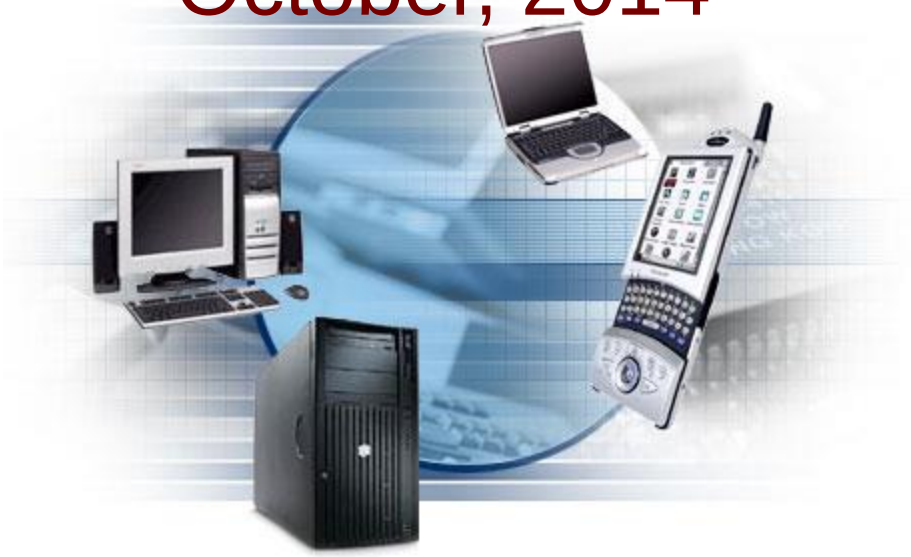
# School of Parallel Programming & Parallel Architecture for HPC
# ICTP
# October, 2014



## Hadoop for HPC

Instructor: Ekpe Okorafor

# Outline

- Hadoop – Basics
- Hadoop Infrastructure
- HDFS
- MapReduce
- Hadoop & HPC

# Hadoop - Why ?

- Need to process huge datasets on large clusters of computers
- Very expensive to build reliability into each application
- Nodes fail every day
  - Failure is expected, rather than exceptional
  - The number of nodes in a cluster is not constant
- Need a common infrastructure
  - Efficient, reliable, easy to use
  - Open Source, Apache Licence

# Who uses Hadoop?

- Amazon/A9
- Facebook
- Google
- New York Times
- Veoh
- Yahoo!
- …. many more

# Hadoop Infrastructure

Original Slides by
Ekpe Okorafor, TJ Glazier, Justin Lowe
Big Data Academy, Digital Analytics, Accenture

# Hadoop Infrastructure Overview

**Clients**

Makes the requests to the hadoop cluster to store or retrieve files .
Sends the job requests to process the data stored in the cluster

**Master Server Daemons**
Namenode
Secondary Namenode
Jobtracker

Masters control the operations that take place in the cluster

- Namenode tracks all of the block locations of data in the cluster
- Secondary Namenode does all of the housekeeping for the namenode
- Jobtrack tracks the mapreduce jobs submitted to the cluster

**Slave Server Daemons**
Datanode
Tasktracker

Slaves process and store the data

- Datanodes are tasked with holding all of the data blocks in hadoop, data is stored nowhere else
- Tasktrackers run the jobs on the cluster and report back to the Jobtracker

CLIENTS

Submits jobs to the cluster, can also store & retrieve data

NAMENODE

SECONDARY NAMENODE

Tracks all files in HDFS
Responsible for data replication
Handles Datanode failures

JOBTRACKER

Runs all jobs on the cluster
Handles tasktracker failures
Job Queue management

DATANODES/TASKTRACKERS

Holds the HDFS file system & process all jobs on behalf of the client

CLIENTS

NAMENODE

SECONDARY NAMENODE

JOBTRACKER

JOBTRACKER = FAILS

Unable to run jobs until the jobtracker is restored or replaced

DATANODES/TASKTRACKERS

**HDFS is still in-tact!**

CLIENTS

NAMENODE

SECONDARY NAMENODE

JOBTRACKER

SECONDARY NAMENODE = FAILS

HDFS is addressable however catastrophic failure is imminent as the NAMENODE housekeeping server is down

DATANODES/TASKTRACKERS

**HDFS is still in-tact!**

CLIENTS

NAMENODE

SECONDARY NAMENODE

JOBTRACKER

NAMENODE = FAILS

Master file of block metadata lost,
extremely high potential for lost data

DATANODES/TASKTRACKERS

**HDFS is *NOT* addressable**

# Namenode & Secondary Namenode


NAMENODE

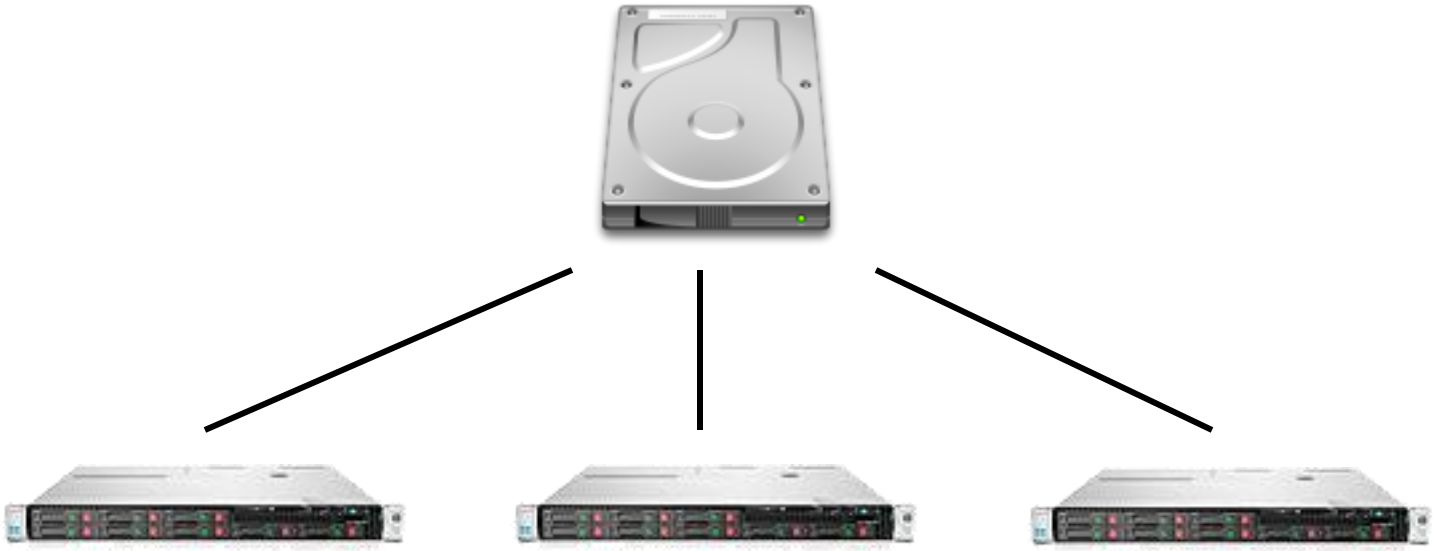
- Stores the file metadata of every block in HDFS
- This information is stored on the LOCAL hard disk of the NAMENODE

SECONDARY NAMENODE

- Does the housekeeping for the NAMENODE
- Is **NOT** a backup for the NAMENODE
- Will periodically retrieve the block location list and build a new image for the NAMENODE

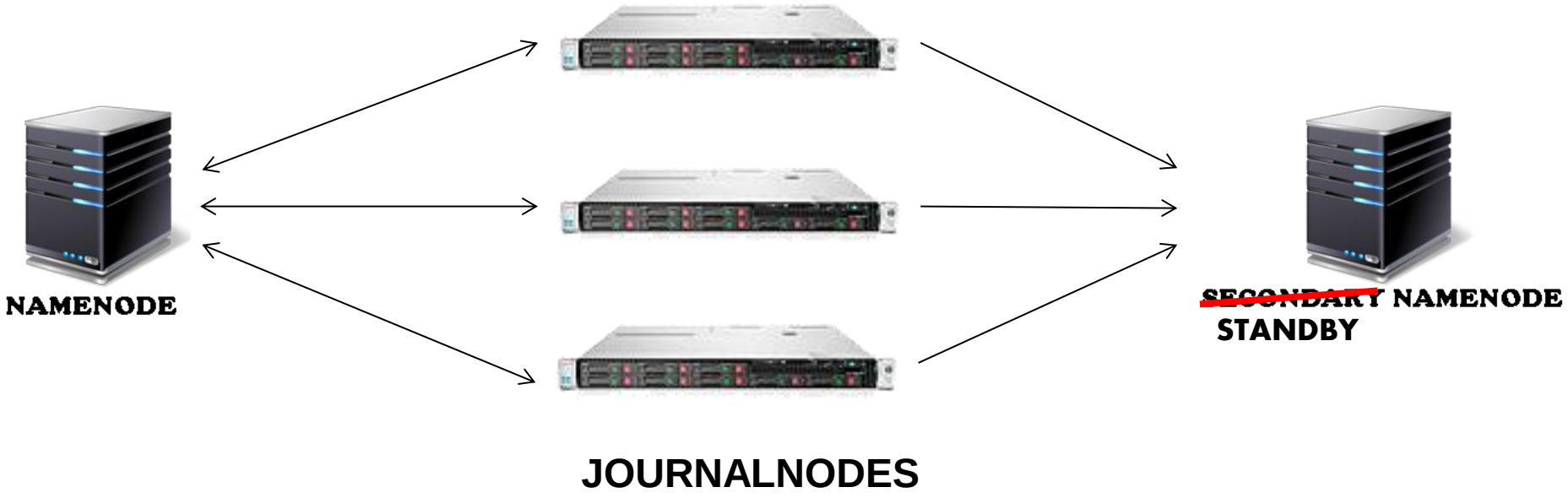
# Hadoop: High Availability


**SPOF!**
Single
Point
Of
Failure

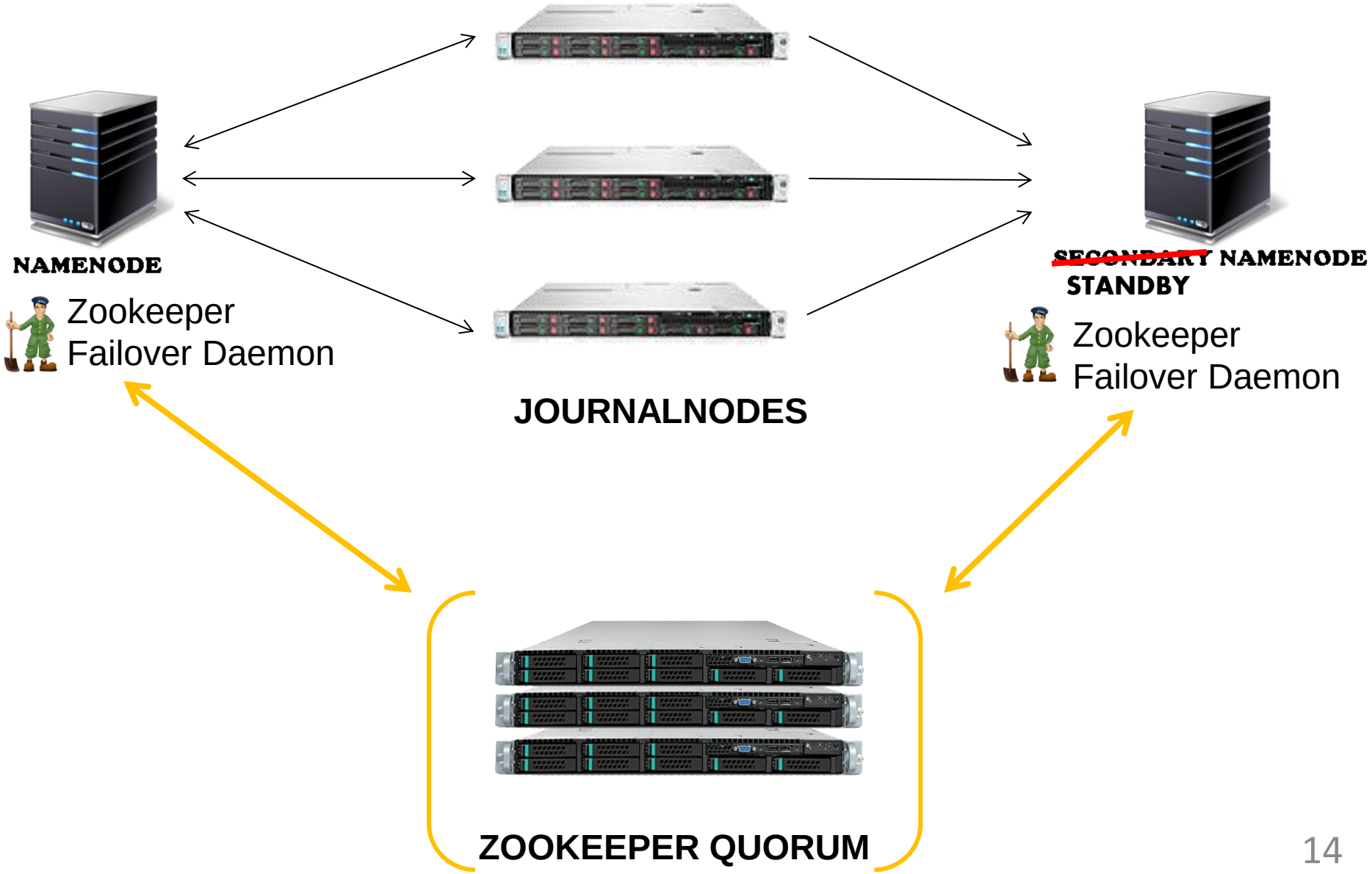


Hadoop High Availability removes the Single Point of Failure by offloading the block metadata from the local hard drive onto 3 “JOURNAL NODES”

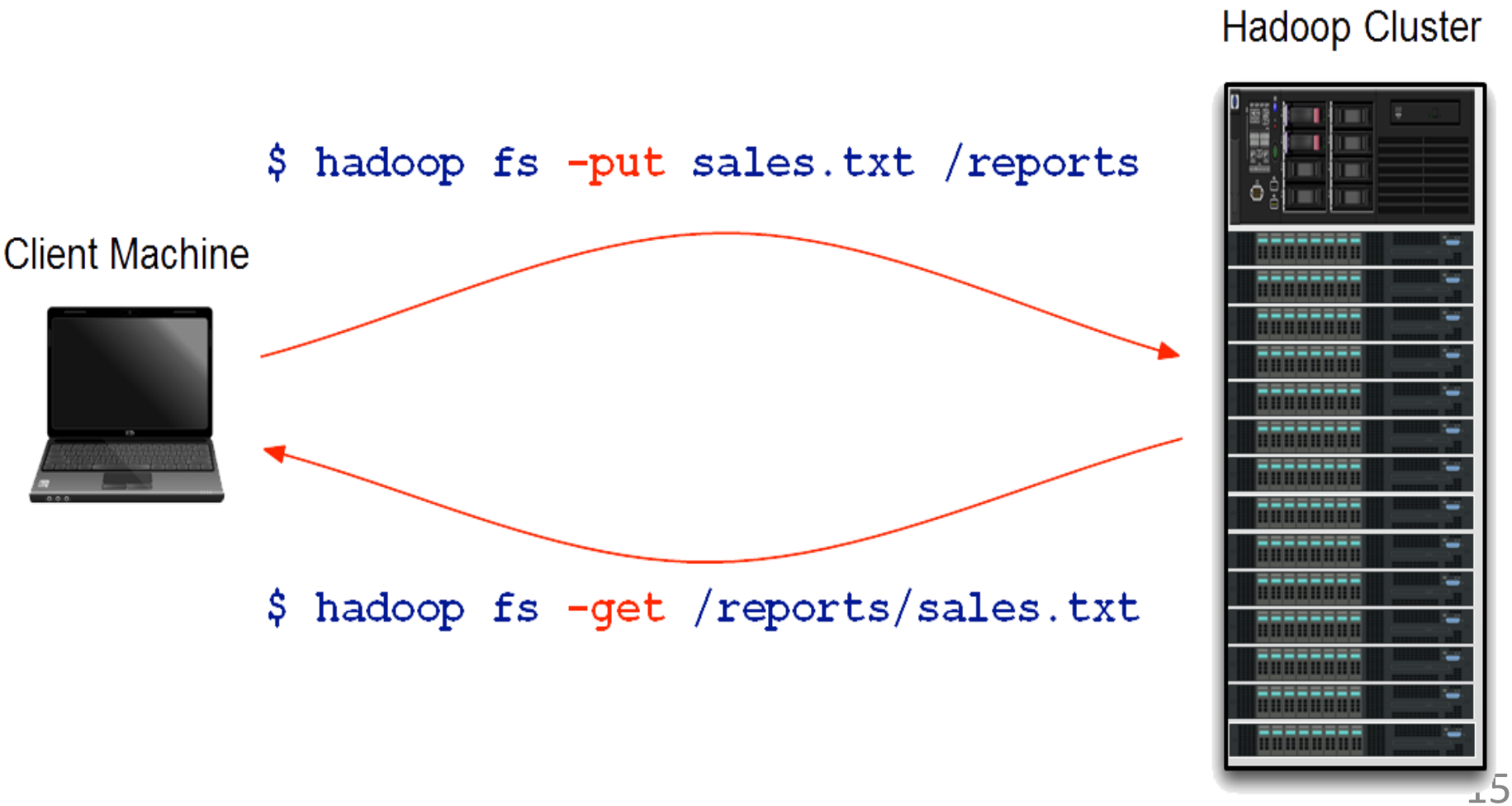
# Hadoop: High Availability



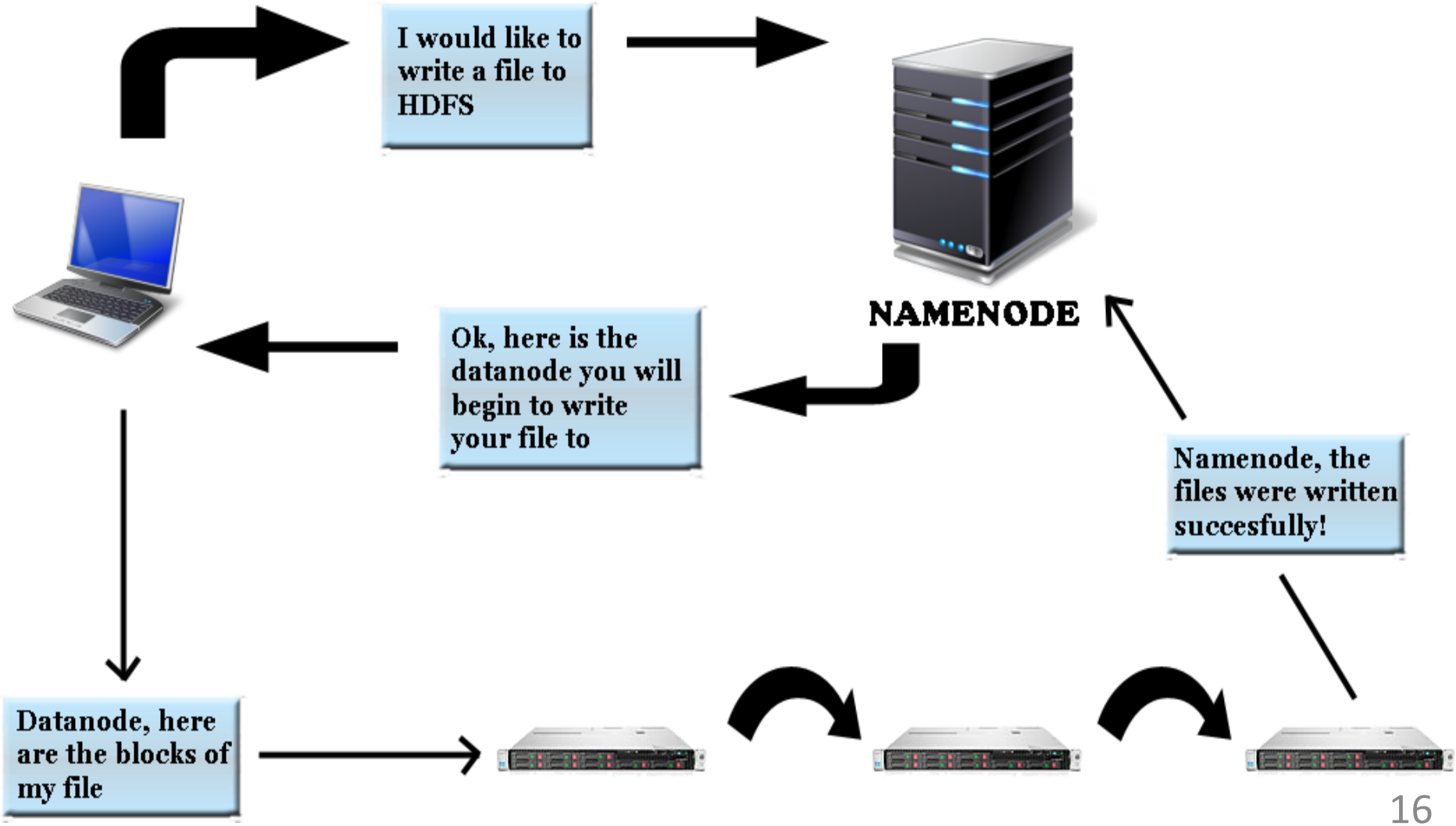
# Hadoop: High Availability

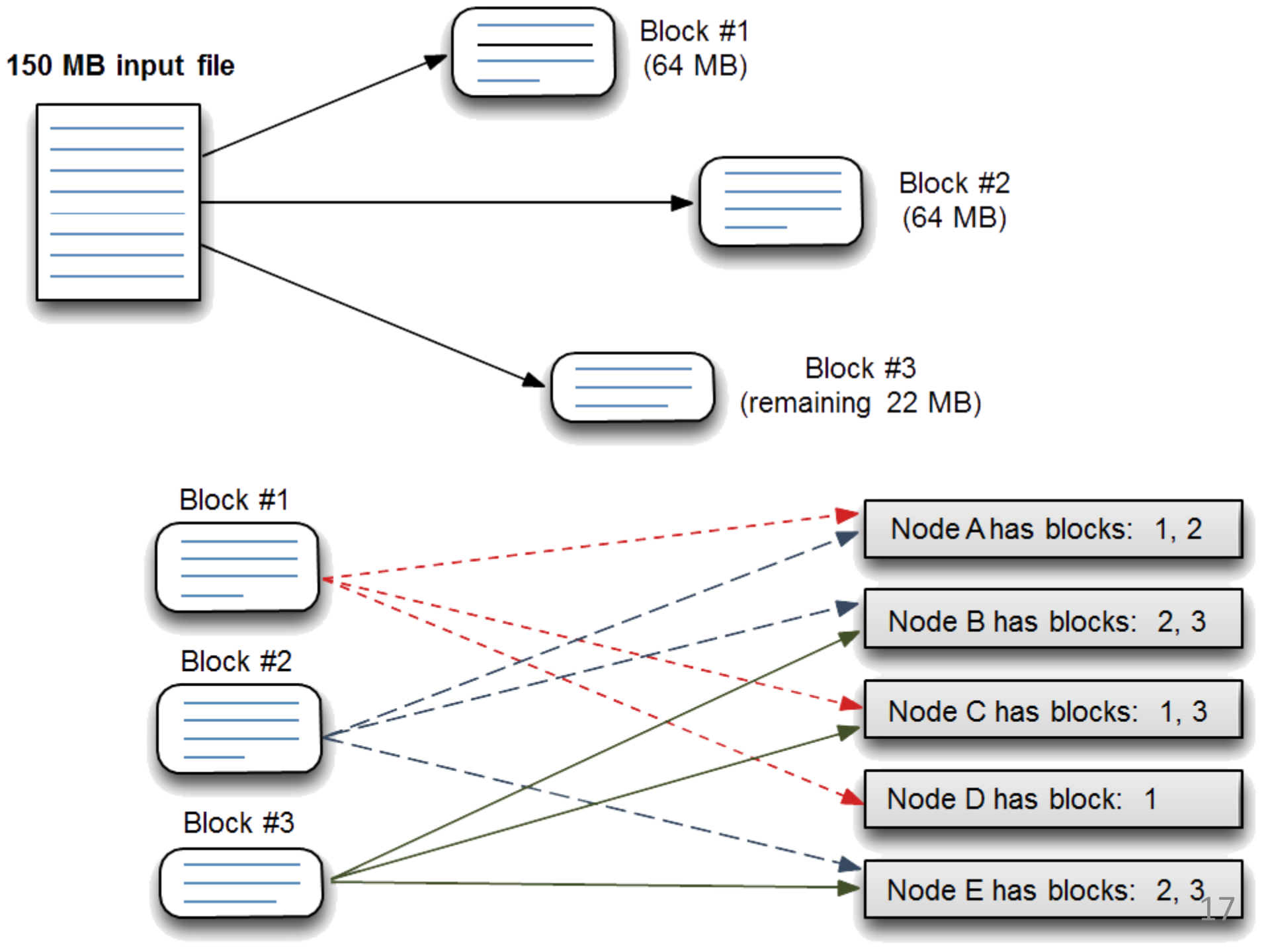


# HDFS is this easy*

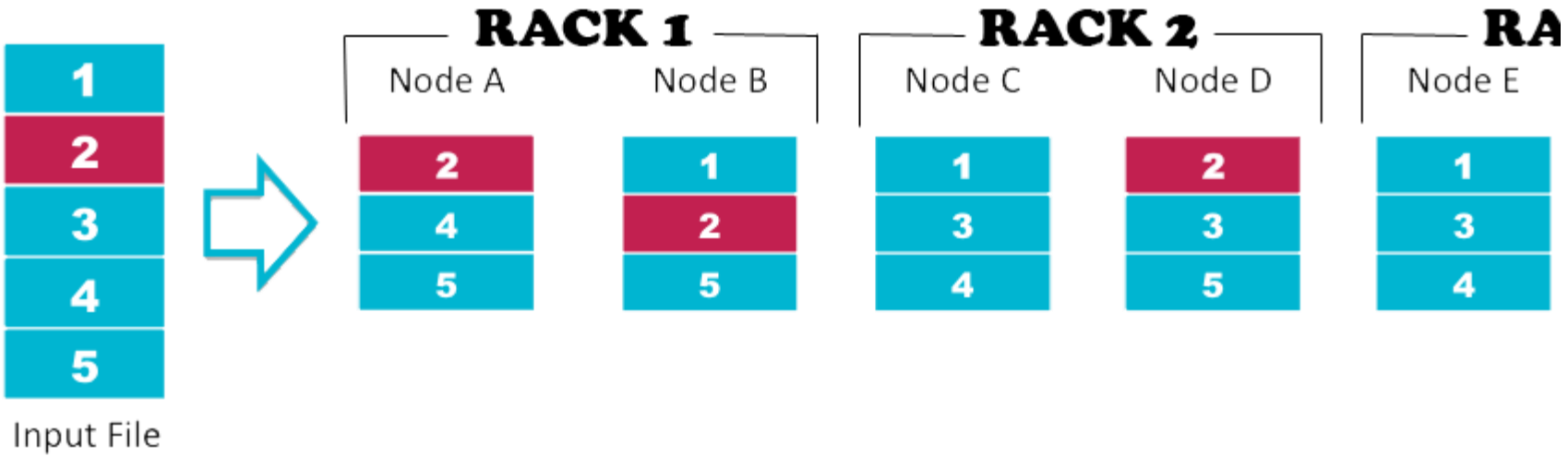


# Client File Write into HDFS




150 MB input file
Block #1
(64 MB)
Block #2
(64 MB)
Block #3
(remaining 22 MB)
Block #1
Block #2
Block #3
Node A has blocks: 1, 2
Node B has blocks: 2, 3
Node C has blocks: 1, 3
Node D has block: 1
Node E has blocks: 2, 3

# Rack Awareness



# Hadoop Distributed File System (HDFS)

Original Slides by
Dhruba Borthakur
Apache Hadoop Project Management Committee

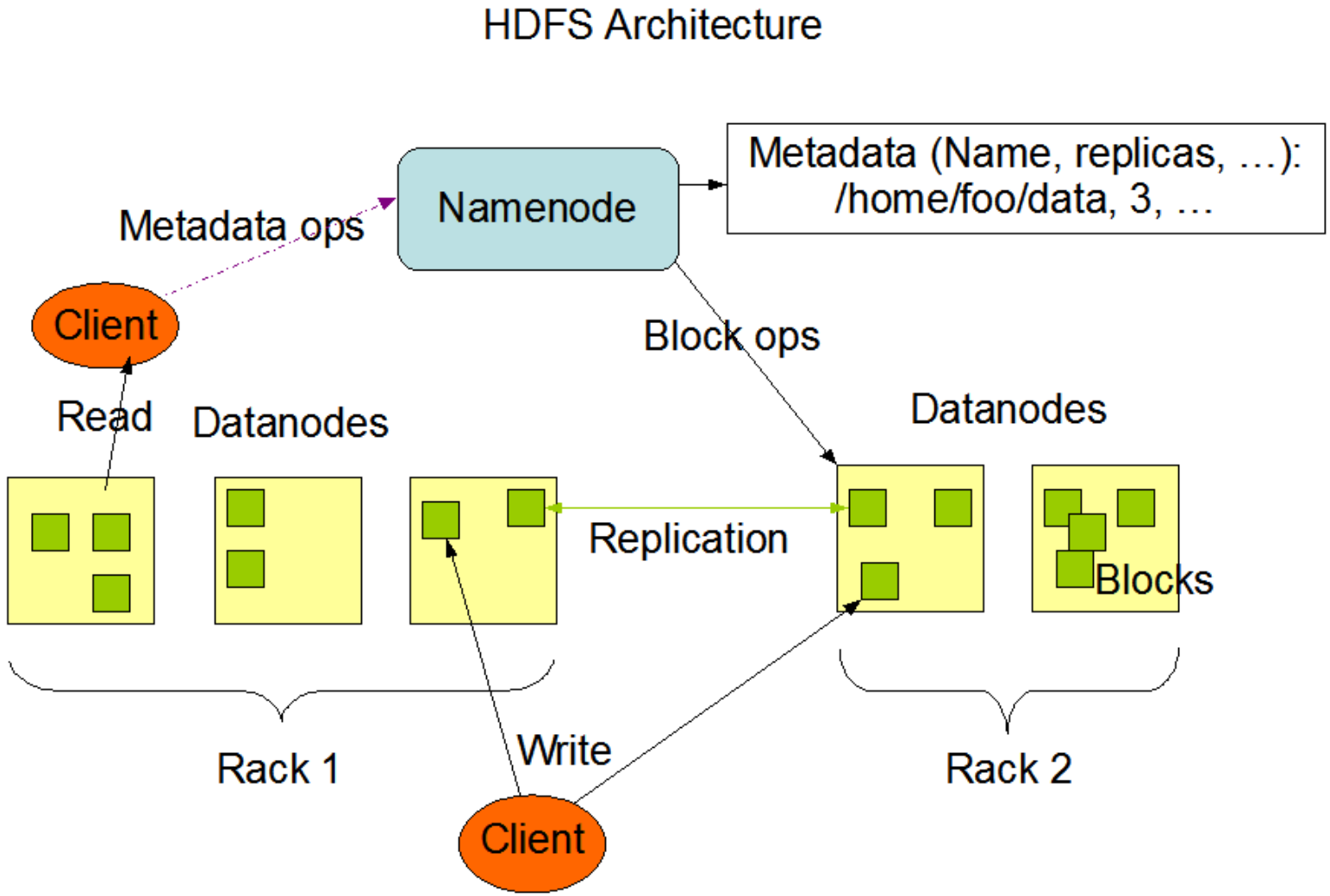
# Goals of HDFS

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

# Distributed File System

- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode

# HDFS Architecture



# Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

# NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

# DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

# Block Placement

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replicas
- Would like to make this policy pluggable

# Heartbeats

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

# Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

# Data Correctness

- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 bytes
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

# NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- Need to develop a real HA solution

# Data Pipelining

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next node in the Pipeline
- When all replicas are written, the Client moves on to write the next block in file

# Rebalancer

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Cluster is online when Rebalancer is active
  - Rebalancer is throttled to avoid network congestion
  - Command line tool

# Secondary NameNode

- Copies FsImage and Transaction Log from Namenode to a temporary directory
- Merges FSImage and Transaction Log into a new FSImage in temporary directory
- Uploads new FSImage to the NameNode
  - Transaction Log on NameNode is purged

# User Interface

- Commads for HDFS User:
  - hadoop dfs -mkdir /foodir
  - hadoop dfs -cat /foodir/myfile.txt
  - hadoop dfs -rm /foodir/myfile.txt
- Commands for HDFS Administrator
  - hadoop dfsadmin -report
  - hadoop dfsadmin -decommision datanodename
- Web Interface
  - http://host:port/dfshealth.jsp

# MapReduce

Original Slides by

Owen O'Malley (Yahoo!)

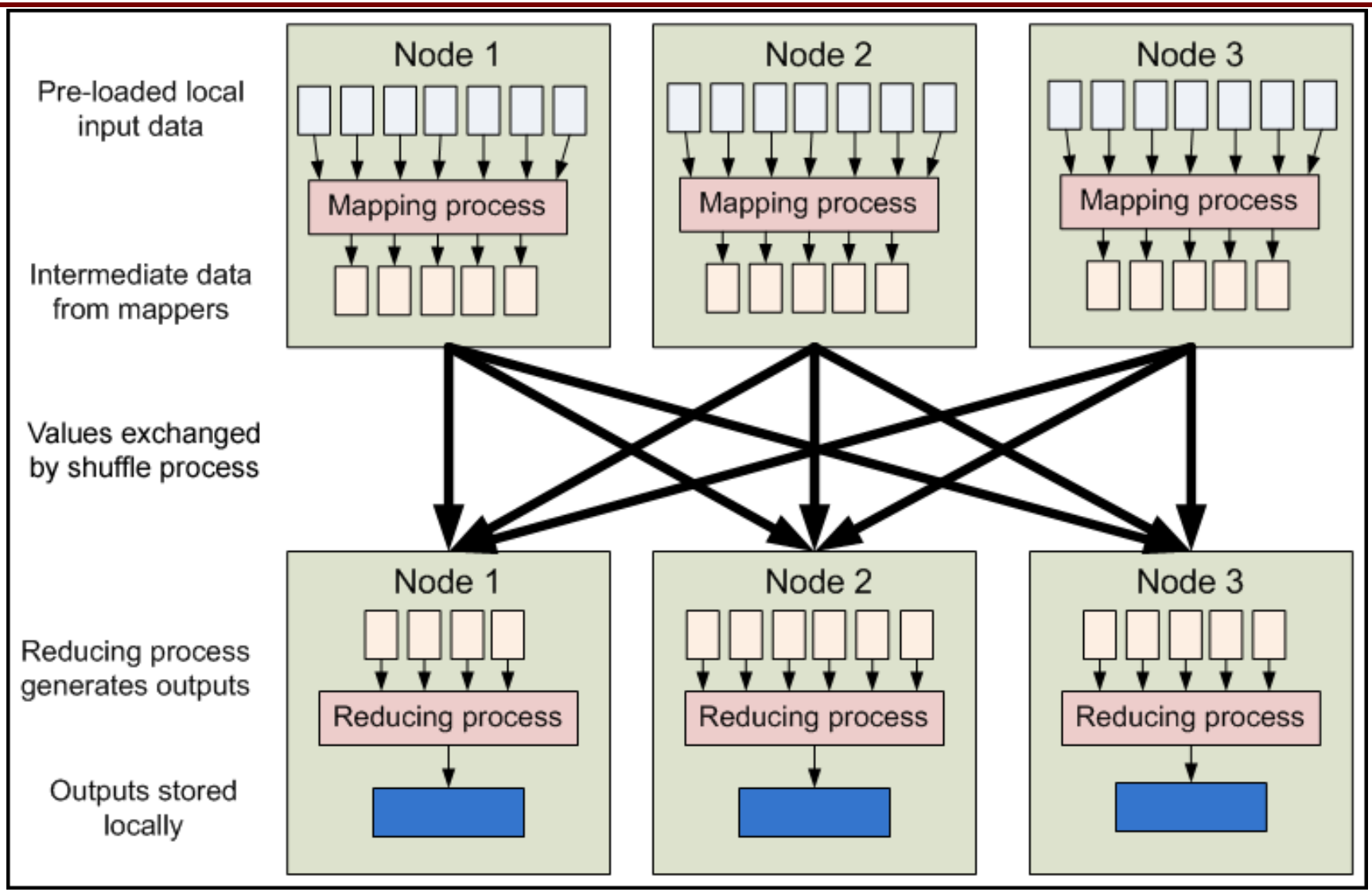
&

Christophe Bisciglia, Aaron Kimball & Sierra Michells-Slettvet

# MapReduce - What?

- MapReduce is a programming model for efficient distributed computing
- It works like a Unix pipeline
  - cat input | grep | sort | uniq -c | cat > output
  - **Input | Map |** Shuffle & Sort **| Reduce | Output**
- Efficiency from
  - Streaming through data, reducing seeks
  - Pipelining
- A good fit for a lot of applications
  - Log processing
  - Web index building

# MapReduce - Dataflow



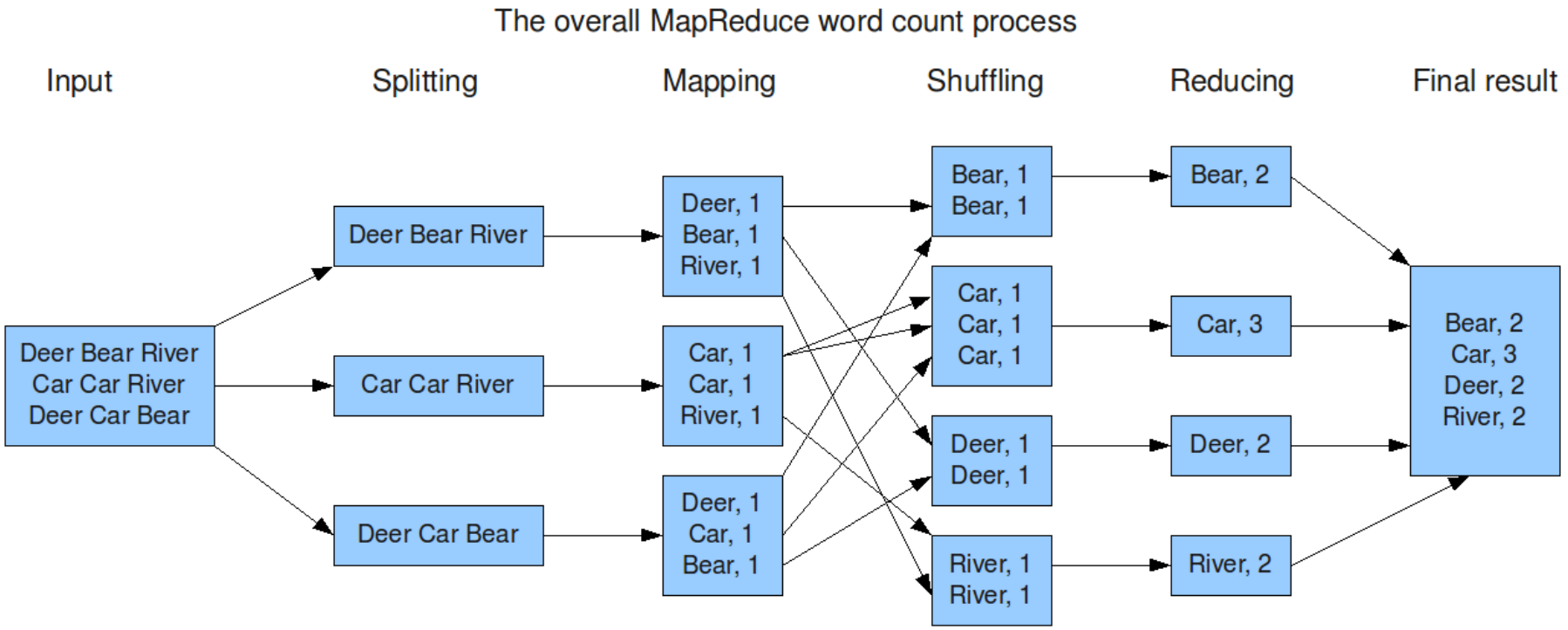
# MapReduce - Features

- Fine grained Map and Reduce tasks
  - Improved load balancing
  - Faster recovery from failed tasks
- Automatic re-execution on failure
  - In a large cluster, some nodes are always slow or flaky
  - Framework re-executes failed tasks
- Locality optimizations
  - With large data, bandwidth to data is a problem
  - Map-Reduce + HDFS is a very effective solution
  - Map-Reduce queries HDFS for locations of input data
  - Map tasks are scheduled close to the inputs when possible

# Word Count Example

- Mapper
  - Input: value: lines of text of input
  - Output: key: word, value: 1
- Reducer
  - Input: key: word, value: set of counts
  - Output: key: word, value: sum
- Launching program
  - Defines this job
  - Submits job to cluster

# Word Count Dataflow



# Word Count Mapper

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable,Text,Text,IntWritable> {
  private static final IntWritable one = new IntWritable(1);
  private Text word  = new Text();

  public static void map(LongWritable key, Text value,
    OutputCollector<Text,IntWritable> output, Reporter reporter) throws
    IOException {
    String line = value.toString();
    StringTokenizer = new StringTokenizer(line);
    while(tokenizer.hasNext()) {
      word.set(tokenizer.nextToken());
      output.collect(word,one);
      }
    }
}
```

# Word Count Reducer

```
public static class Reduce extends MapReduceBase implements
    Reducer<Text,IntWritable,Text,IntWritable> {
public static void map(Text key, Iterator<IntWritable> values,
    OutputCollector<Text,IntWritable> output, Reporter reporter) throws
    IOException {
      int sum = 0;
      while(values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
 }
```

# Word Count Example

- Jobs are controlled by configuring *JobConfs*
- JobConfs are maps from attribute names to string values
- The framework defines attributes to control how the job is executed
  - conf.set(“mapred.job.name”, “MyApp”);
- Applications can add arbitrary values to the JobConf
  - conf.set(“my.string”, “foo”);
  - conf.set(“my.integer”, 12);
- JobConf is available to all tasks

# Putting it all together

- Create a launching program for your application
- The launching program configures:
  - The *Mapper* and *Reducer* to use
  - The output key and value types (input types are inferred from the *InputFormat*)
  - The locations for your input and output
- The launching program then submits the job and typically waits for it to complete

# Putting it all together

```
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducer(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
Conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);
```

# Input and Output Formats

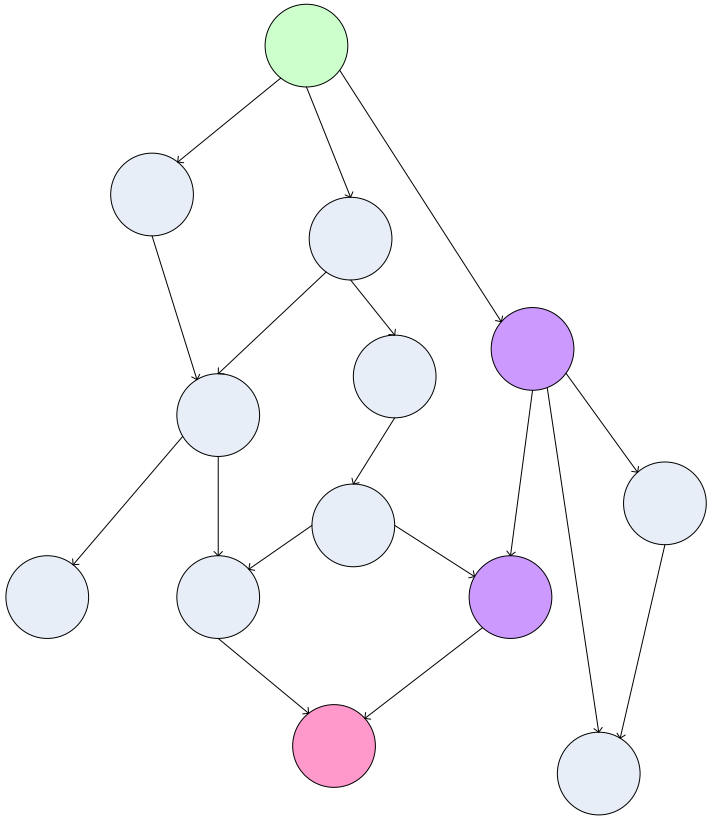
- A Map/Reduce may specify how it's input is to be read by specifying an *InputFormat* to be used
- A Map/Reduce may specify how it's output is to be written by specifying an *OutputFormat* to be used
- These default to *TextInputFormat* and *TextOutputFormat*, which process line-based text data
- Another common choice is *SequenceFileInputFormat* and *SequenceFileOutputFormat* for binary data
- These are file-based, but they are not required to be

# How many Maps and Reduces

- Maps
  - Usually as many as the number of HDFS blocks being processed, this is the default
  - Else the number of maps can be specified as a hint
  - The number of maps can also be controlled by specifying the *minimum split size*
  - The actual sizes of the map inputs are computed by:
    - *max(min(block_size,data/#maps), min_split_size*
- Reduces
  - Unless the amount of data being processed is small
    - *0.95*num_nodes*mapred.tasktracker.tasks.maximum*

# Finding the Shortest Path

- A common graph search application is finding the shortest path from a start node to one or more target nodes
- Commonly done on a single machine with *Dijkstra's Algorithm*
- Can we use BFS (breadth first search) to find the shortest path via MapReduce?



# Finding the Shortest Path: Intuition

- We can define the solution to this problem inductively
  - DistanceTo(startNode) = 0
  - For all nodes *n* directly reachable from startNode, DistanceTo(*n*) = 1
  - For all nodes *n* reachable from some other set of nodes *S*,

    DistanceTo(*n*) = 1 + min(DistanceTo(*m*), $m \in S$)

# From Intuition to Algorithm

- A map task receives a node *n* as a key, and (*D*, *points-to*) as its value
  - *D* is the distance to the node from the start
  - *points-to* is a list of nodes reachable from *n*
- $\forall$p $\in$ points-to, emit (p, D+1)
- Reduces task gathers possible distances to a given *p* and selects the minimum one

# What This Gives Us

- This MapReduce task can advance the known frontier by one hop
- To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration
  - Problem: Where'd the *points-to* list go?
  - Solution: Mapper emits (*n*, *points-to*) as well

# Blow-up and Termination

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as the frontier advances
- Does this ever terminate?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should emit (*n*,*D*) to ensure that “current distance” is carried into the reducer

# Hadoop Related Subprojects

- Pig
  - High-level language for data analysis
- HBase
  - Table storage for semi-structured data
- Zookeeper
  - Coordinating distributed applications
- Hive
  - SQL-like Query language and Metastore
- Mahout
  - Machine learning

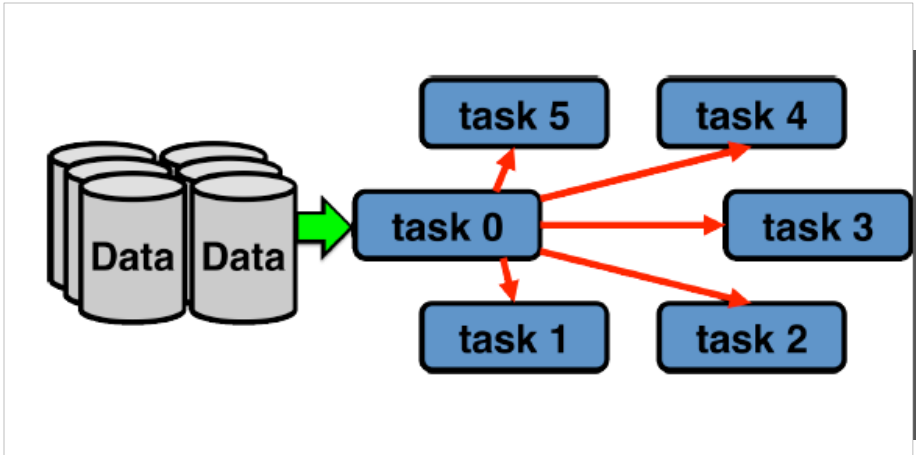
# Hadoop & HPC

# Hadoop and HPC

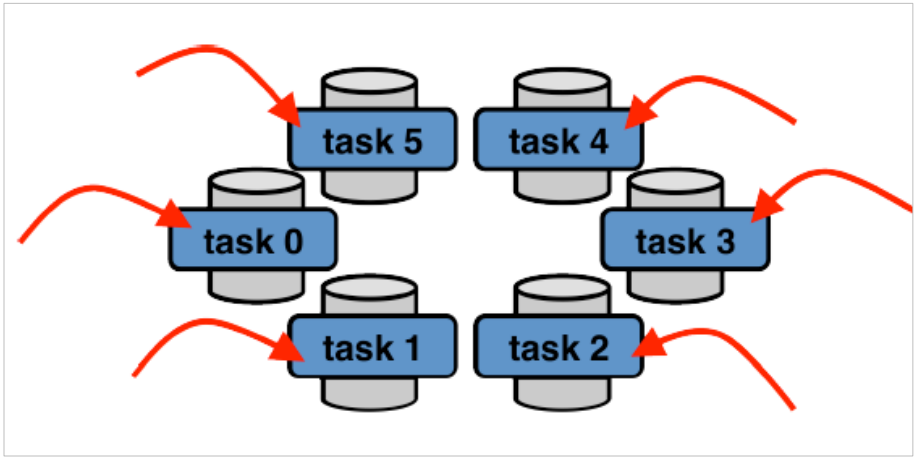
- **PROBLEM:** domain scientists/researchers aren't using Hadoop“
  - Hadoop is commonly used in the data analytics industry but not so common in domain science academic areas.“
  - Java is *not* always high-performance“
  - Hurdles for domain scientists to learn Java, Hadoop tools.
- **SOLUTION:** make Hadoop easier for HPC users“
  - use existing HPC clusters and software"
  - use Perl/Python/C/C++/Fortran instead of Java"
  - make starting Hadoop as easy as possible

# Compute: Traditional vs. Data-Intensive

- **Traditional HPC**
  - CPU-bound problems
  - Solution: OpenMP and MPI-based parallelism




- **Data-Intensive**
  - IO-bound problems
  - Solution: Map/reduce based parallelism



# Architecture for Both Workloads

## PROs

- High-speed interconnect
- Complementary object storage
- Fast CPUs, RAM
- Less faulty

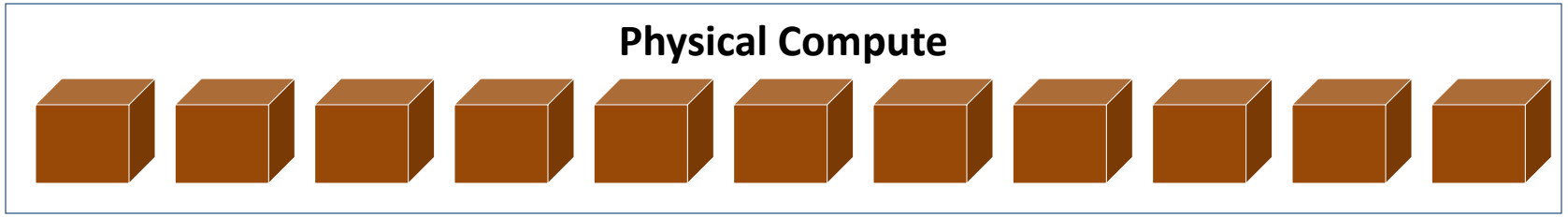
## CONs

- Nodes aren't storage rich
- Transferring data between HDFS and object storage*

- unless using Lustre, S3, etc backends

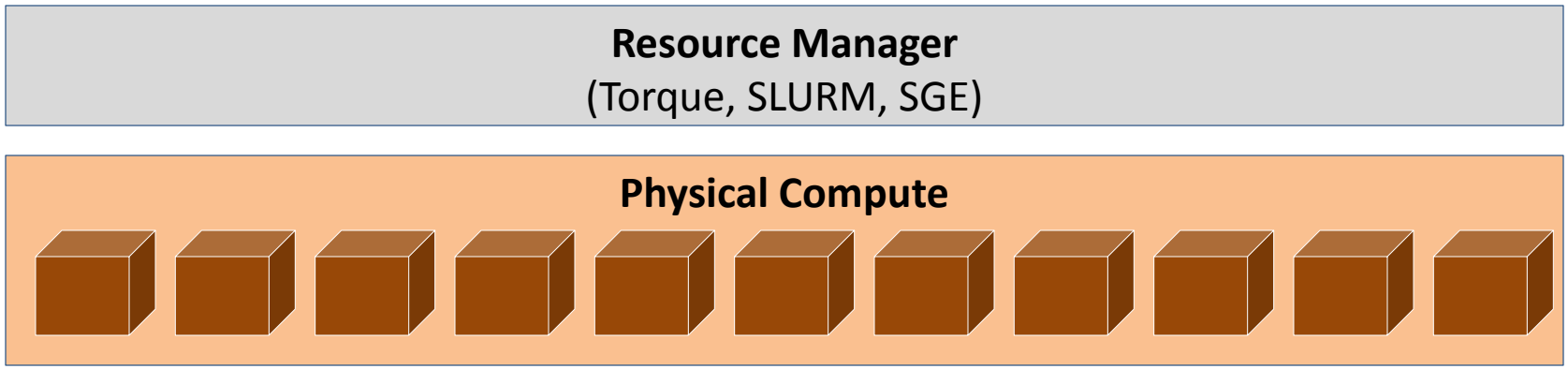
# Hadoop & HPC

- Add Data Analysis to Existing Compute Infrastructure



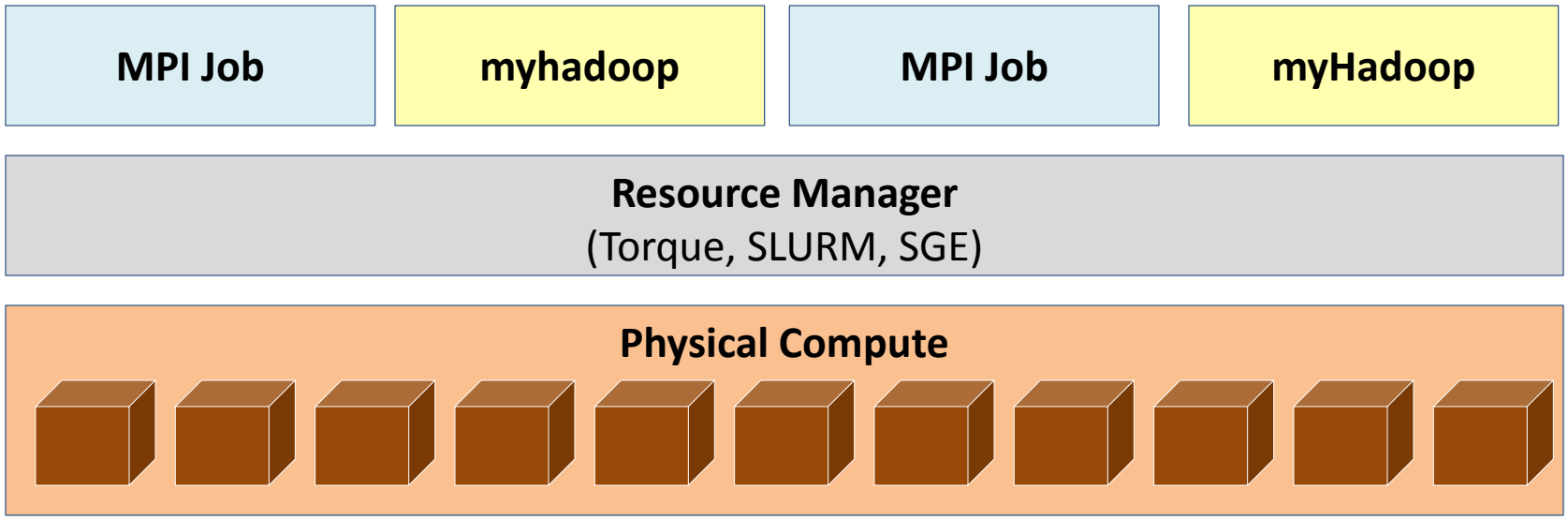
# Hadoop & HPC

- Add Data Analysis to Existing Compute Infrastructure



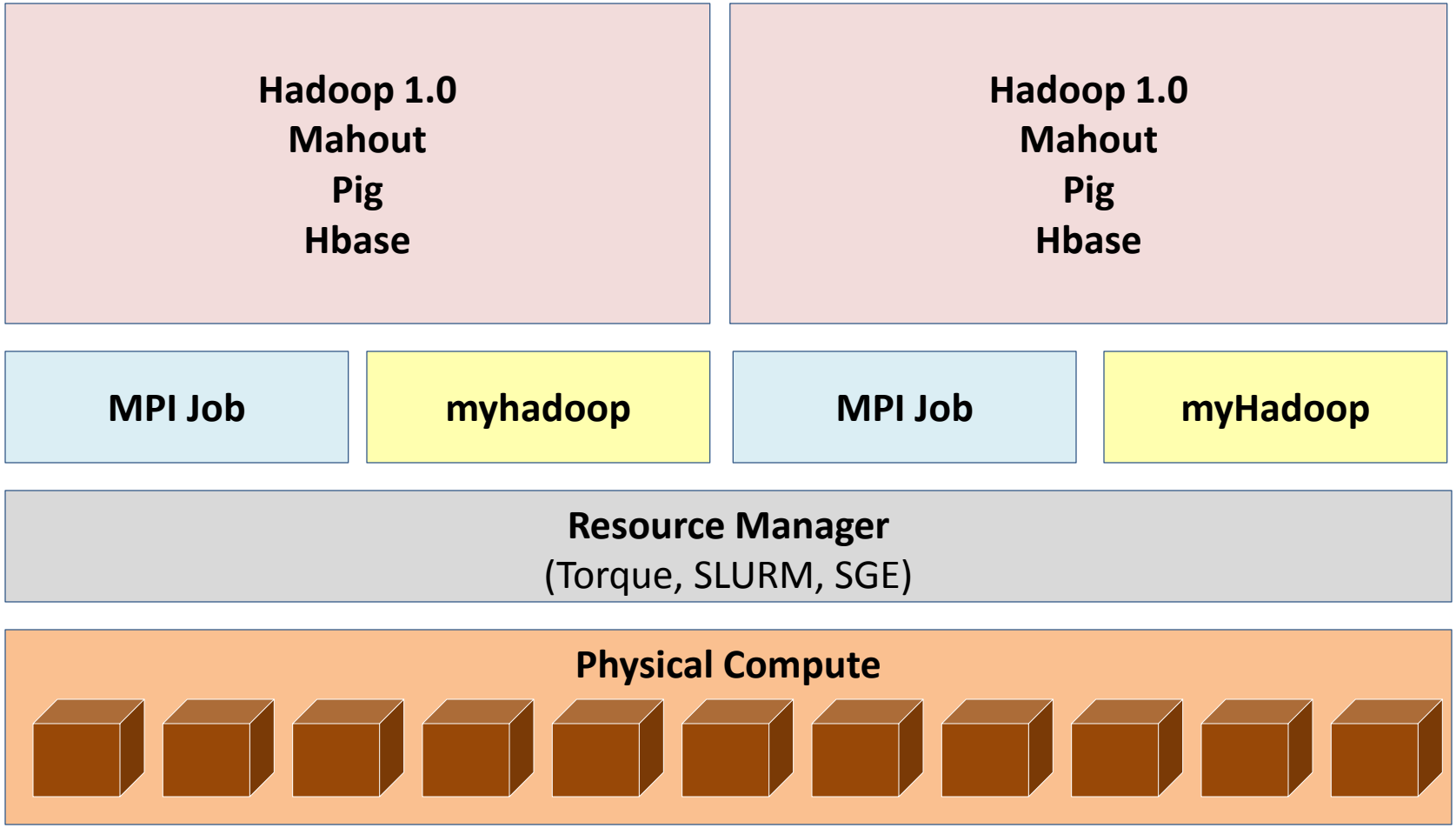
# Hadoop & HPC

- Add Data Analysis to Existing Compute Infrastructure



# Hadoop & HPC

- Add Data Analysis to Existing Compute Infrastructure



# myHadoop: 3-step Install

1. **Download Apache Hadoop 1.x and myHadoop 0.30**
   - `$ wget http://apache.cs.utah.edu/hadoop/common/hadoop-1.2.1/hadoop-1.2.1-bin.tar.gz`
   - `$ wget http://users.sdsc.edu/~glockwood/files/myhadoop-0.30.tar.gz`
2. **Unpack both Hadoop and myHadoop!**
   - `$ tar zxvf hadoop-1.2.1-bin.tar.gz`
   - `$ tar zxvf myhadoop-0.30.tar.gz`
3. **Apply myHadoop patch to Hadoop!**
   - `$ cd hadoop-1.2.1/conf`
   - `$ patch < ../myhadoop-0.30/myhadoop-1.2.1.patch`

# myHadoop: 3-step Install

1. **Set a few environment variables**

```
# sets HADOOP_HOME, JAVA_HOME, and PATH
$ module load hadoop
$ export HADOOP_CONF_DIR=$HOME/mycluster.conf
```

2. **Run myhadoop-configure.sh to set up Hadoop**

```
$ myhadoop-configure.sh -s
/scratch/$USER/$PBS_JOBID!
```

3. **Start cluster with Hadoop's start-all.sh**

```
$ start-all.sh!
```

# Advanced Features - Useability!

- **System-wide default configurations**
  - `myhadoop-0.30/conf/myhadoop.conf`
  - `MH_SCRATCH_DIR` – specify location of node-local storage for all users"
  - `MH_IPOIB_TRANSFORM` – specify regex to transform node hostnames into IP over InfiniBand hostnames"
- **Users can remain totally ignorant of scratch disks and InfiniBand**
- **Literally define** `HADOOP_CONF_DIR` **and run** `myhadoop-configure.sh` **with no parameters – myHadoop figures out everything else**

# Advanced Features - Useability!

- **Parallel filesystem support!**
  - HDFS on Lustre via myHadoop persistent mode (-p)"
  - Direct Lustre support (IDH)"
  - No performance loss at smaller scales for HDFS on Lustre"
- **Resource managers supported in unified framework:**
  - Torque 2.x and 4.x – Tested on SDSC Gordon"
  - SLURM 2.6 – Tested on TACC Stampede"
  - Grid Engine"
  - Can support LSF, PBSpro, Condor easily (need testbeds)

# Scientific Computing - BioPIG

- Hadoop based analytic toolkit for large-scale sequence data
- Built using Apache Hadoop MapReduce and Pig Data Flow language.
- Benefits – reduced development time for parallel bioinformatics apps, good scaling with data size, portability.
- Run on systems at NERSC (Magellan), Amazon
- https://sites.google.com/a/lbl.gov/biopig/

Questions?