-2013.
- gzipped file for each station.
- avg 10^4 stations.
- $112 \text{ dir} * 10^4 \sim$ million files.

PERFECT!!

dataset: Data Format - ASCII records.

 “who, when, where”
keys

AWS station id
WBAN station id
date
time
latitude
longitude
elevation

 weather data
always with a “quality score”

wind
cloud
visibility
temperature

Description of a record with fields and position

Position	Field	Len	Format	Description
Control Data Section				
1 - 4	Rec Length	4	9999	Length, in bytes, of the portion of the record beyond position 105.
5 - 10	AWS Id	6	999999	AWS station identifier, padded on the left with zeros.
11 - 15	WBAN Id	5	99999	WBAN station identifier, padded on the left with zeros.
16 - 19	Year	4	9999	Observation year.
20 - 21	Month	2	99	Observation month, padded with zero.
22 - 23	Day	2	99	Observation day, padded with zero.
24 - 25	Hour	2	99	Observation hour in 24-hr. notation, padded with zero.
26 - 27	Minute	2	99	Observation minute, padded with zero.
Mandatory Data Section				
61 - 63	Wind Dir	3	999	Wind direction.
64 - 64	Wind Dir QC	1	X	Wind direction QC code.
65 - 65	Wind Type	1	X	Wind observation type code.
66 - 69	Wind Speed	4	9999	Wind speed in m/s.
70 - 70	Wind Spd QC	1	X	Wind speed QC code.
71 - 75	Cloud Hgt	5	99999	Ceiling height in meters.
76 - 76	Cld Hgt QC	1	X	Ceiling height QC.
77 - 77	Clg Det Cd	1	X	Ceiling determination code.
78 - 78	Cavok Cd	1	X	CAVOK code.
79 - 84	Visibility	6	999999	Visibility in meters.
85 - 85	Visibility QC	1	X	Visibility QC code.
86 - 86	Vis Var Cd	1	9	Visibility variability code.
87 - 87	Vis Var QC	1	9	Visibility variability QC code.
88 - 92	Temp	5	+9999	Air temperature in degrees Celsius.
93 - 93	Temp QC	1	X	Air temperature QC code.

creating a baseline

- Use linux tools
 - Make awk script, max_temp.sh
 - Run it.

>> ./max_temp.sh

1901 31.7

1902 24.4

1903 28.9

1904 25.6

1905 28.3

...

```
#!/usr/bin/env bash
for year in all/*
do
    echo -ne 'baseyear $ year . gz' "t"
    gunzip -c $year | \
        awk '{ temp = substr($0, 88, 5) + 0;
              q = substr($0, 93, 1);
              if (temp != 9999 && q ~ /[01459]/ && temp >
max) max = temp  }
              END { print max/10 } '
done
```

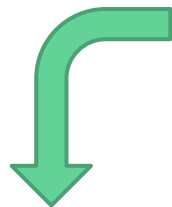

baseline: can we make it faster?

- Run each year in parallel.
 - ... spawn 112 processes.
- Divide work into equal-sized chunks
 - do extreme bookkeeping.
 - ... and then combine results.

It still doesn't scale: runs on a single machine.

Map-reduce processing

Map



```
0, 00430119909999991950010118004+68750+023550FM-12+...9999999N9-02281+99...  
106, 00560119909999991950080407004+68750+023550FM-12+...9999999N9+02611+99...  
212, 00430119909999991950120212004+68750+023550FM-12+...9999999N9-02311+99...  
318, 00430119909999991949010218004+68750+023550FM-12+...9999999N9-03001+99...  
424, 00560119909999991949080307004+68750+023550FM-12+...9999999N9+02891+99...
```

(1950, -22.8)
(1950, +26.1)
(1950, -23.1)
(1949, -30.0)
(1949, +28.9)

Shuffle-sort



(1949, {-30, +28.9})
(1950, {-22.8, +26.1, -23.1})

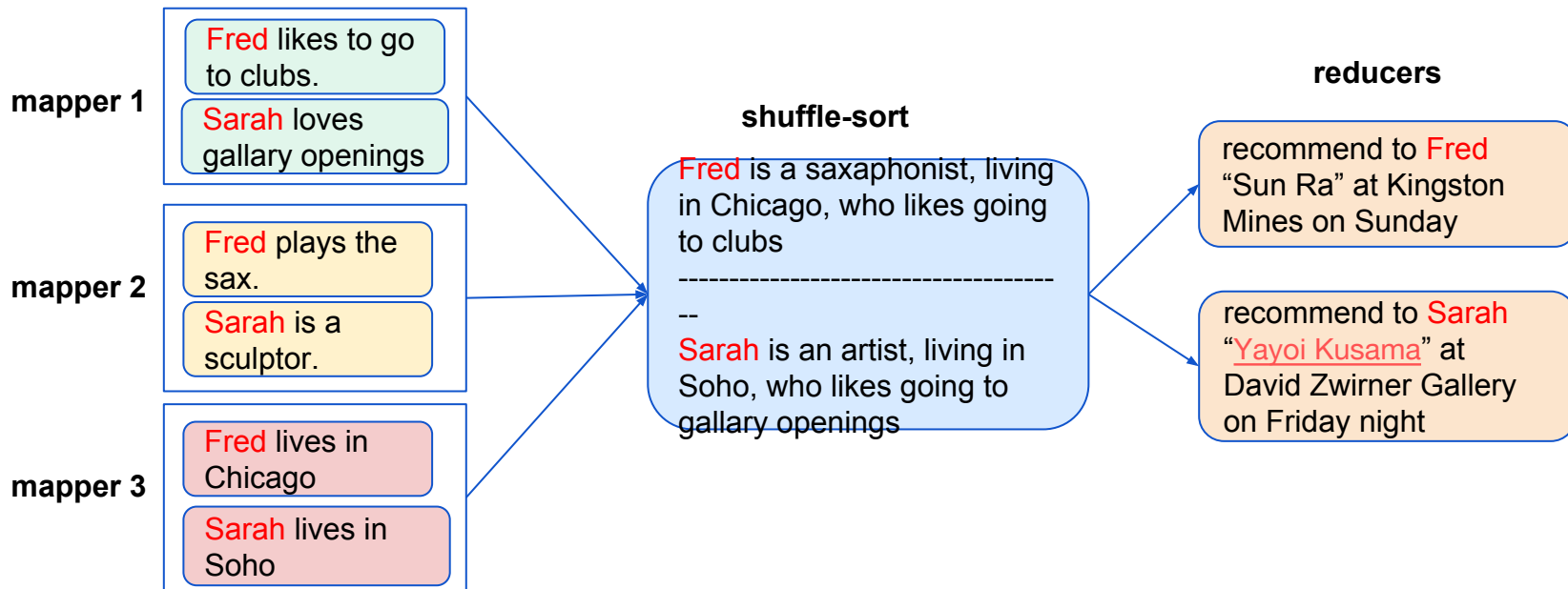
Reduce



(1949, 28.9)
(1950, 26.1)

Map-reduce works in three phrases

- Map extracts information from a record and gives it a key.
- Shuffle-sort aggregates information by key.
- Reduce analyzes the aggregated information.



Code for the weather example

Write three classes:

Driver - defines a Job, contains a main method

MaxTempMapper

- **extends** Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
- overrides the **map** function

MaxTempReducer

- **extends** Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
- overrides the **reduce** function

MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable>

Generic types

4 formal type parameters

- input key, value
- output key, value

The output from the Mapper is called
“intermediate output”

Input and output types

Optimized for network serialization

Most common

- IntWritable
- LongWritable
- FloatWritable
- DoubleWritable
- Text

see org.apache.hadoop.io

```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
```

```
    @Override
```

```
    public void map(LongWritable key, Text value, Context context) {
```

```
        String line = value.toString();
```

```
        String year = line.substring(15, 19);
```

← read the year

```
        int temp;
```

```
        if (line.charAt(87) == '+')
```

```
            temp = Integer.parseInt(line.substring(88, 92));
```

```
        else
```

```
            temp = Integer.parseInt(line.substring(87, 92));
```

← parse sign (+/-)

} parse temperature

```
        String quality = line.substring(92, 93);
```

```
        if (temp != 9999 && quality.matches("[01459]")) ;
```

```
            context.write(new Text(year), new IntWritable(temp));
```

← check validity and quality

```
    }
```

```
}
```

```
public class MaxTempReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
```

```
    public void reduce(Text key, Iterator<IntWritable> values, Context context)  
        throws IOException, InterruptedException {
```

```
        int maxVal = Integer.MIN_VALUE;
```

```
        while (values.hasNext()) {  
            maxVal = Math.max(maxVal, values.next().get());  
        }
```

Often, this is where you put
aggregative statistics.

```
        context.write(key, new IntWritable(maxVal));
```

```
    }
```

The driver: code that defines and starts a Job

Create job object

Set I/O:

- path(s) to the input data
- types of the input data
- path for the output data
- types of the output data

Define classes in the job:

- Classes to execute (Mapper, Reducer, Partitioner, etc.)
- Job name (for monitoring)

```
public class MaxTemp {
```

```
    public static void main(String[] args) throws Exception {
```

```
        Job job = new Job();
```

← Create a new Job object.

```
        job.setJarByClass(MaxTemp.class);
```

← MaxTemp.class identifies the job's jar.

```
        job.setJobName("MaxTemperature");
```

```
        FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

} FS paths - input and output.

```
        job.setOutputKeyClass(Text.class);
```

```
        job.setOutputValueClass(IntWritable.class);
```

} Hadoop IO classes for final data

```
        job.setMapperClass(MaxTempMapper.class);
```

```
        job.setReducerClass(MaxTempReducer.class);
```

} Mapper and Reducer classes

```
        System.exit(job.waitForCompletion(true) ? 0 : 1);
```

← Submit the job

```
}
```


Running the job on a Hadoop client

1. Compile your code:

```
$ javac -classpath `hadoop classpath` jobDir/*.java
```

2. Collect your .class files into a jar:

```
jar cvf myJob.jar jobDir/*.class
```

3. Submit your job:

```
$ hadoop jar myJob.jar jobDir.MyDriver input output_directory
```

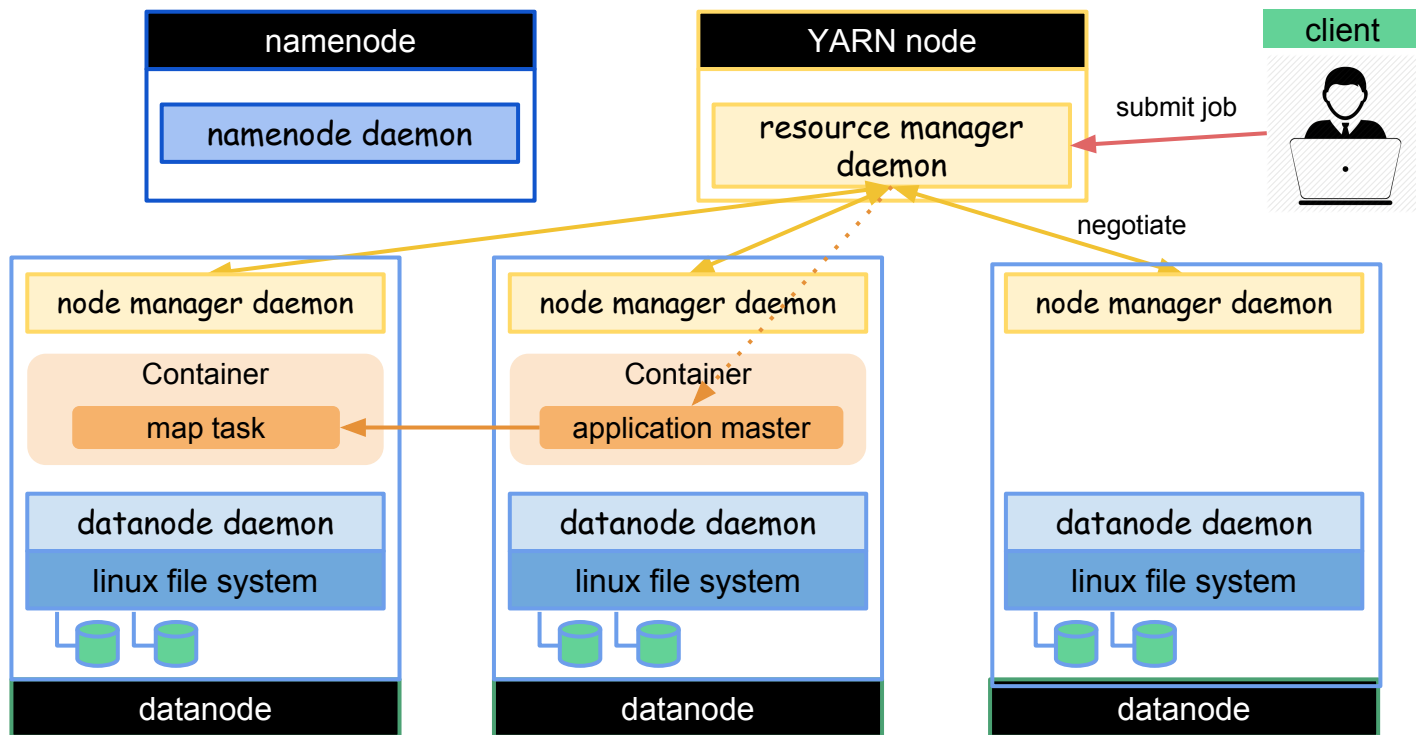
We will practice this next week.

Running the job in Eclipse

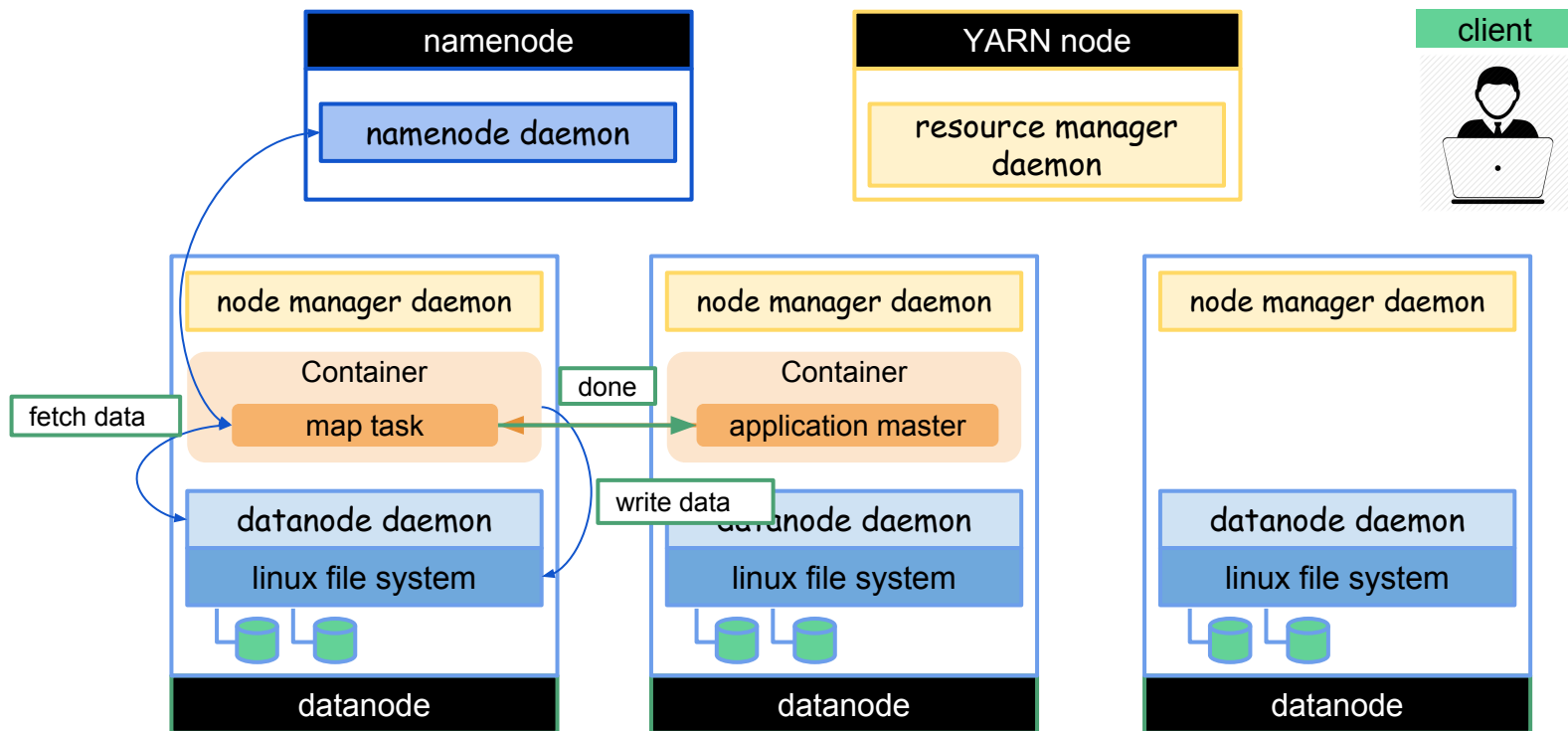
1. In **Eclipse**, locate the driver class in the **Package Explorer**.
2. Right click on the driver class and select **Run As > Run Configurations...**
3. This will bring up the “**Run Configurations**” window. Select “**Java Application**”.
4. Click on the little “new” icon at the top right
5. This creates a new run configuration for your **driver**. Check the Main class - it should be your driver class
6. Select the ‘**Arguments**’ tab. In the arguments, specify the input and output folders or files.
 - Note, these will be local files, not HDFS files.
7. Click “**run**”

We will practice this next week.

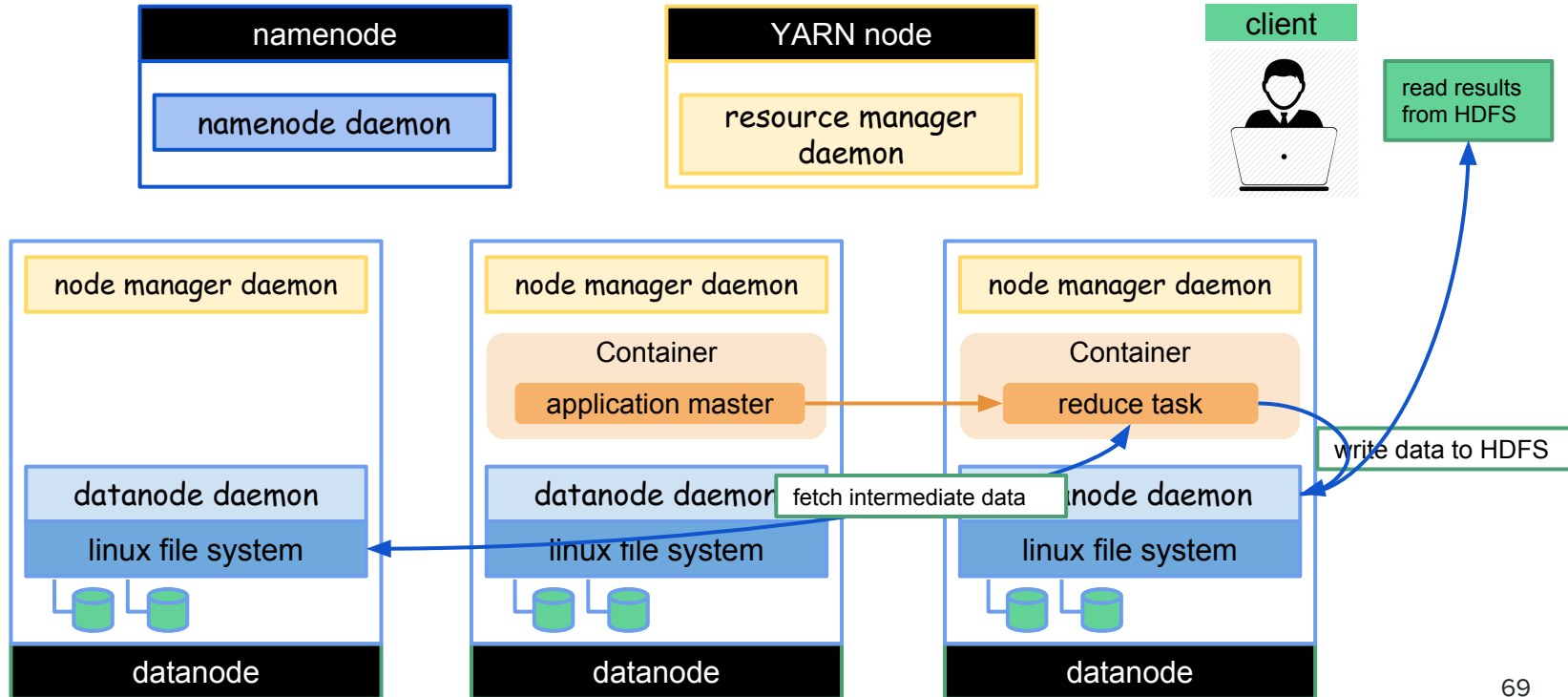
Running a job



What happens when you run a job



Running a job



Viewing the results in HDFS

Access the file system using Java

```
Path path = new Path(outputPath);
```

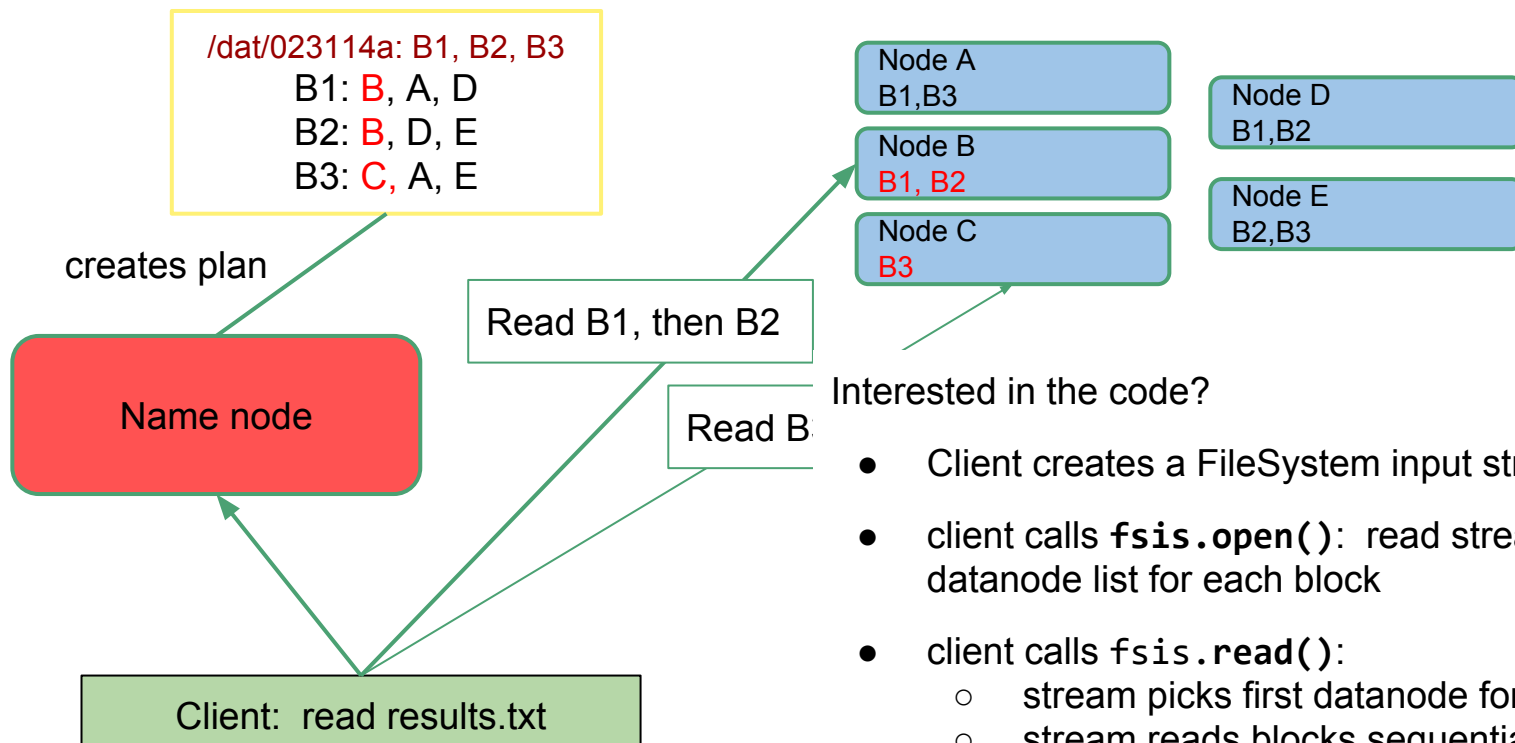
```
FileSystem fs = path.getFileSystem(conf);
```

```
FSDataInputStream inputStream = fs.open(path);
```

Access the file system using `hadoop fs -cat <outputFile>` (as in practice)

- this runs a Java program for you, doing the above.
- what does this do (internally)?

Client reading files

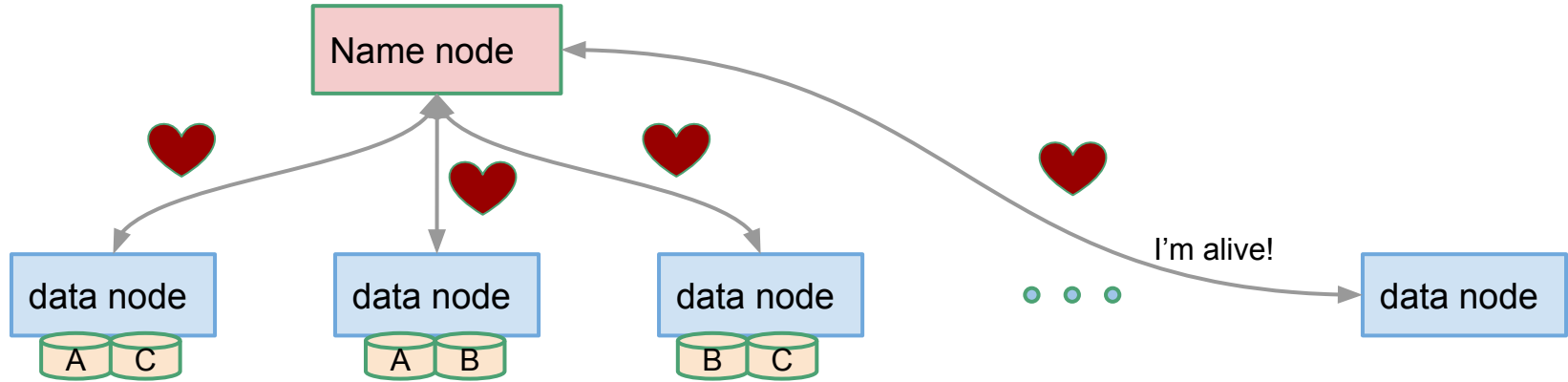


Interested in the code?

- Client creates a FileSystem input stream (fsis)
- client calls `fsis.open()`: read stream receives datanode list for each block
- client calls `fsis.read()`:
 - stream picks first datanode for each block
 - stream reads blocks sequentially
- client calls `fsis.close()`.

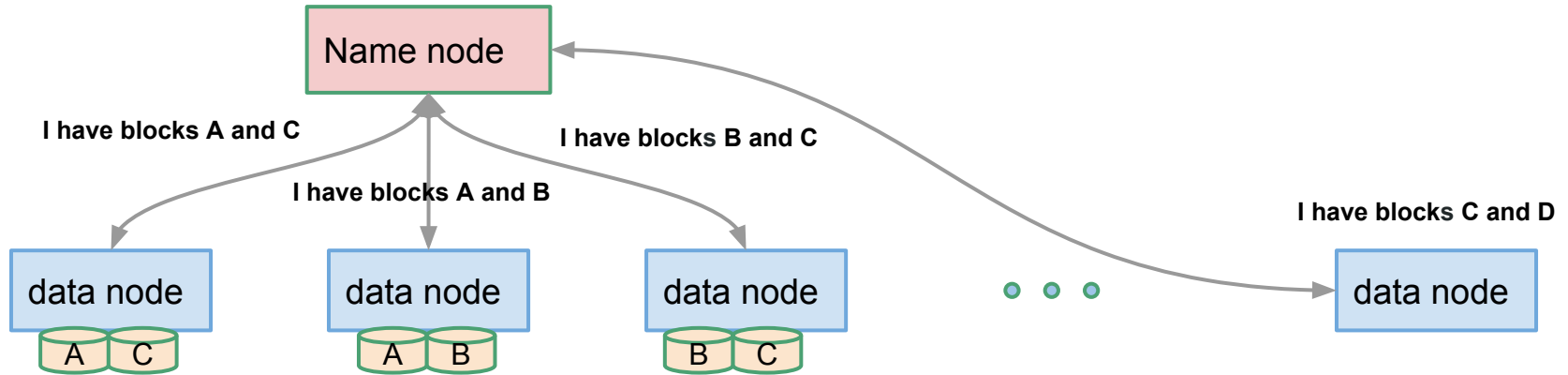
Hadoop health and well-being

Data node reports health to the Name node



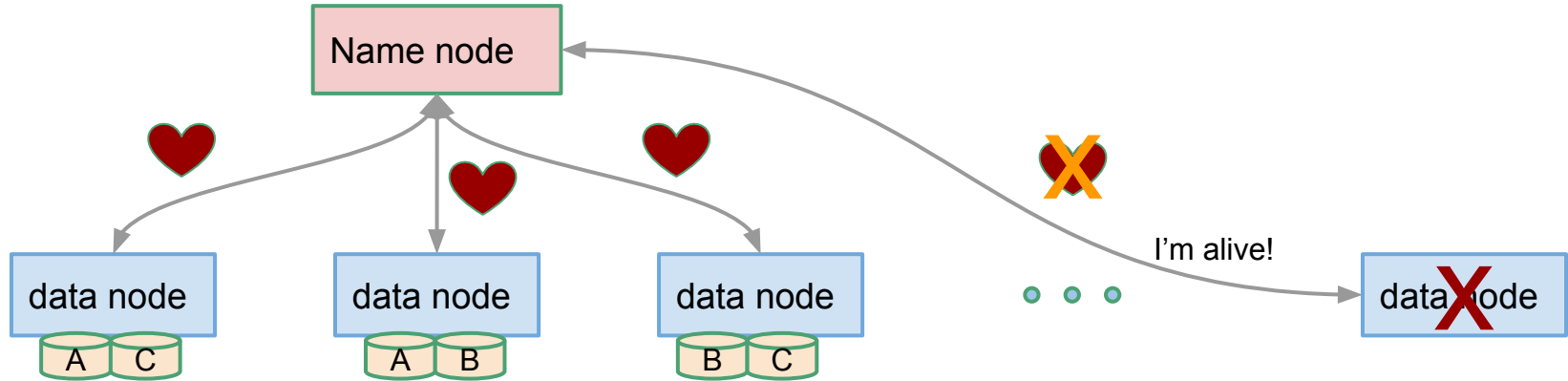
- Data Node sends Heartbeats every few seconds
- 3 second is default
- Communicate via TCP handshake

Data node reports data to the Name node



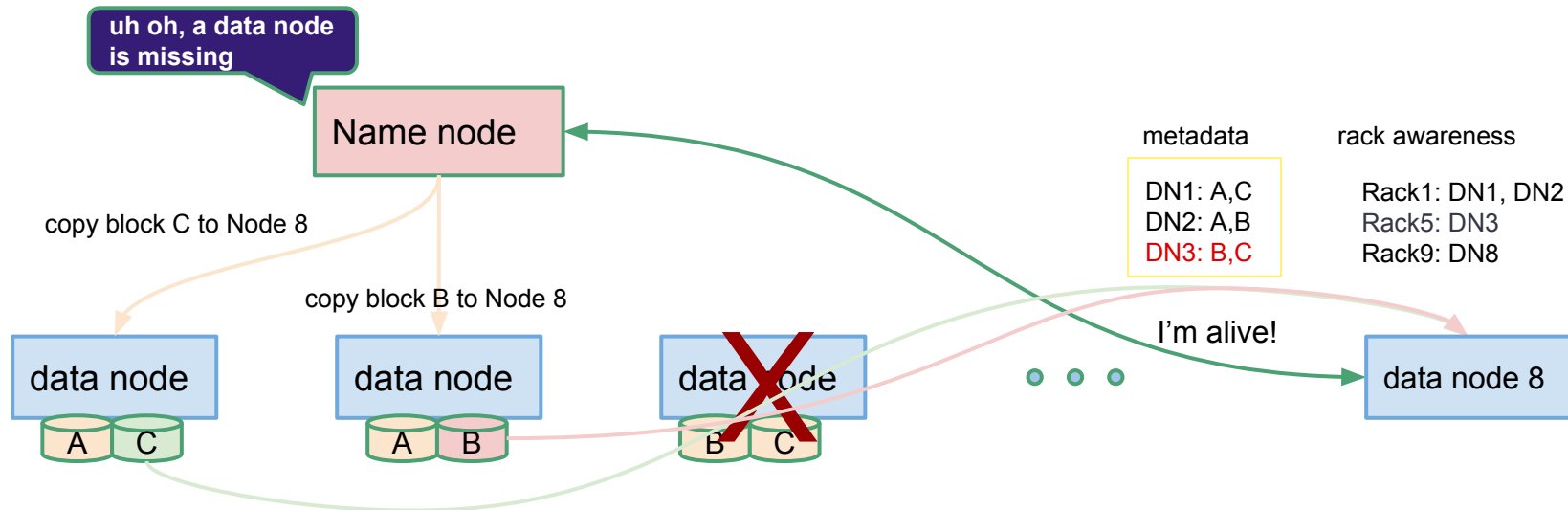
- Every 10 - 60 minutes, data nodes send block report
- Name Node checks metadata using the reports
- Safety check - can be used to rebuild Name node memory

No heartbeat: Data node is stale and then dead



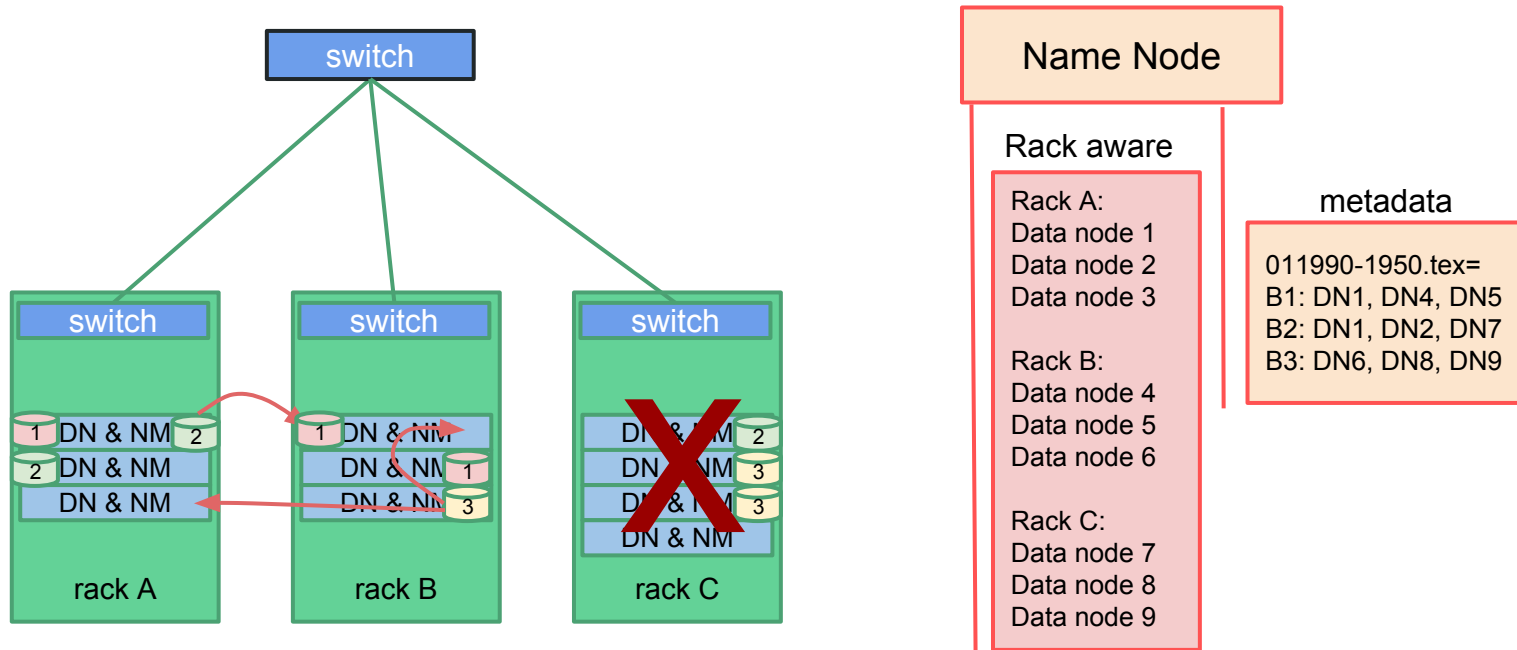
- Several heartbeats are missed: node is “stale”
- Much longer interval (> 10 minutes): node is “dead”
- If intervals are too short, good nodes are replaced

When a disk goes down: re-replicate



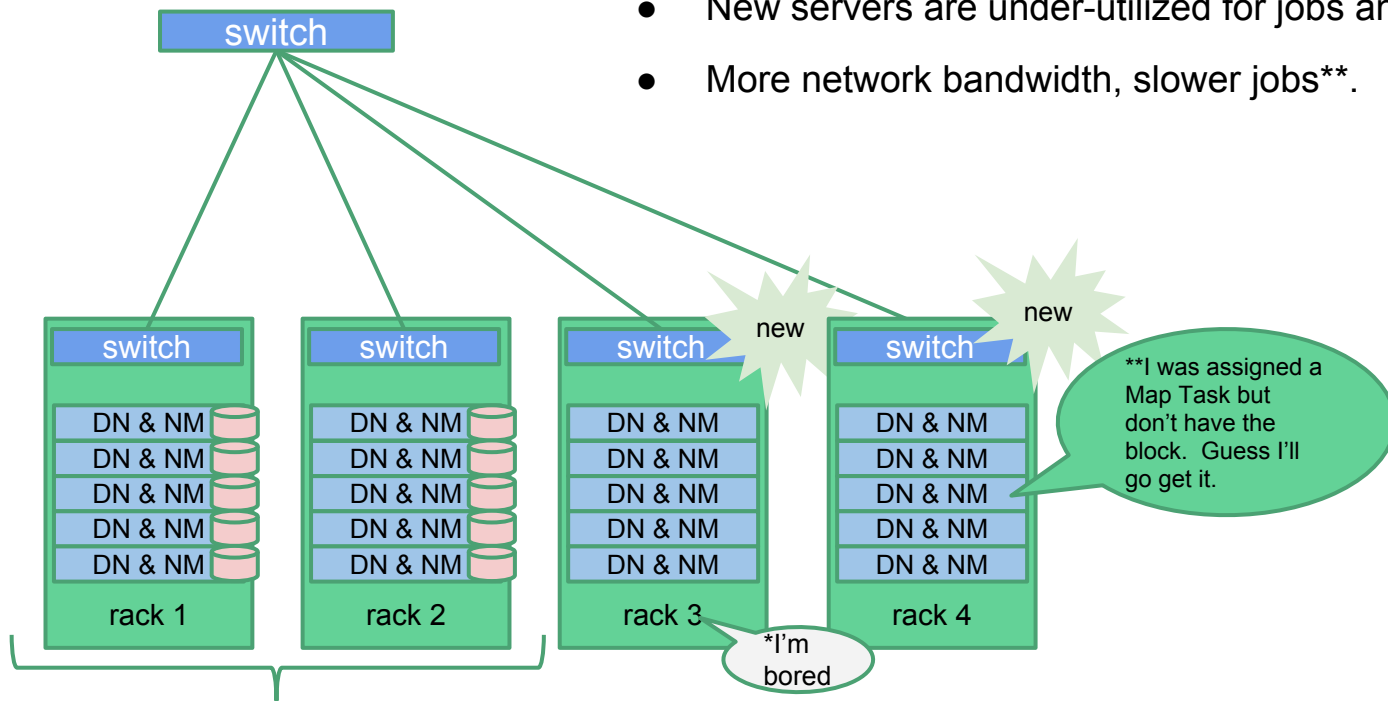
- Missing heartbeats signify lost nodes
- Name node consults metadata & rack awareness
=> finds affected data
- Name node tells live data nodes where to replicate

Rack awareness and rack recovery



The unbalanced cluster

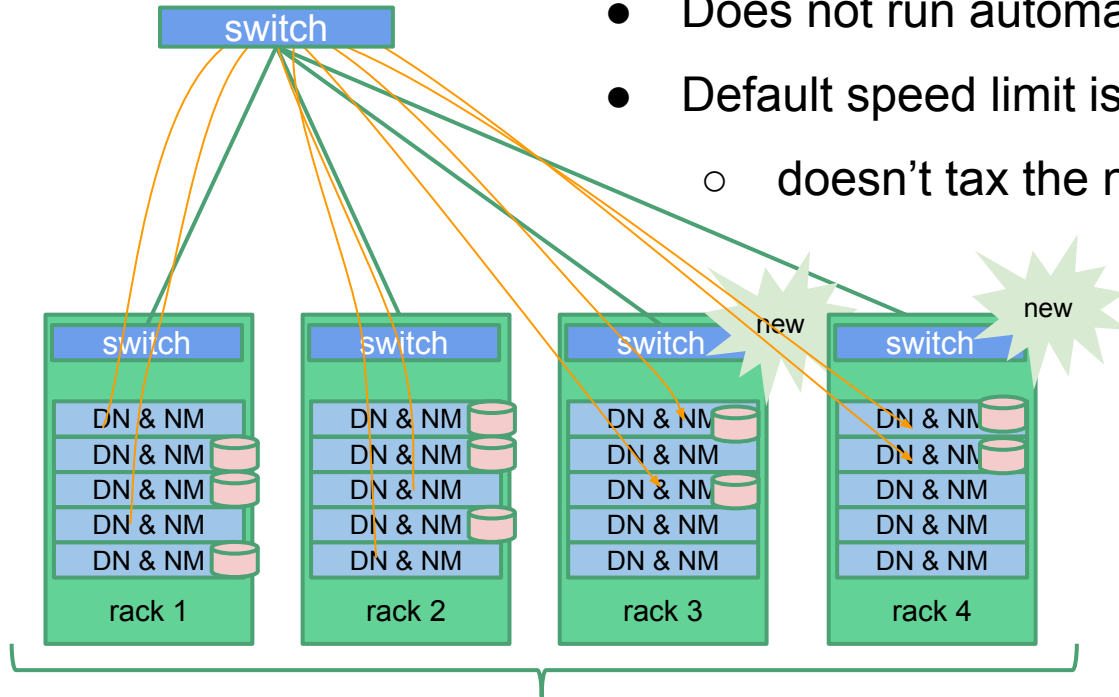
- Adding new machines: data is not automatically moved
- New servers are under-utilized for jobs and storage
- More network bandwidth, slower jobs**.



011990-1950.txt

cluster balancing

- Copies data to new under-utilized nodes
- Does not run automatically
- Default speed limit is 1 MB/s
 - doesn't tax the network



Admin runs the balancer:
\$ hadoop balancer

Key points

Name nodes know everything about the data (metadata, FS images)

- data nodes are dumb
- blocks are BIG - 128 M
- block contents are unstructured

Writing HDFS data is distributed and efficient

- pipelines are created for replication
- writes are thoroughly verified

Rack awareness should be configured for multi-rack clusters

- used for efficient data access

References

- HDFS in more depth:

Hadoop: The Definitive Guide, 4rd Edition, by Tom White, Chapter 3.

- File system shell guide

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html>

- Namenode startup

<http://hortonworks.com/blog/understanding-namenode-startup-operations-in-hdfs/>

- Namenode availability: checking-pointing explained

<http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop/>

- HDFS Metadata directories explained

<http://hortonworks.com/blog/hdfs-metadata-directories-explained/>

See you next week....

