

High-level languages

Crunch

Cascading



Pig



Predictive analytics

RHadoop

RHIPE



R



Miscellaneous



Sqoop



HBASE



HDFS



MapReduce



Hadoop

Hadoop and related technologies

Running Hadoop on a Single-Node Cluster

- In this lecture, we will describe the required steps for setting up a pseudo-distributed, single-node Hadoop cluster backed by the Hadoop Distributed File System, running on CygWin and Ubuntu Linux.
- Hadoop is a framework written in Java for running applications on large clusters of commodity hardware and incorporates features similar to those of the Google File System (GFS) and of the MapReduce computing paradigm.
- Hadoop's HDFS is a highly fault-tolerant distributed file system and, like Hadoop in general, designed to be deployed on low-cost hardware.
- It provides high throughput access to application data and is suitable for applications that have large data sets.

Hadoop Installation

- The main goal of this lecture is to get a simple Hadoop installation up and running so that you can play around with the software and learn more about it.
- This tutorial has been tested with the following software versions:
 - Ubuntu Linux
 - Cygwin



Cluster of machines running Hadoop at Yahoo! (Source: Yahoo!)

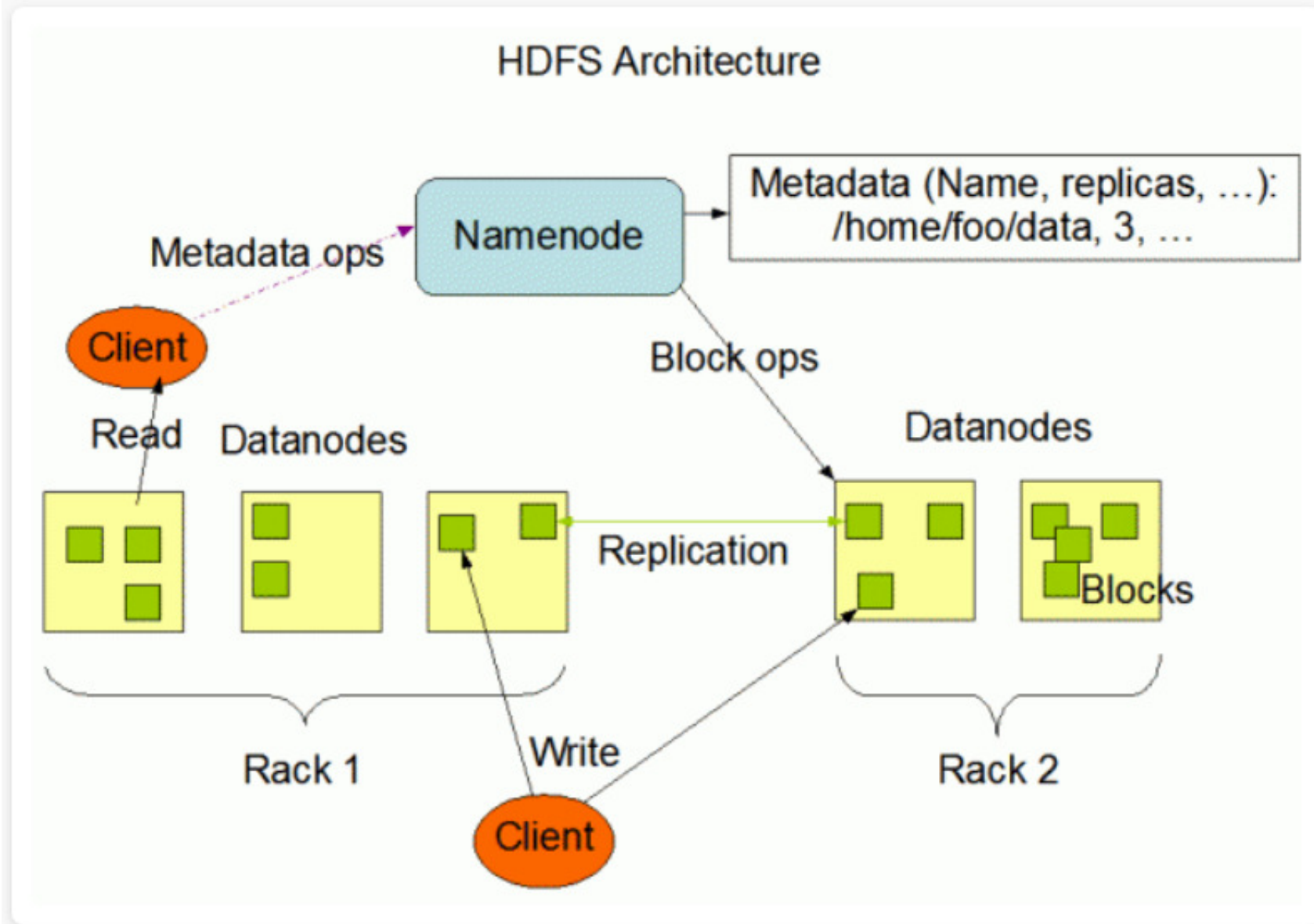
Installation

- Download Hadoop from the Apache Download Mirrors and extract the contents of the Hadoop package to a location of your choice.
- I picked `/usr/local/hadoop`.
- `$ cd /usr/local`
- `$ sudo tar xzf hadoop-1.0.3.tar.gz`
- `$ sudo mv hadoop-1.0.3 hadoop`
- `$ sudo chown -R hduser:hadoop hadoop`


Hadoop Distributed File System (HDFS)

- Before we continue let's briefly learn a bit more about Hadoop's distributed file system.
- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.
- It has many similarities with existing distributed file systems.
- However, the differences from other distributed file systems are significant.
 - HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware.
 - HDFS provides high throughput access to application data and is suitable for applications that have large data sets.
 - HDFS relaxes a few POSIX requirements to enable streaming access to file system data.
 - HDFS was originally built as infrastructure for the Apache Nutch web search engine project.
 - HDFS is part of the Apache Hadoop project, which is part of the Apache Lucene project.

The following picture gives an overview of the most important HDFS components.



Configuration

- 
- Our goal in this tutorial is a single-node setup of Hadoop.

Prerequisites



□ **Supported Platforms**

- ▣ GNU/Linux is supported as a development and production platform.
- ▣ Hadoop has been demonstrated on GNU/Linux clusters with 2000 nodes.
- ▣ Win32 is supported as a development platform.
- ▣ Distributed operation has not been well tested on Win32, so it is not supported as a production platform.

□ **Required Software**

- ▣ Java 1.6 or later, preferably from Sun, must be installed
- ▣ ssh must be installed and sshd must be running to use the Hadoop scripts that manage remote Hadoop daemons.

□ **Additional requirements for Windows include:**

- ▣ Cygwin - Required for shell support in addition to the required software above.
- ▣ Vmware or VirtualBox to run Linux.

Installing SSH

- **on Ubuntu Linux:**

 - \$ sudo apt-get install ssh

 - \$ sudo apt-get install rsync

- On Windows, if you did not install the required software when you installed cygwin, start the cygwin installer and select the packages:

 - openssh - the Net category

- **Setup passphraseless ssh**

- Now check that you can ssh to the localhost without a passphrase:

 - \$ ssh localhost

- If you cannot ssh to localhost without a passphrase, execute the following commands:

 - \$ ssh-keygen -t dsa -P "" -f ~/.ssh/id_dsa

 - \$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys

hadoop-env.sh

The only required environment variable we have to configure for Hadoop in this tutorial is `JAVA_HOME`. Open `conf/hadoop-env.sh` in the editor of your choice (if you used the installation path in this tutorial, the full path is `/usr/local/hadoop/conf/hadoop-env.sh`) and set the `JAVA_HOME` environment variable to the Sun JDK/JRE 6 directory.

Change

```
conf/hadoop-env.sh
1  # The java implementation to use.  Required.
2  # export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

to

```
conf/hadoop-env.sh
1  # The java implementation to use.  Required.
2  export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

Note: If you are on a Mac with OS X 10.7 you can use the following line to set up `JAVA_HOME` in `conf/hadoop-env.sh`.

```
conf/hadoop-env.sh (on Mac systems)
1  # for our Mac users
2  export JAVA_HOME=`/usr/libexec/java_home`
```

Configuration

□ **core-site.xml:**

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

□ **hdfs-site.xml:**

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

□ **conf/mapred-site.xml:**

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
</configuration>
```

Formatting the HDFS filesystem via the NameNode

- The first step to starting up your Hadoop installation is formatting the Hadoop filesystem which is implemented on top of the local filesystem of your “cluster”.
- This is only needed for the first time you set up a Hadoop cluster.
- Don't format a running Hadoop filesystem as you will lose all the data currently in the cluster (in HDFS)!
- To format the filesystem (which simply initializes the directory specified by the `dfs.name.dir` variable), run the command

```
hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format
```

The output will look like this:

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop namenode -format
14/06/10 16:59:56 INFO namenode.NameNode: STARTUP_MSG:
/*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = ubuntu/127.0.1.1
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 0.20.2
*****/
14/06/10 16:59:56 INFO namenode.FSNamesystem: fsOwner=hduser,hadoop
14/06/10 16:59:56 INFO namenode.FSNamesystem: supergroup=supergroup
14/06/10 16:59:56 INFO namenode.FSNamesystem: isPermissionEnabled=true
14/06/10 16:59:56 INFO common.Storage: Image file of size 96 saved in 0 seconds.
14/06/10 16:59:57 INFO common.Storage: Storage directory has been successfully formatted.
14/06/10 16:59:57 INFO namenode.NameNode: SHUTDOWN_MSG:
/*****
SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1
*****/
hduser@ubuntu:/usr/local/hadoop$
```

Starting Single-Node Cluster

```
yusuf@ubuntu:/usr/local/bin/hadoop/sbin$ ./start-all.sh
```

This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh

14/06/25 23:33:13 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable

Starting namenodes on [localhost]

localhost: starting namenode, logging to /usr/local/bin/hadoop/logs/hadoop-yusuf-namenode-ubuntu.out

localhost: starting datanode, logging to /usr/local/bin/hadoop/logs/hadoop-yusuf-datanode-ubuntu.out

Starting secondary namenodes [0.0.0.0]

0.0.0.0: starting secondarynamenode, logging to /usr/local/bin/hadoop/logs/hadoop-yusuf-secondarynamenode-ubuntu.out

14/06/25 23:33:32 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable

starting yarn daemons

starting resourcemanager, logging to /usr/local/bin/hadoop/logs/yarn-yusuf-resourcemanager-ubuntu.out

localhost: starting nodemanager, logging to /usr/local/bin/hadoop/logs/yarn-yusuf-nodemanager-ubuntu.out

JPS is a tool to check if the Hadoop Processes are Running



❑ **yusuf@ubuntu:/usr/local/bin/hadoop/sbin\$ jps**

2945 NameNode

3542 NodeManager

3418 ResourceManager

3067 DataNode

3275 SecondaryNameNode

3902 Jps

Stopping Single-Node Cluster

```
yusuf@ubuntu:/usr/local/bin/hadoop/sbin$ ./stop-all.sh
```

This script is Deprecated. Instead use stop-dfs.sh and stop-yarn.sh

14/06/25 23:37:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

Stopping namenodes on [localhost]

localhost: stopping namenode

localhost: stopping datanode

Stopping secondary namenodes [0.0.0.0]

0.0.0.0: stopping secondarynamenode

14/06/25 23:38:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

stopping yarn daemons

stopping resourcemanager

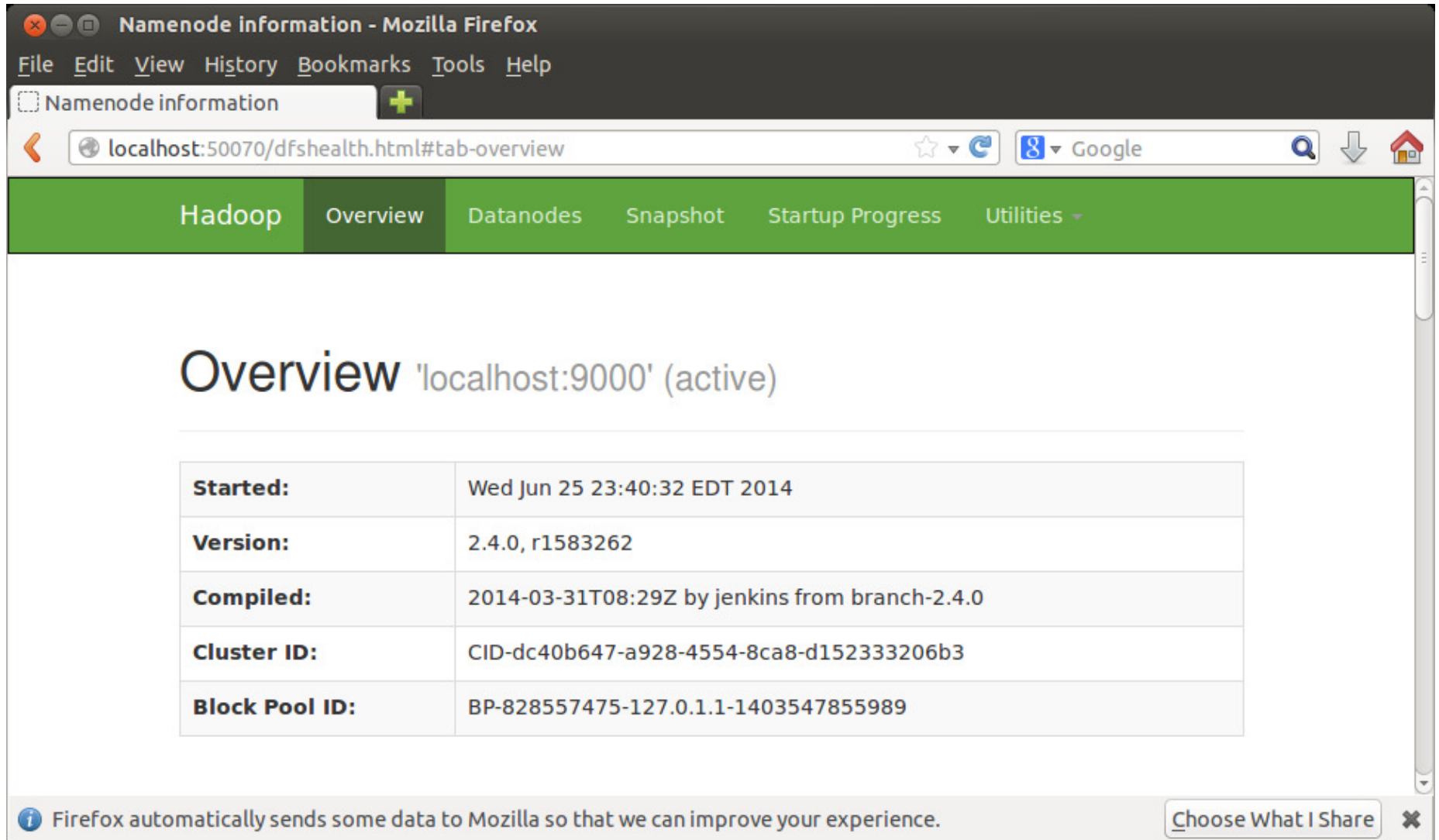
localhost: stopping nodemanager

no proxyserver to stop

Runnig the WordCount Example

```
yusuf@ubuntu:/usr/local/bin/hadoop/bin$ ./hadoop jar
../share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.0.jar wordcount
/yusuf/books /yusuf/books/output
14/06/25 23:45:07 WARN util.NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
14/06/25 23:45:08 INFO Configuration.deprecation: session.id is deprecated. Instead,
use dfs.metrics.session-id
...
14/06/25 23:45:11 INFO mapreduce.Job: Job job_local798365534_0001 running in
uber mode : false
14/06/25 23:45:11 INFO mapreduce.Job: map 0% reduce 100%
14/06/25 23:45:11 INFO mapreduce.Job: Job job_local798365534_0001 completed
successfully
14/06/25 23:45:11 INFO mapreduce.Job: Counters: 32
    File System Counters
        FILE: Number of bytes read=270285
    ...
    Map-Reduce Framework
        ...
```

Hadoop Web Interface

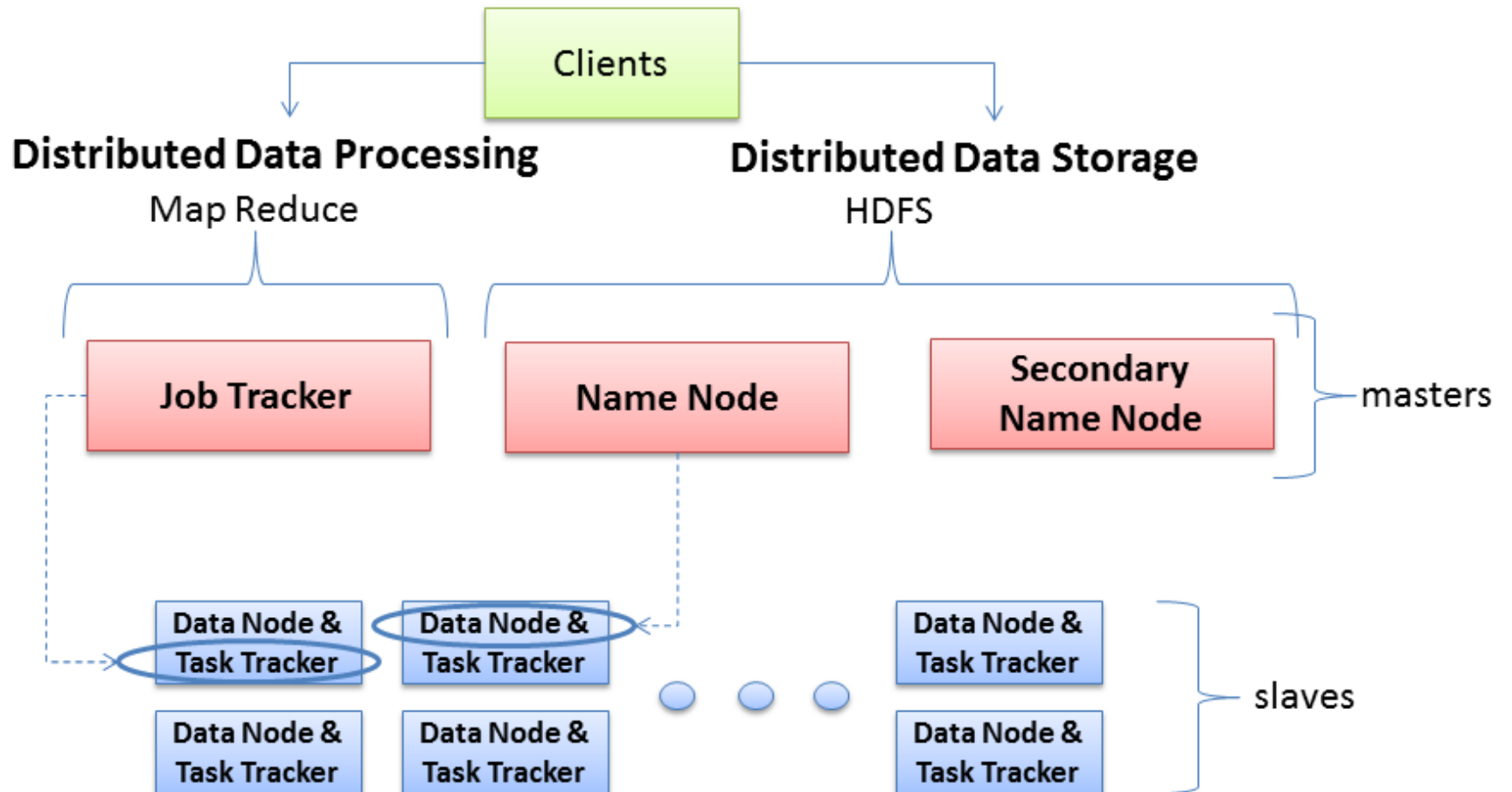


The screenshot shows a Mozilla Firefox browser window titled "Namenode information - Mozilla Firefox". The address bar displays "localhost:50070/dfshealth.html#tab-overview". The browser's menu bar includes File, Edit, View, History, Bookmarks, Tools, and Help. A green navigation bar at the top of the page contains links: Hadoop, Overview, Datanodes, Snapshot, Startup Progress, and Utilities. The main content area is titled "Overview 'localhost:9000' (active)". Below this title is a table with the following information:


Started:	Wed Jun 25 23:40:32 EDT 2014
Version:	2.4.0, r1583262
Compiled:	2014-03-31T08:29Z by jenkins from branch-2.4.0
Cluster ID:	CID-dc40b647-a928-4554-8ca8-d152333206b3
Block Pool ID:	BP-828557475-127.0.1.1-1403547855989

At the bottom of the browser window, a notification bar states: "Firefox automatically sends some data to Mozilla so that we can improve your experience." with a "Choose What I Share" button and a close icon.

Hadoop Server Roles



Three major categories of machine roles in Hadoop

- 
- ❑ Client machines
 - ❑ Masters nodes
 - ❑ Slave nodes

Master Nodes

- The Master nodes oversee the two key functional pieces that make up Hadoop:
 - ▣ storing lots of data (HDFS)
 - ▣ running parallel computations on all that data (Map Reduce).
- The Name Node oversees and coordinates the data storage function (HDFS).
- The Job Tracker oversees and coordinates the parallel processing of data using Map Reduce.

Slave Nodes

- Slave Nodes make up the vast majority of machines and do all the dirty work of storing the data and running the computations.
- Each slave runs both a Data Node and Task Tracker daemon that communicate with and receive instructions from their master nodes.
- The Task Tracker daemon is a slave to the Job Tracker, the Data Node daemon a slave to the Name Node.

Client Machines

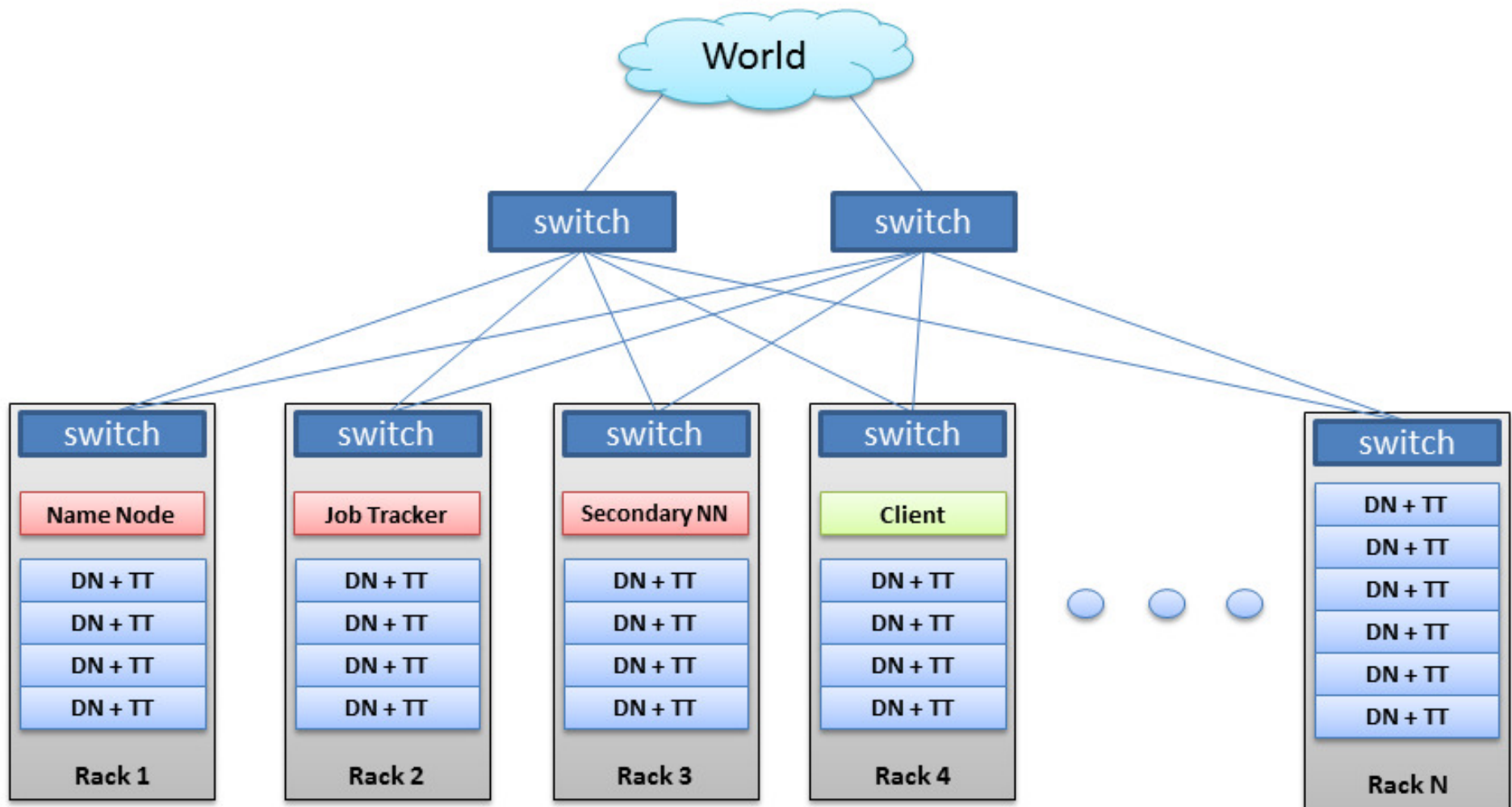


- Client machines have Hadoop installed with all the cluster settings, but are neither a Master or a Slave.
- Instead, the role of the Client machine is to load data into the cluster, submit Map Reduce jobs describing how that data should be processed, and then retrieve or view the results of the job when its finished.
- In smaller clusters (~40 nodes) you may have a single physical server playing multiple roles, such as both Job Tracker and Name Node.
- With medium to large clusters you will often have each role operating on a single server machine.

In real production clusters there is no server virtualization

- In real production clusters there is no server virtualization, no hypervisor layer.
- That would only amount to unnecessary overhead impeding performance.
- Hadoop runs best on Linux machines, working directly with the underlying hardware.
- That said, Hadoop does work in a virtual machine.
- That's a great way to learn and get Hadoop up and running fast and cheap.
- You could run multi-node cluster up and running in VMware Workstation on Windows laptop.

Hadoop Cluster



Typical architecture of a Hadoop cluster

- You will have rack servers populated in racks connected to a top of rack switch usually with 1 or 2 GigE bonded links.
- 10GigE nodes are uncommon but gaining interest as machines continue to get more dense with CPU cores and disk drives.
- The rack switch has uplinks connected to another tier of switches connecting all the other racks with uniform bandwidth, forming the cluster.
- The majority of the servers will be Slave nodes with lots of local disk storage and moderate amounts of CPU and DRAM.
- Some of the machines will be Master nodes that might have a slightly different configuration favoring more DRAM and CPU, less local storage.

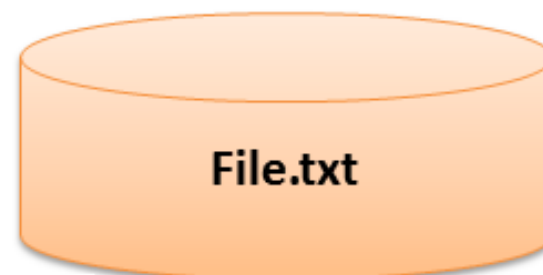
Typical Workflow

- Load data into the cluster (HDFS writes)
- Analyze the data (Map Reduce)
- Store results in the cluster (HDFS writes)
- Read the results from the cluster (HDFS reads)

Sample Scenario:

How many times did our customers type the word **“Refund”** into emails sent to customer service?

Huge file containing all emails sent
to customer service



Why did Hadoop come to exist?

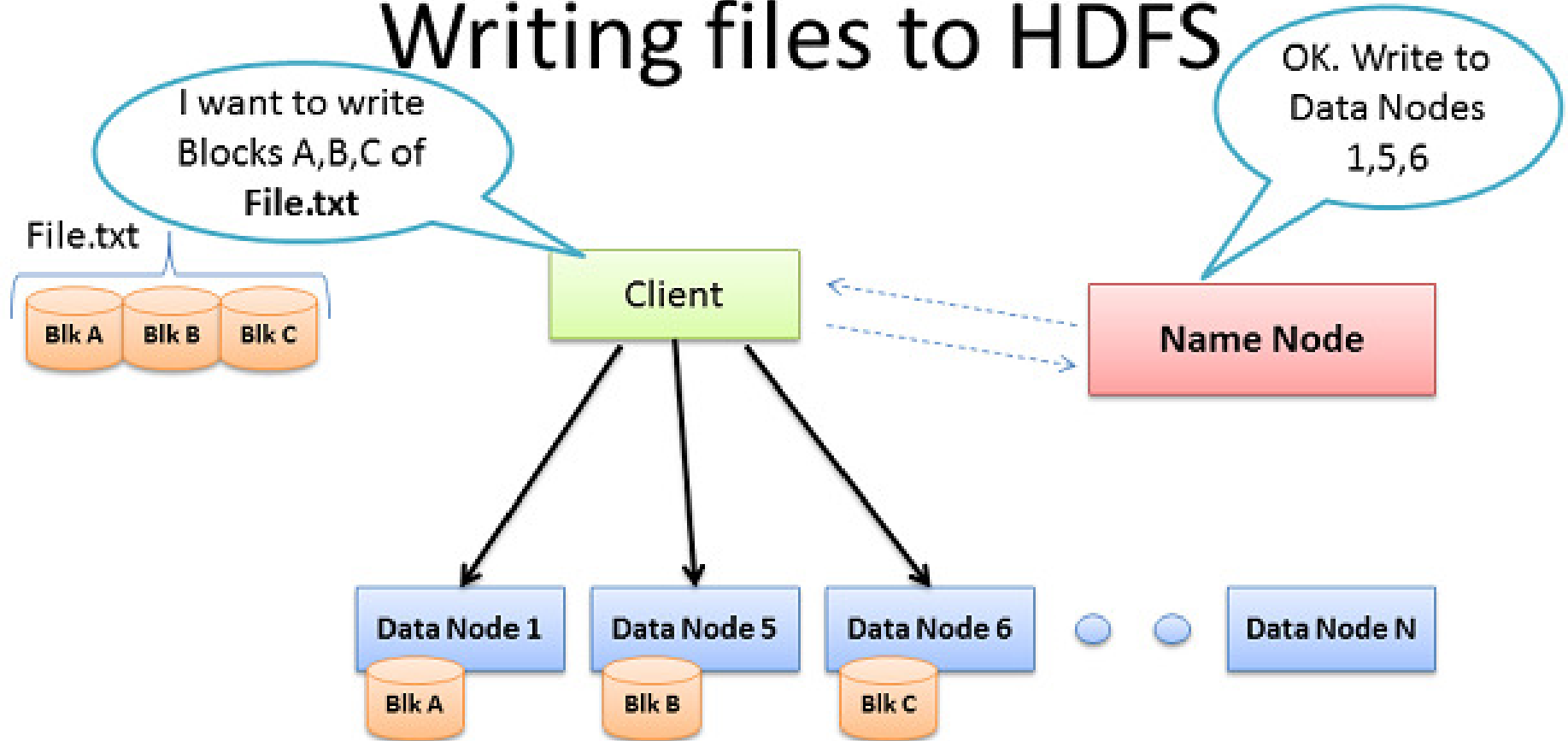
What problem does it solve?

- Simply put, businesses and governments have a tremendous amount of data that needs to be analyzed and processed very quickly.
- If we can chop that huge chunk of data into small chunks and spread it out over many machines, and have all those machines process their portion of the data in parallel — we can get answers extremely fast — and that, in a nutshell, is what Hadoop does.

Simple Example

- In our simple example, we'll have a huge data file containing emails sent to the customer service department.
- We want a quick snapshot to see how many times the word “Refund” was typed by my customers.
- This might help me to anticipate the demand on our returns and exchanges department, and staff it appropriately.
- It's a simple word count exercise.
- The Client will load the data into the cluster (File.txt), submit a job describing how to analyze that data (word count), the cluster will store the results in a new file (Results.txt), and the Client will read the results file.

Writing files to HDFS



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

Loading files into Hadoop Clusters


- Your Hadoop cluster is useless until it has data, so we'll begin by loading our huge File.txt into the cluster for processing.
- The goal here is fast parallel processing of lots of data.
- To accomplish that we need as many machines as possible working on this data all at once.
- To that end, the Client is going to break the data file into smaller “Blocks”, and place those blocks on different machines throughout the cluster.
- The more blocks we have, the more machines that will be able to work on this data in parallel.
- At the same time, these machines may be prone to failure, so I want to insure that every block of data is on multiple machines at once to avoid data loss.

Replication

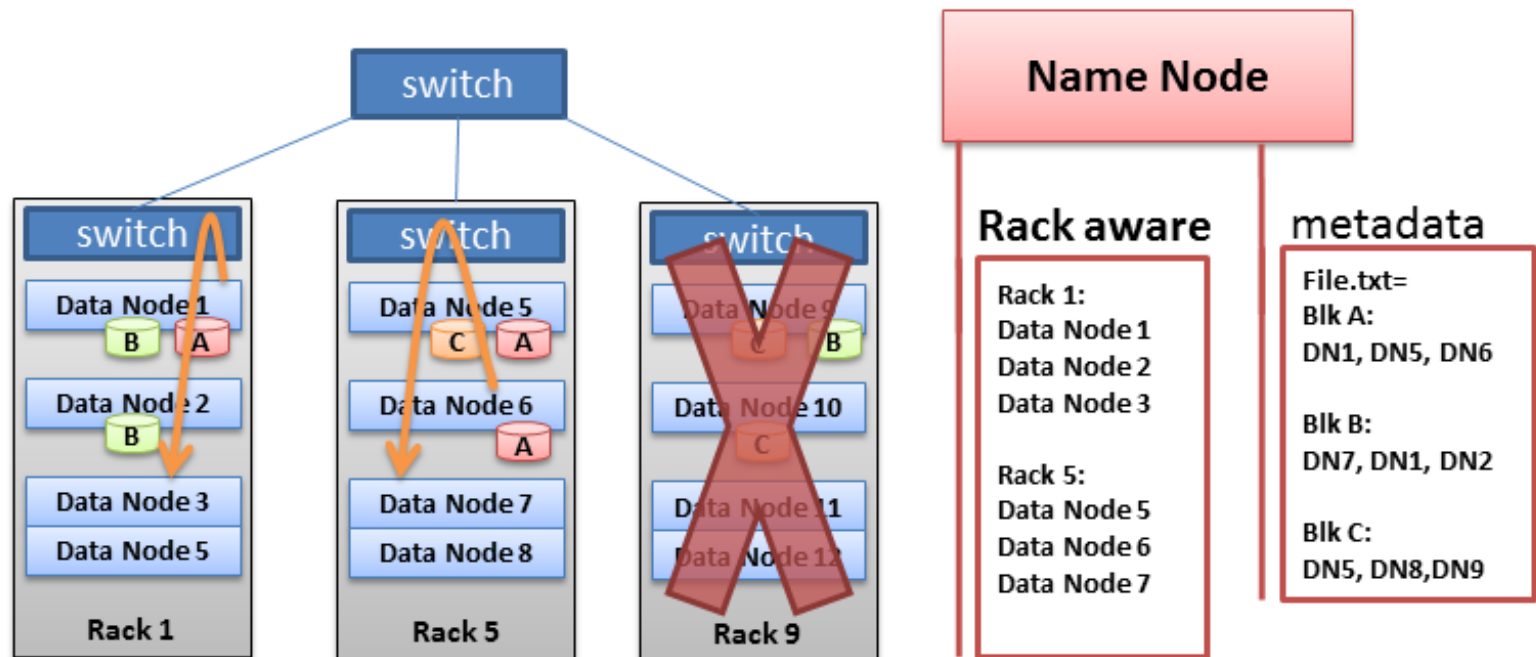


- So each block will be replicated in the cluster as its loaded.
- The standard setting for Hadoop is to have (3) copies of each block in the cluster.
- This can be configured with the `dfs.replication` parameter in the file `hdfs-site.xml`.

Name Node

- 
- The Client breaks File.txt into (3) Blocks.
 - For each block, the Client consults the Name Node (usually TCP port 9000) and receives a list of (3) Data Nodes that should have a copy of this block.
 - The Client then writes the block directly to the Data Node (usually TCP port 50010).
 - The receiving Data Node replicates the block to other Data Nodes, and the cycle repeats for the remaining blocks.
 - The Name Node is not in the data path.
 - The Name Node only provides the map of where data is and where data should go in the cluster (file system metadata).

Hadoop Rack Awareness



- Never loose all data if entire rack fails
- Keep bulky flows in-rack when possible
- Assumption that in-rack is higher bandwidth, lower latency

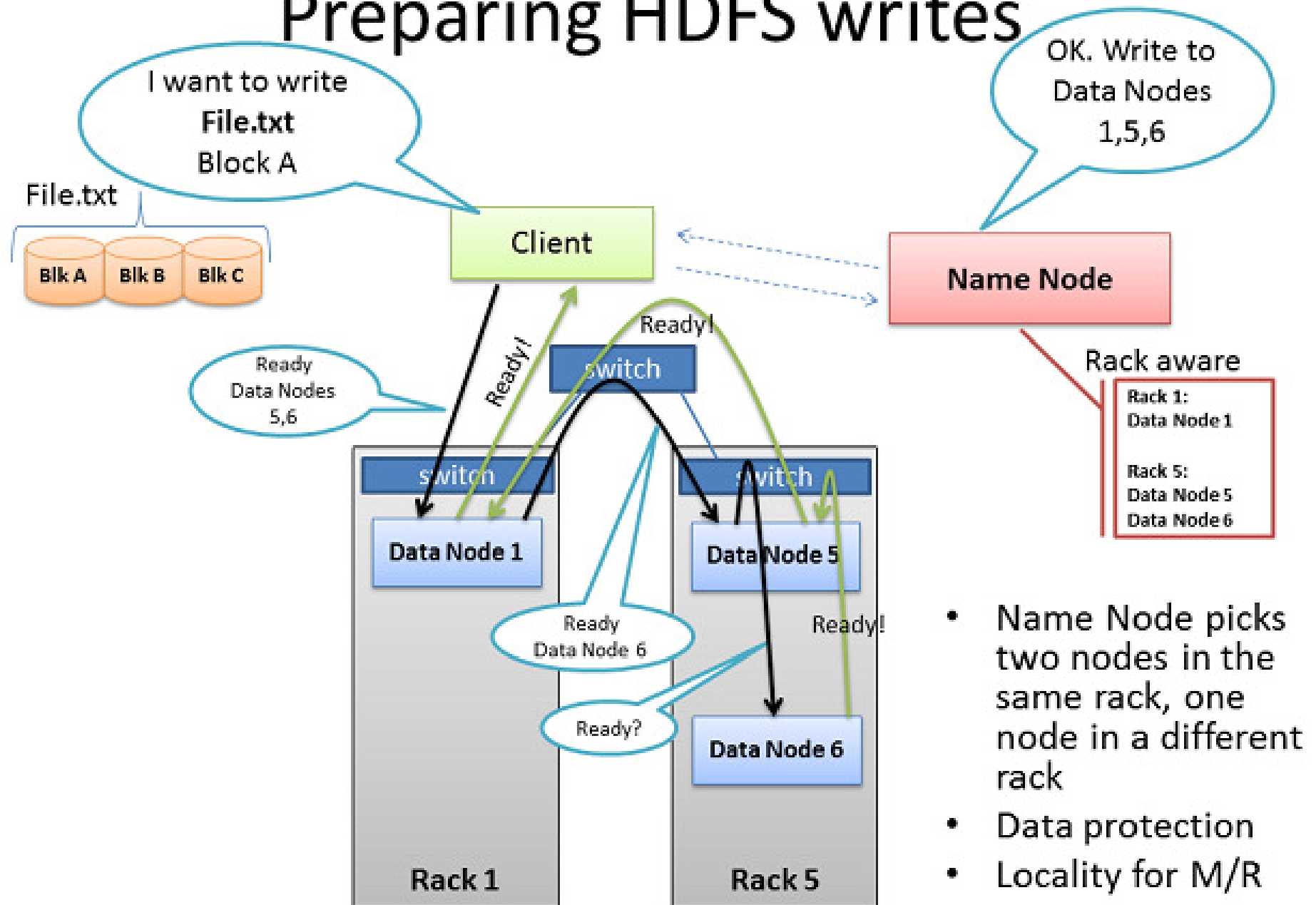
Hadoop has the concept of “Rack Awareness”

- As the Hadoop administrator you can manually define the rack number of each slave Data Node in your cluster.
- Why would you go through the trouble of doing this?
- There are two key reasons for this: Data loss prevention, and network performance.
- Remember that each block of data will be replicated to multiple machines to prevent the failure of one machine from losing all copies of data.
- Wouldn't it be unfortunate if all copies of data happened to be located on machines in the same rack, and that rack experiences a failure?
- Such as a switch failure or power failure. That would be a mess. So to avoid this, somebody needs to know where Data Nodes are located in the network topology and use that information to make an intelligent decision about where data replicas should exist in the cluster.
- That “somebody” is the Name Node.

Bandwidth and latency across racks

- There is also an assumption that two machines in the same rack have more bandwidth and lower latency between each other than two machines in two different racks.
- This is true most of the time.
- The rack switch uplink bandwidth is usually (but not always) less than its downlink bandwidth.
- Furthermore, in-rack latency is usually lower than cross-rack latency (but not always).
- If at least one of those two basic assumptions are true, wouldn't it be cool if Hadoop can use the same Rack Awareness that protects data to also optimally place work streams in the cluster, improving network performance? Well, it does!
- What is NOT cool about Rack Awareness at this point is the manual work required to define it the first time, continually update it, and keep the information accurate.
- If the rack switch could auto-magically provide the Name Node with the list of Data Nodes it has, that would be cool.
- Or vice versa, if the Data Nodes could auto-magically tell the Name Node what switch they're connected to, that would be cool too.

Preparing HDFS writes



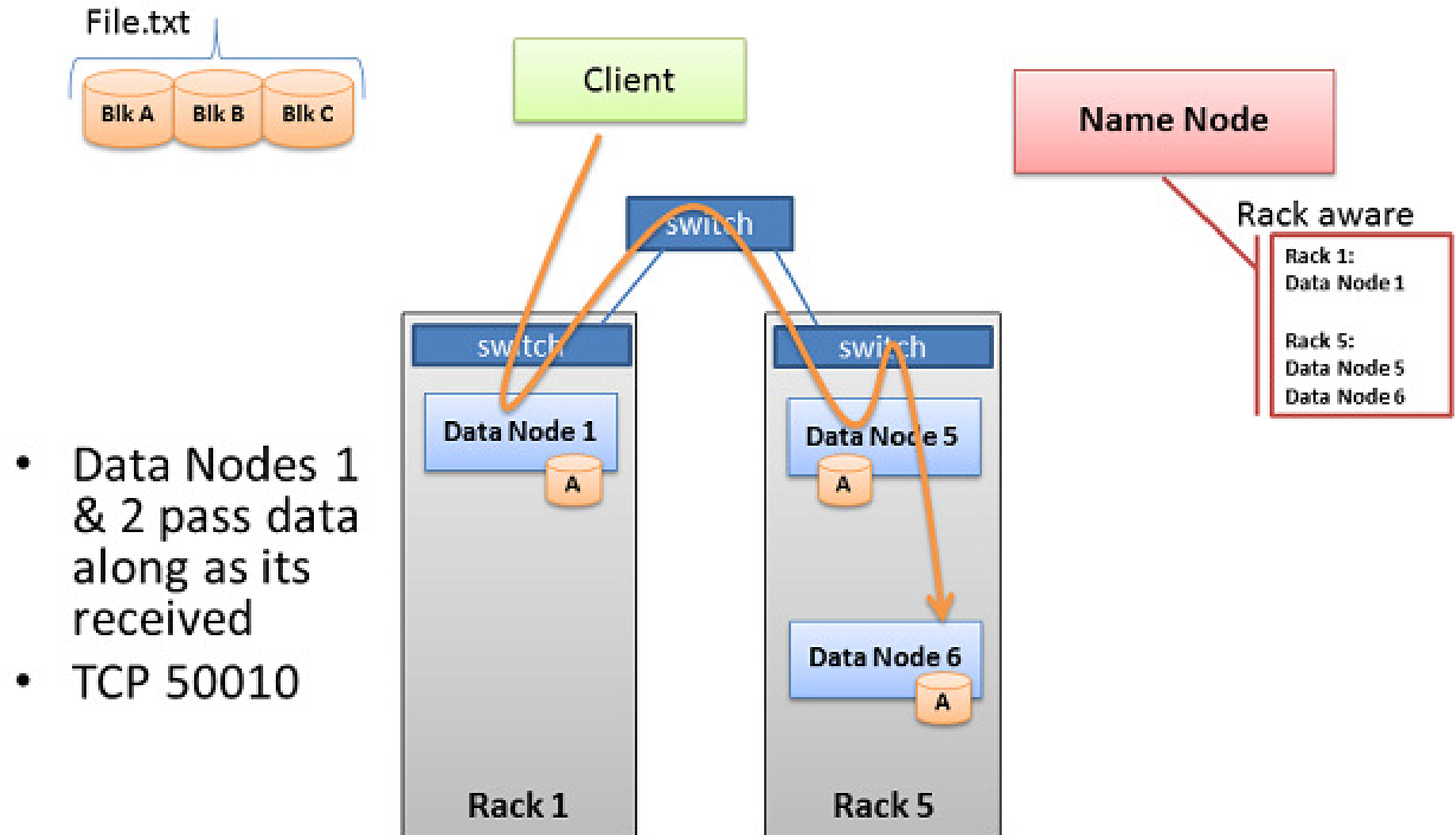
HDFS Writes

- The Client is ready to load File.txt into the cluster and breaks it up into blocks, starting with Block A.
- The Client consults the Name Node that it wants to write File.txt, gets permission from the Name Node, and receives a list of (3) Data Nodes for each block, a unique list for each block.
- The Name Node used its Rack Awareness data to influence the decision of which Data Nodes to provide in these lists.
- The key rule is that for every block of data, two copies will exist in one rack, another copy in a different rack.
- So the list provided to the Client will follow this rule.

Write acknowledgement

- Before the Client writes “Block A” of File.txt to the cluster it wants to know that all Data Nodes which are expected to have a copy of this block are ready to receive it.
- It picks the first Data Node in the list for Block A (Data Node 1), opens a TCP 50010 connection and says, “Hey, get ready to receive a block, and here’s a list of (2) Data Nodes, Data Node 5 and Data Node 6.
- Go make sure they’re ready to receive this block too.” Data Node 1 then opens a TCP connection to Data Node 5 and says, “Hey, get ready to receive a block, and go make sure Data Node 6 is ready is receive this block too.” Data Node 5 will then ask Data Node 6, “Hey, are you ready to receive a block?”
- The acknowledgments of readiness come back on the same TCP pipeline, until the initial Data Node 1 sends a “Ready” message back to the Client.
- At this point the Client is ready to begin writing block data into the cluster.

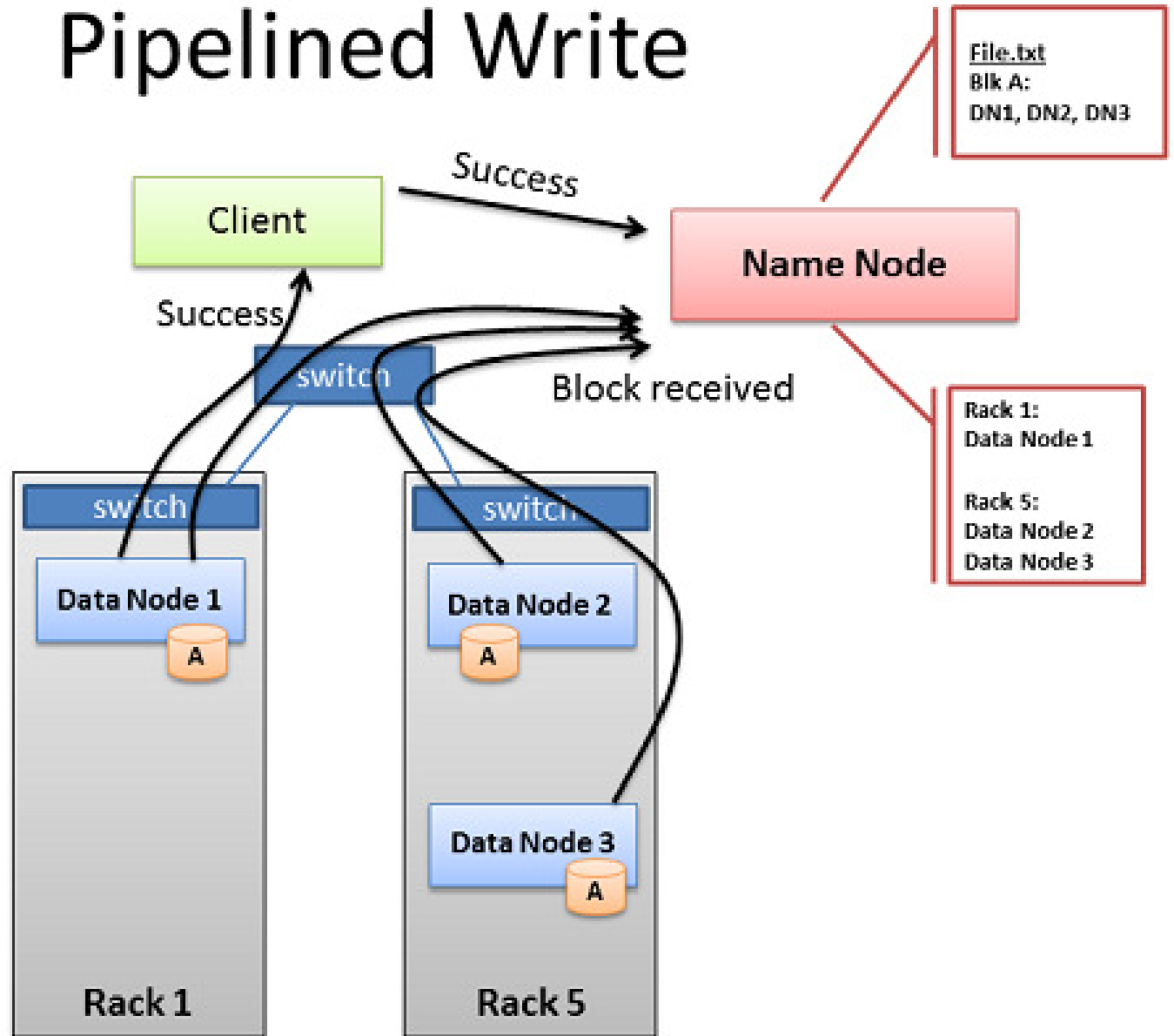
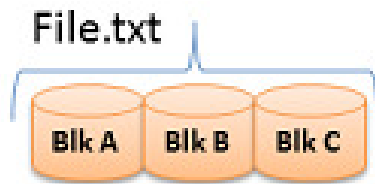
Pipelined Write



Pipeline Write

- As data for each block is written into the cluster a replication pipeline is created between the (3) Data Nodes (or however many you have configured in `dfs.replication`).
- This means that as a Data Node is receiving block data it will at the same time push a copy of that data to the next Node in the pipeline.
- Here too is a primary example of leveraging the Rack Awareness data in the Name Node to improve cluster performance.
- Notice that the second and third Data Nodes in the pipeline are in the same rack, and therefore the final leg of the pipeline does not need to traverse between racks and instead benefits from in-rack bandwidth and low latency.
- The next block will not be begin until this block is successfully written to all 3 nodes.

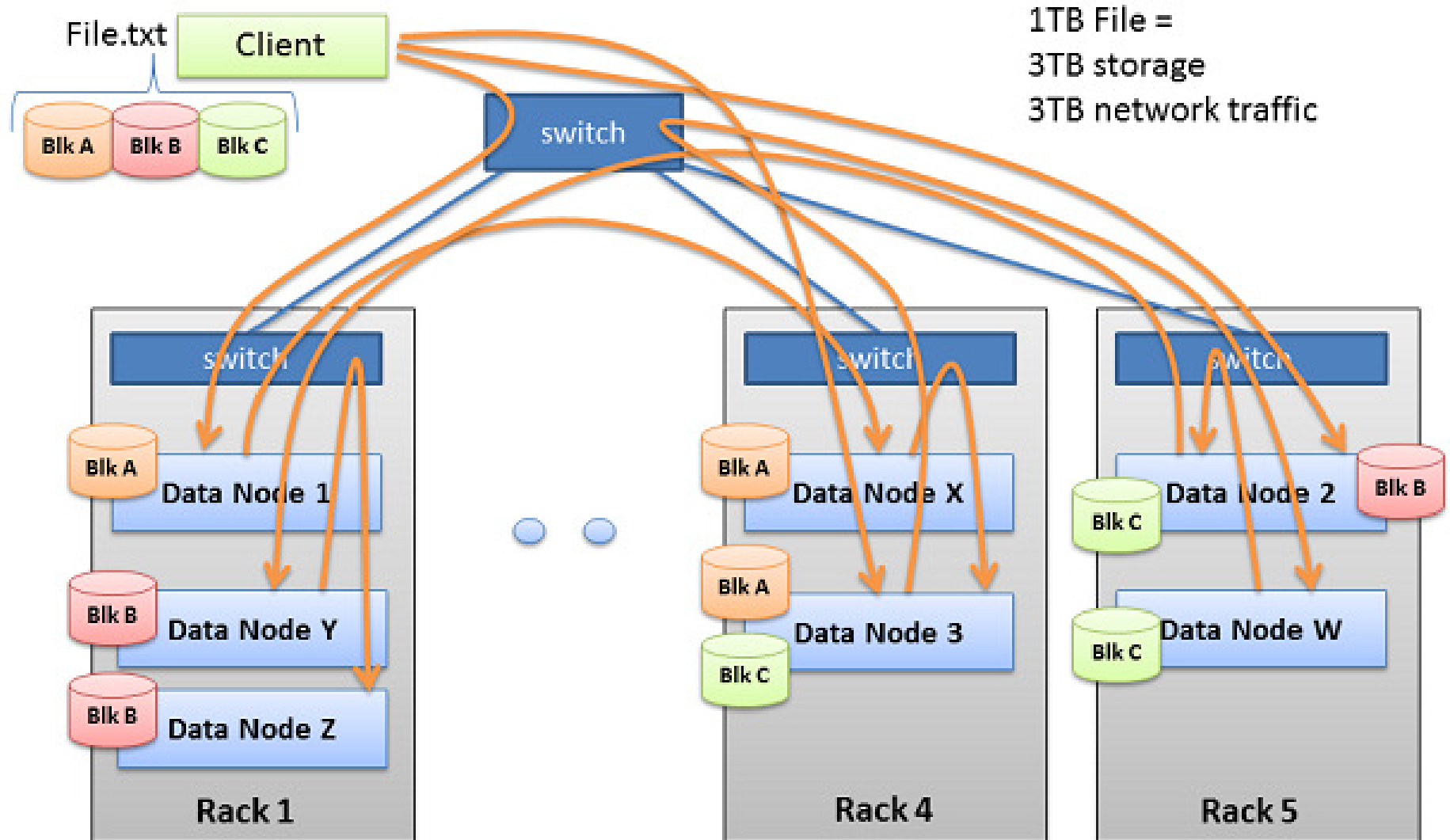
Pipelined Write



Success Message

- When all three Nodes have successfully received the block they will send a “Block Received” report to the Name Node.
- They will also send “Success” messages back up the pipeline and close down the TCP sessions.
- The Client receives a success message and tells the Name Node the block was successfully written.
- The Name Node updates its metadata info with the Node locations of Block A in File.txt.
- The Client is ready to start the pipeline process again for the next block of data.

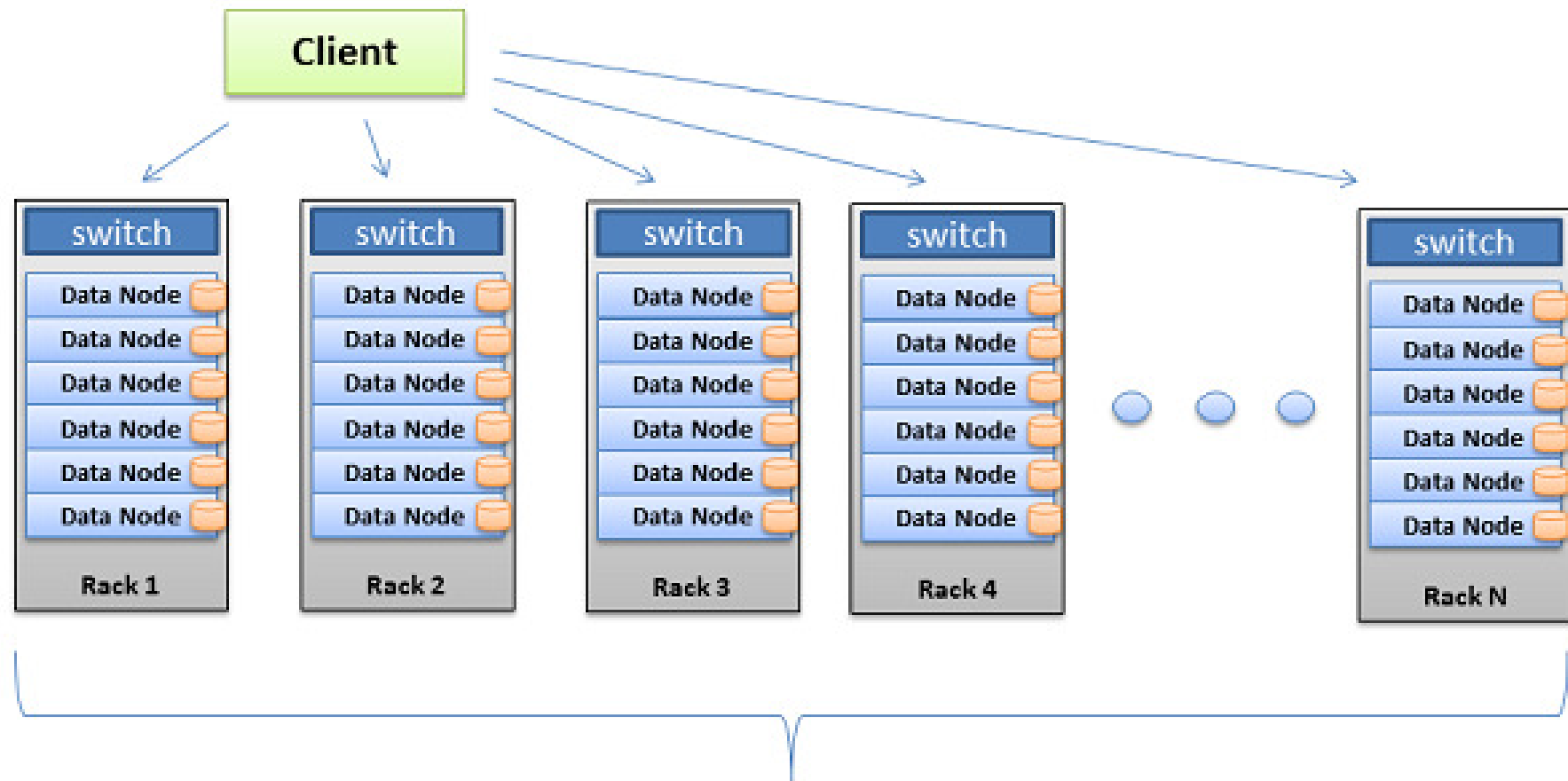
Multi-block Replication Pipeline



Hadoop uses a lot of network bandwidth and storage

- As the subsequent blocks of File.txt are written, the initial node in the pipeline will vary for each block, spreading around the hot spots of in-rack and cross-rack traffic for replication.
- Hadoop uses a lot of network bandwidth and storage.
- We are typically dealing with very big files, Terabytes in size.
- And each file will be replicated onto the network and disk (3) times.
- If you have a 1TB file it will consume 3TB of network traffic to successfully load the file, and 3TB disk space to hold the file.

Client writes Span the HDFS Cluster



Factors:

- Block size
- File Size

File.txt

More blocks = Wider spread

After the replication pipeline of each block is complete the file is successfully written to the cluster.

As intended the file is spread in blocks across the cluster of machines, each machine having a relatively small part of the data.

The more blocks that make up a file, the more machines the data can potentially spread.

The more CPU cores and disk drives that have a piece of my data mean more parallel processing power and faster results.

This is the motivation behind building large, wide clusters.

To process more data, faster.

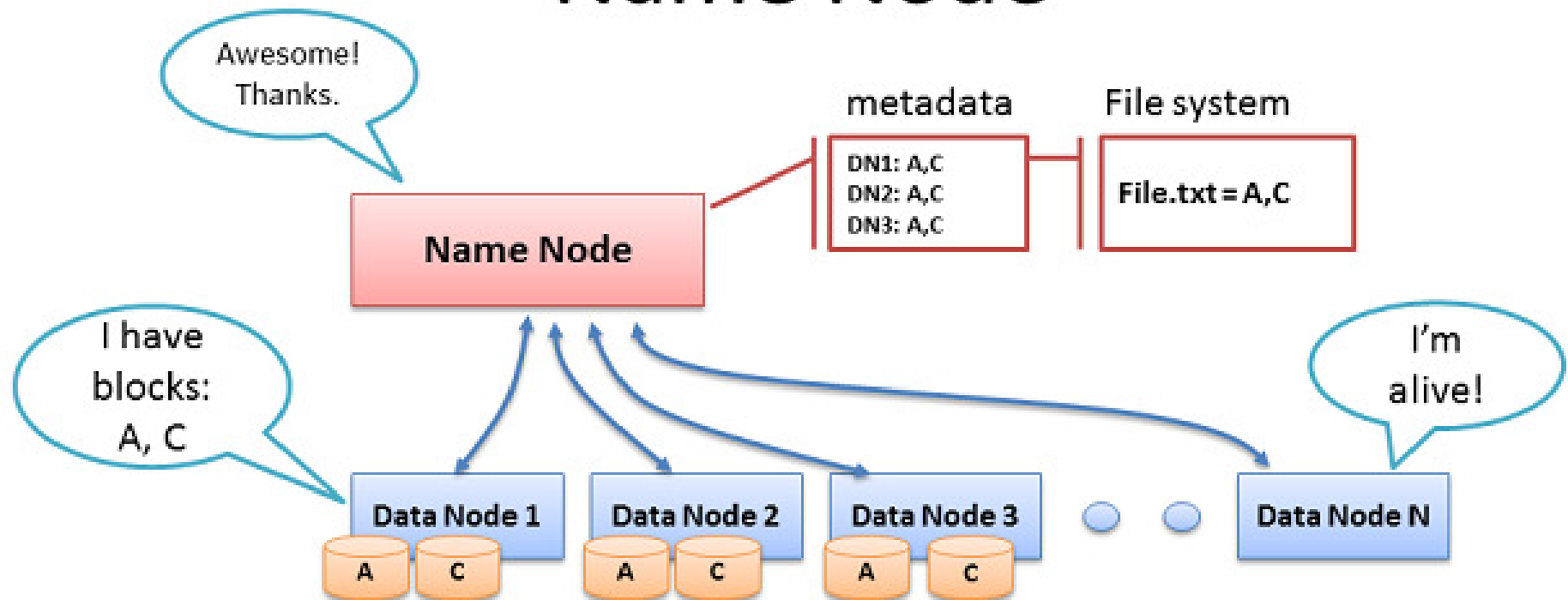
When the machine count goes up and the cluster goes wide, our network needs to scale appropriately.

Scaling Deep



- Another approach to scaling the cluster is to go deep.
- This is where you scale up the machines with more disk drives and more CPU cores.
- Instead of increasing the number of machines you begin to look at increasing the density of each machine.
- In scaling deep, you put yourself on a trajectory where more network I/O requirements may be demanded of fewer machines.
- In this model, how your Hadoop cluster makes the transition to 10GigE nodes becomes an important consideration.

Name Node



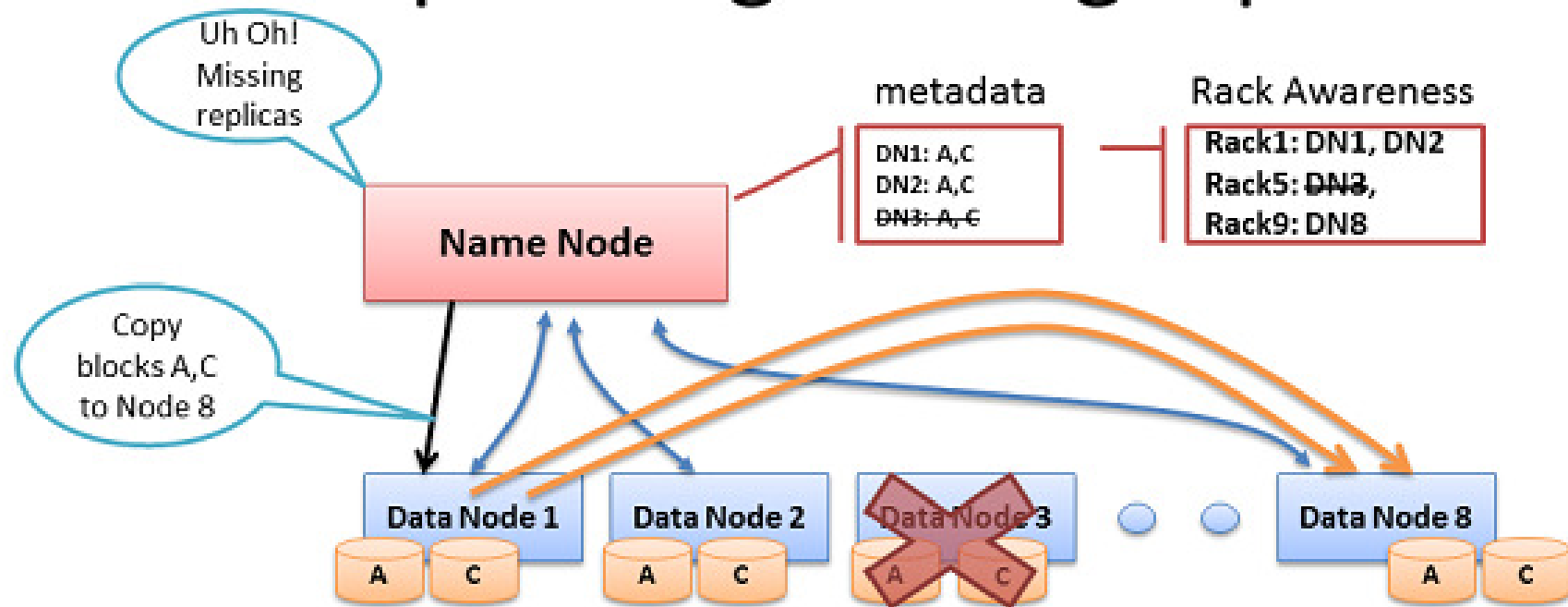
- Data Node sends Heartbeats
- Every 10th heartbeat is a Block report
- Name Node builds metadata from Block reports
- TCP – every 3 seconds
- If Name Node is down, HDFS is down

The Name Node holds all the file system metadata for the cluster and oversees the health of Data Nodes and coordinates access to data. The Name Node is the central controller of HDFS. It does not hold any cluster data itself. The Name Node only knows what blocks make up a file and where those blocks are located in the cluster. The Name Node points Clients to the Data Nodes they need to talk to and keeps track of the cluster's storage capacity, the health of each Data Node, and making sure each block of data is meeting the minimum defined replica policy.

Data Nodes send heartbeats to the Name Node every 3 seconds via a TCP handshake, using the same port number defined for the Name Node daemon, usually TCP 9000. Every tenth heartbeat is a Block Report, where the Data Node tells the Name Node about all the blocks it has. The block reports allow the Name Node build its metadata and insure (3) copies of the block exist on different nodes, in different racks.

The Name Node is a critical component of the Hadoop Distributed File System (HDFS). Without it, Clients would not be able to write or read files from HDFS, and it would be impossible to schedule and execute Map Reduce jobs. Because of this, it's a good idea to equip the Name Node with a highly redundant enterprise class server configuration; dual power supplies, hot swappable fans, redundant NIC connections, etc.

Re-replicating missing replicas

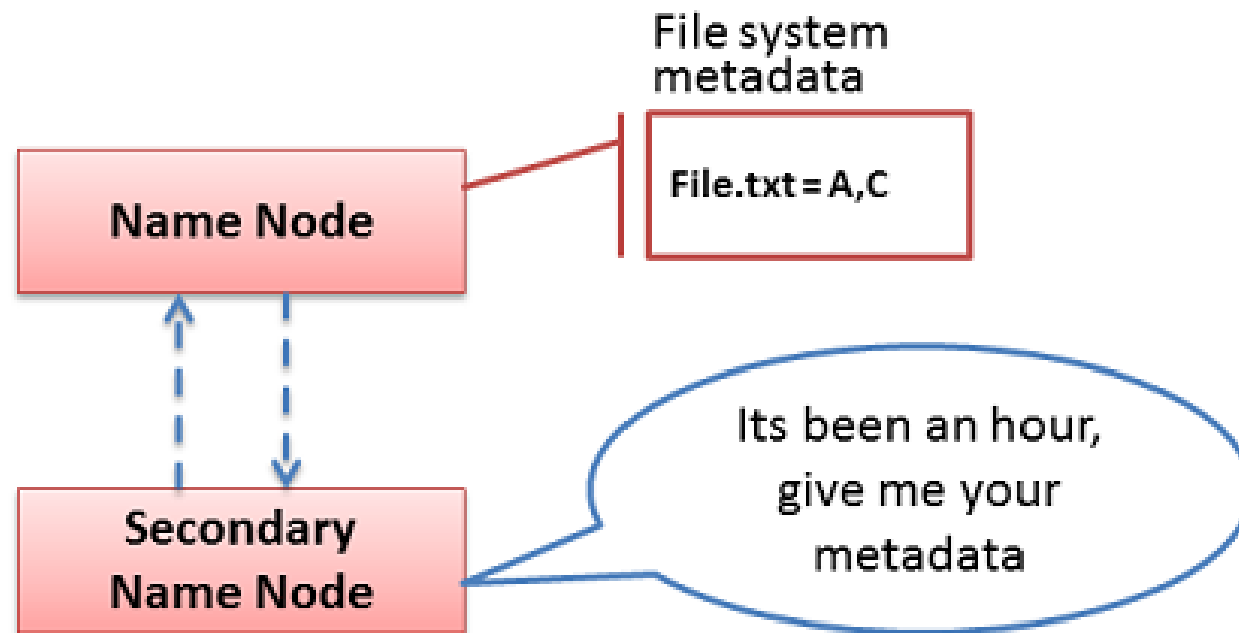


- Missing Heartbeats signify lost Nodes
- Name Node consults metadata, finds affected data
- Name Node consults Rack Awareness script
- Name Node tells a Data Node to re-replicate

If the Name Node stops receiving heartbeats from a Data Node it presumes it to be dead and any data it had to be gone as well. Based on the block reports it had been receiving from the dead node, the Name Node knows which copies of blocks died along with the node and can make the decision to re-replicate those blocks to other Data Nodes. It will also consult the Rack Awareness data in order to maintain the **two copies in one rack, one copy in another rack** replica rule when deciding which Data Node should receive a new copy of the blocks.

Consider the scenario where an entire rack of servers falls off the network, perhaps because of a rack switch failure, or power failure. The Name Node would begin instructing the remaining nodes in the cluster to re-replicate all of the data blocks lost in that rack. If each server in that rack had a modest 12TB of data, this could be hundreds of terabytes of data that needs to begin traversing the network.

Secondary Name Node



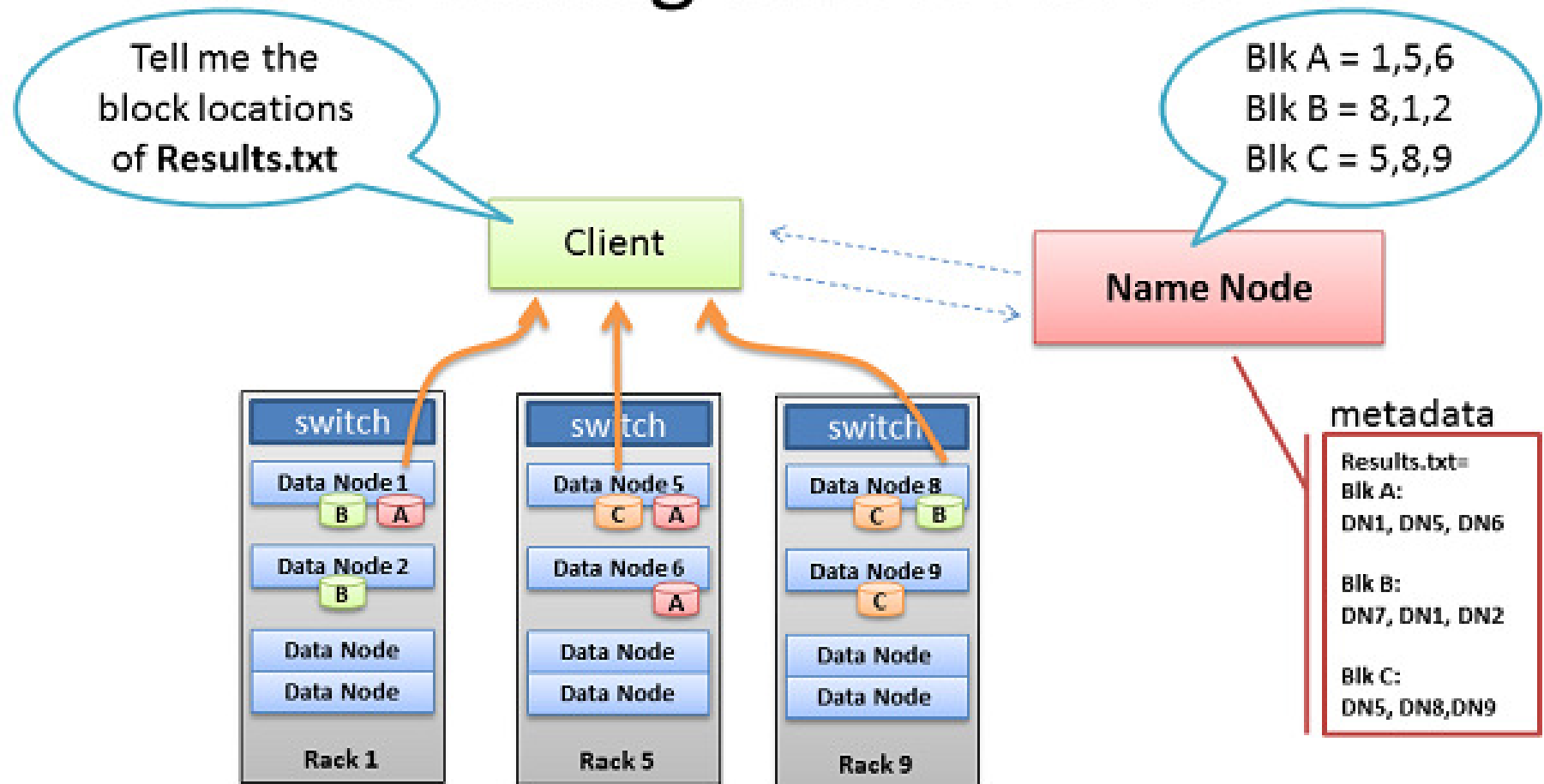
- Not a hot standby for the Name Node
- Connects to Name Node every hour*
- Housekeeping, backup of Name Node metadata
- Saved metadata can rebuild a failed Name Node

Hadoop has server role called the Secondary Name Node. A common misconception is that this role provides a high availability backup for the Name Node. This is not the case.

The Secondary Name Node occasionally connects to the Name Node (by default, every hour) and grabs a copy of the Name Node's in-memory metadata and files used to store metadata (both of which may be out of sync). The Secondary Name Node combines this information in a fresh set of files and delivers them back to the Name Node, while keeping a copy for itself.

Should the Name Node die, the files retained by the Secondary Name Node can be used to recover the Name Node. In a busy cluster, the administrator may configure the Secondary Name Node to provide this housekeeping service much more frequently than the default setting of one hour. Maybe every minute.

Client reading files from HDFS

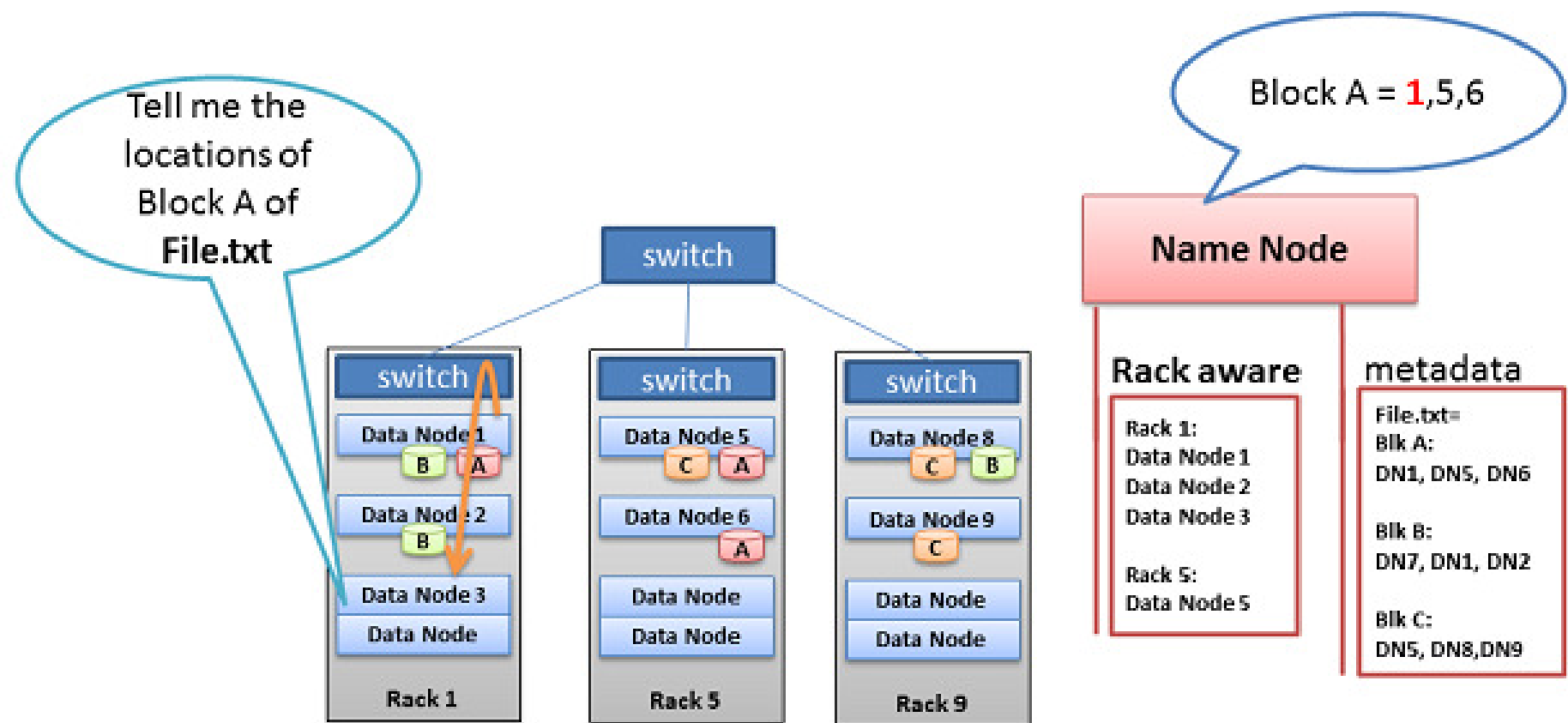


- Client receives Data Node list for each block
- Client picks first Data Node for each block
- Client reads blocks sequentially

When a Client wants to retrieve a file from HDFS, perhaps the output of a job, it again consults the Name Node and asks for the block locations of the file. The Name Node returns a list of each Data Node holding a block, for each block.

The Client picks a Data Node from each block list and reads one block at a time with TCP on port 50010, the default port number for the Data Node daemon. It does not progress to the next block until the previous block completes.

Data Node reading files from HDFS



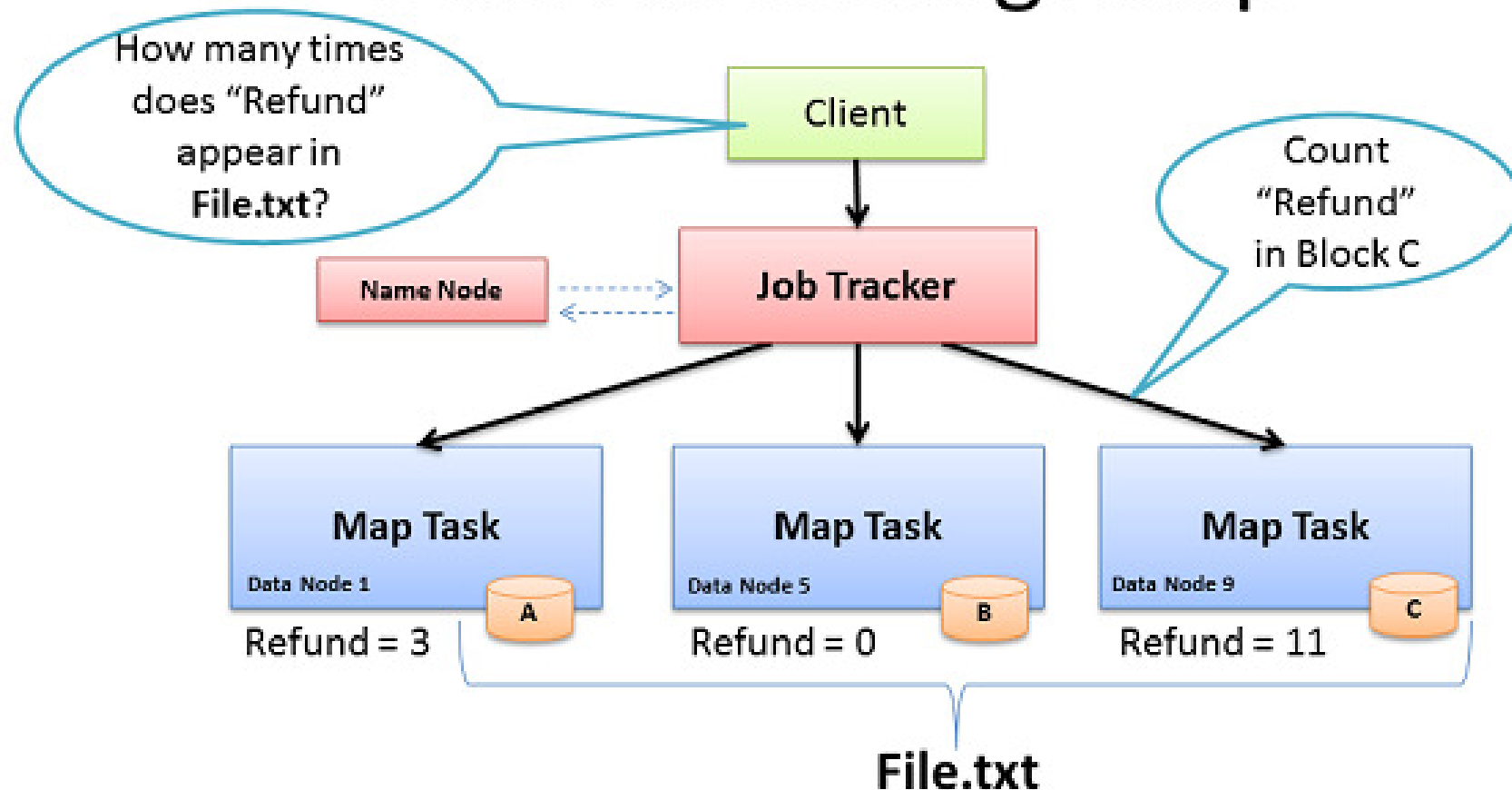
- Name Node provides rack local Nodes first
- Leverage in-rack bandwidth, single hop

There are some cases in which a Data Node daemon itself will need to read a block of data from HDFS. One such case is where the Data Node has been asked to process data that it does not have locally, and therefore it must retrieve the data from another Data Node over the network before it can begin processing.

This is another key example of the Name Node's Rack Awareness knowledge providing optimal network behavior. When the Data Node asks the Name Node for location of block data, the Name Node will check if another Data Node in the same rack has the data. If so, the Name Node provides the in-rack location from which to retrieve the data. The flow does not need to traverse two more switches and congested links find the data in another rack.

With the data retrieved quicker in-rack, the data processing can begin sooner, and the job completes that much faster.

Data Processing: Map



- **Map:** "Run this computation on your local data"
- Job Tracker delivers Java code to Nodes with local data

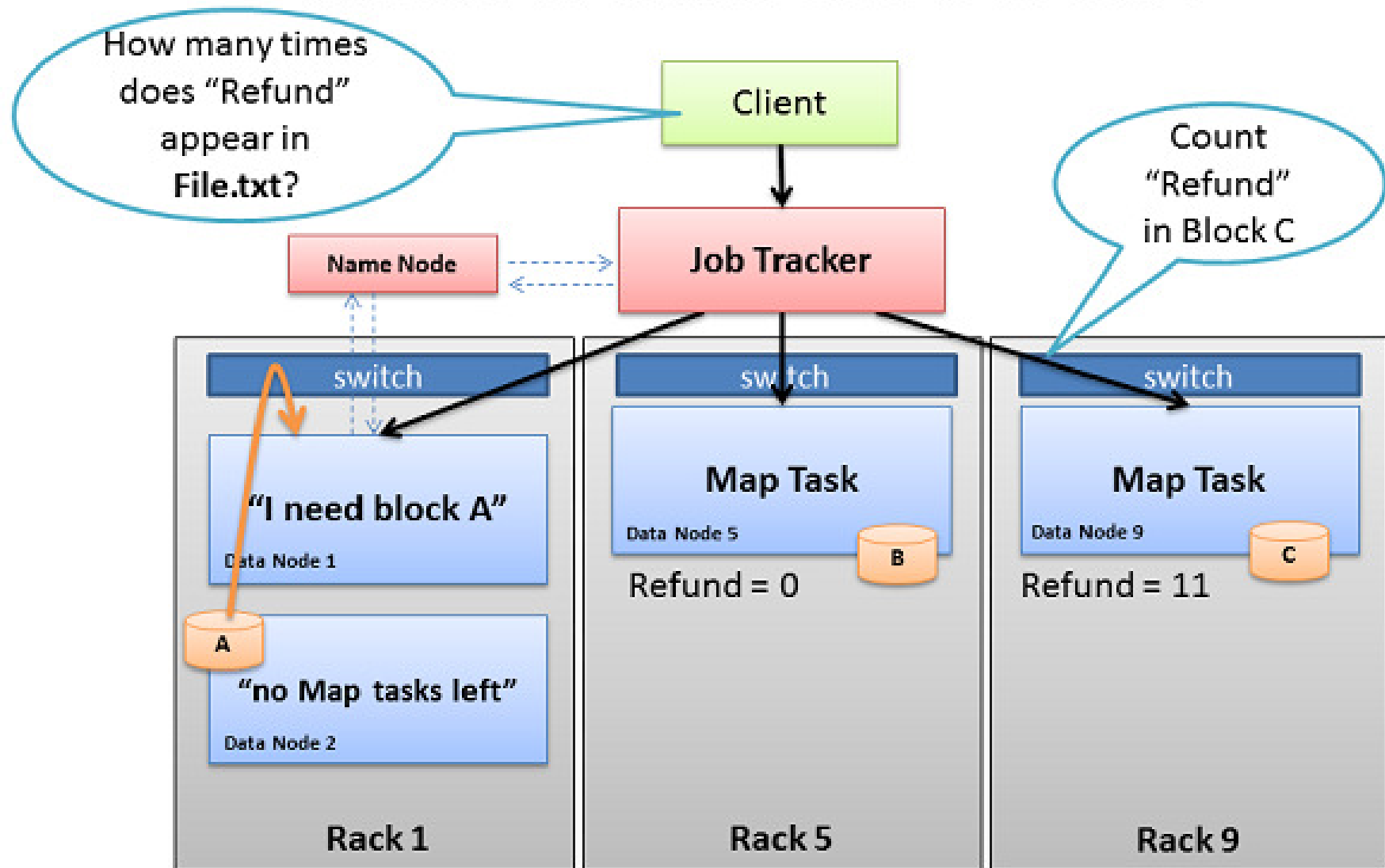
Now that File.txt is spread in small blocks across my cluster of machines I have the opportunity to provide extremely fast and efficient parallel processing of that data. The parallel processing framework included with Hadoop is called Map Reduce, named after two important steps in the model; **Map**, and **Reduce**.

The first step is the Map process. This is where we simultaneously ask our machines to run a computation on their local block of data. In this case we are asking our machines to count the number of occurrences of the word "Refund" in the data blocks of File.txt.

To start this process the Client machine submits the Map Reduce job to the Job Tracker, asking "How many times does Refund occur in File.txt" (paraphrasing Java code). The Job Tracker consults the Name Node to learn which Data Nodes have blocks of File.txt. The Job Tracker then provides the Task Tracker running on those nodes with the Java code required to execute the Map computation on their local data. The Task Tracker starts a Map task and monitors the tasks progress. The Task Tracker provides heartbeats and task status back to the Job Tracker.

As each Map task completes, each node stores the result of its local computation in temporary local storage. This is called the "intermediate data". The next step will be to send this intermediate data over the network to a Node running a Reduce task for final computation.

What if data isn't local?

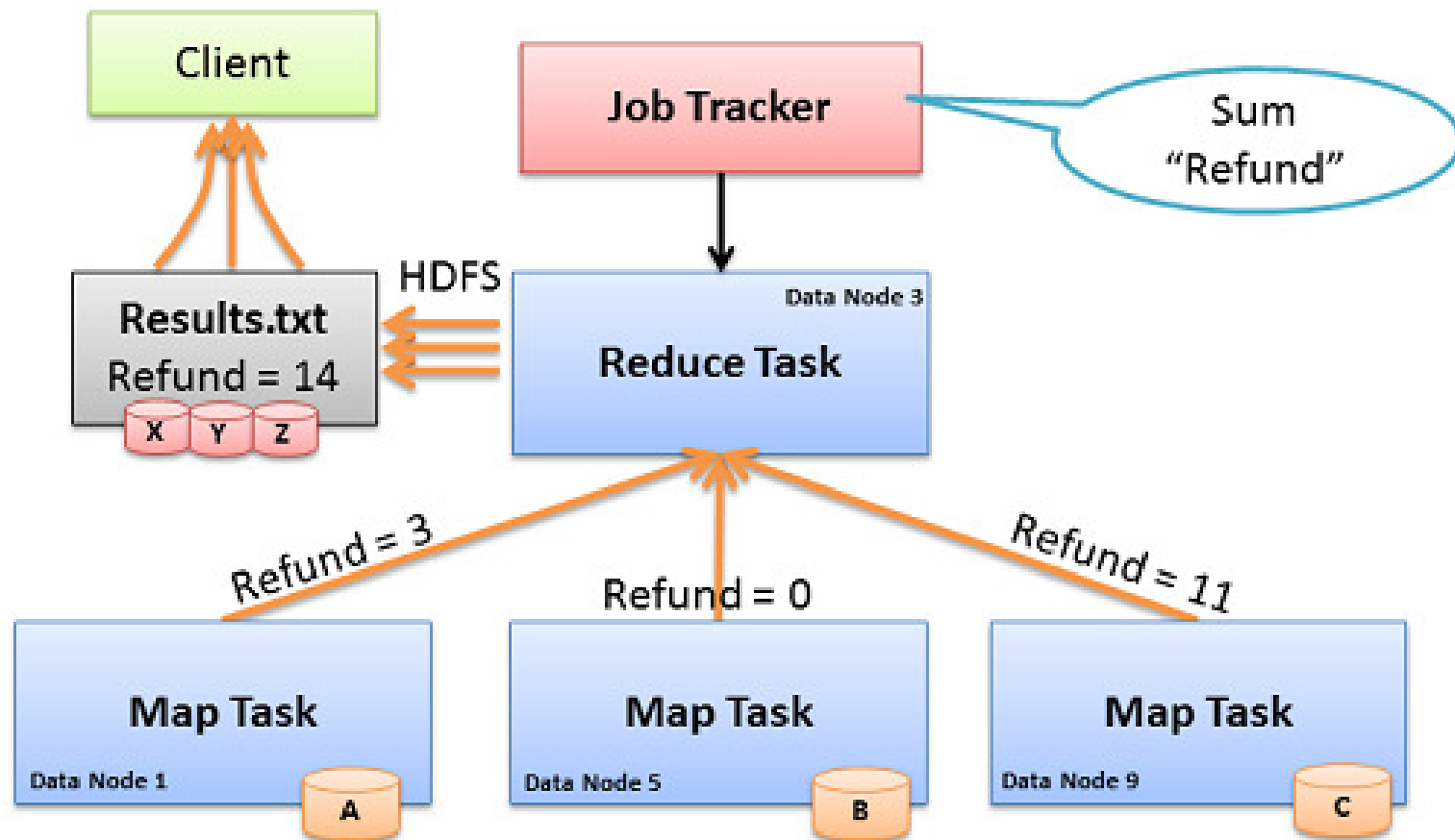


- Job Tracker tries to select Node in same rack as data
- Name Node rack awareness

While the Job Tracker will always try to pick nodes with local data for a Map task, it may not always be able to do so. One reason for this might be that all of the nodes with local data already have too many other tasks running and cannot accept anymore.

In this case, the Job Tracker will consult the Name Node whose Rack Awareness knowledge can suggest other nodes in the same rack. The Job Tracker will assign the task to a node in the same rack, and when that node goes to find the data it needs the Name Node will instruct it to grab the data from another node in its rack, leveraging the presumed single hop and high bandwidth of in-rack switching.

Data Processing: Reduce



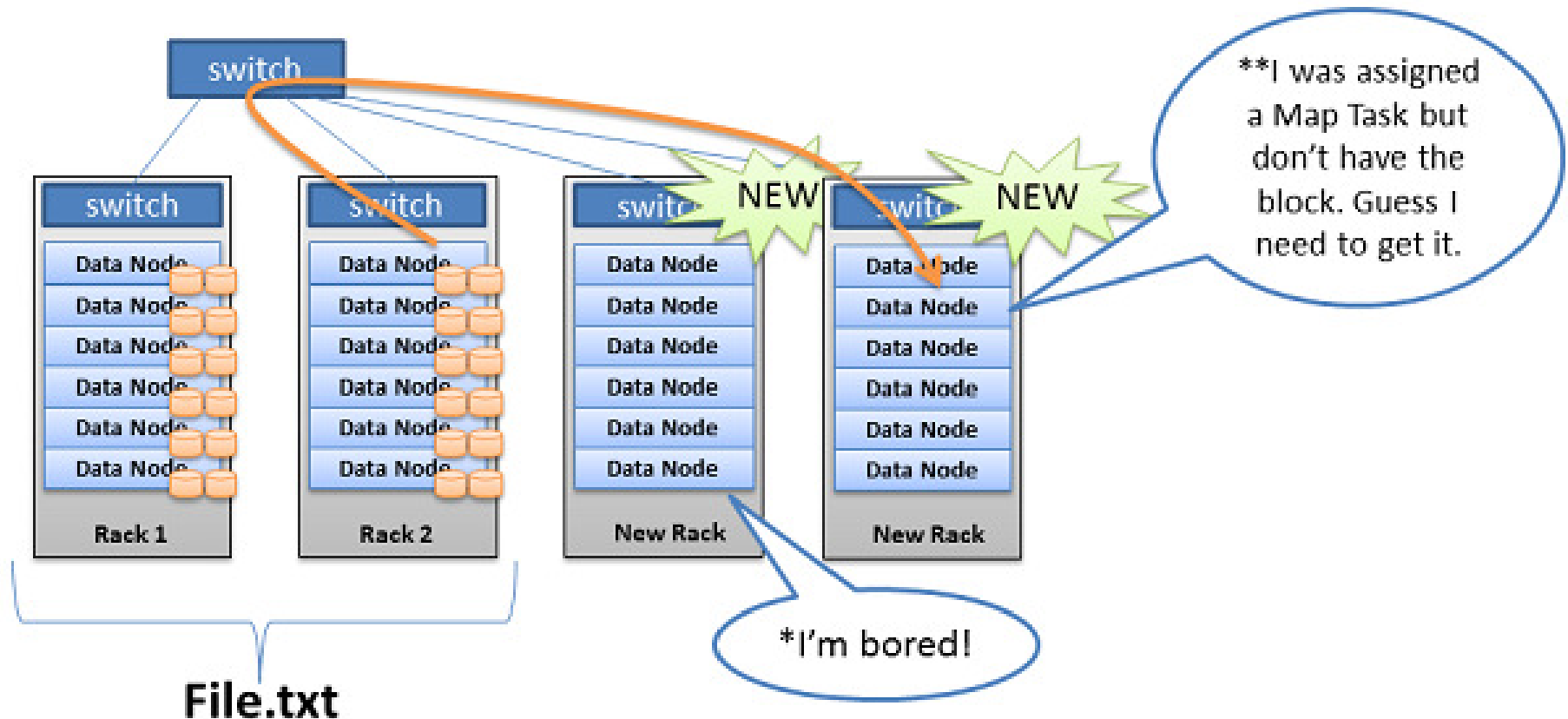
- **Reduce:** “Run this computation across Map results”
- Map Tasks send output data to Reducer over the network
- Reduce Task data output written to and read from HDFS

The second phase of the Map Reduce framework is called, you guess it, **Reduce**. The Map task on the machines have completed and generated their intermediate data. Now we need to gather all of this intermediate data to combine and distill it for further processing such that we have one final result.

The Job Tracker starts a Reduce task on any one of the nodes in the cluster and instructs the Reduce task to go grab the intermediate data from all of the completed Map tasks. The Map tasks may respond to the Reducer almost simultaneously, resulting in a situation where you have a number of nodes sending TCP data to a single node, all at once. This traffic condition is often referred to as "Incast" or "fan-in". For networks handling lots of incast conditions, its important the network switches have well-engineered internal traffic management capabilities, and adequate buffers (not too big, not too small). Throwing gobs of buffers at a switch may end up causing unwanted collateral damage to other traffic. But that's a topic for another day.

The Reducer task has now collected all of the intermediate data from the Map tasks and can begin the final computation phase. In this case, we are simply adding up the sum total occurrences of the word "Refund" and writing the result to a file called Results.txt

Unbalanced Cluster



- Hadoop prefers local processing if possible
- New servers underutilized for Map Reduce, HDFS*
- More network bandwidth, slower job times**

Hadoop may start to be a real success in your organization, providing a lot of previously untapped business value from all that data sitting around. When business folks find out about this you can bet that you'll quickly have more money to buy more racks of servers and network for your Hadoop cluster.

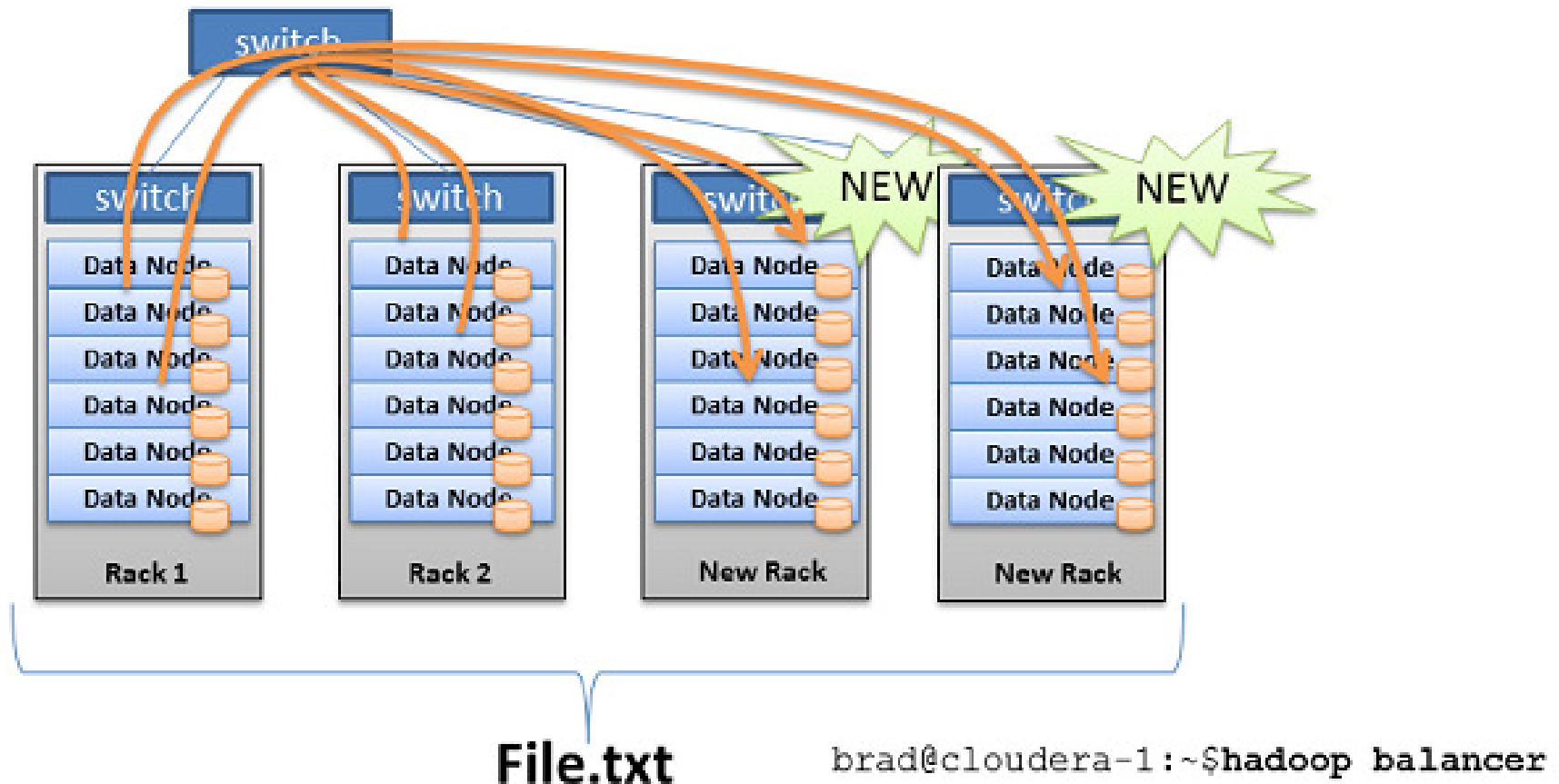
When you add new racks full of servers and network to an existing Hadoop cluster you can end up in a situation where your cluster is unbalanced. In this case, Racks 1 & 2 were my existing racks containing File.txt and running my Map Reduce jobs on that data. When I added two new racks to the cluster, my File.txt data doesn't auto-magically start spreading over to the new racks. All the data stays where it is.

The new servers are sitting idle with no data, until I start loading new data into the cluster.

Furthermore, if the servers in Racks 1 & 2 are really busy, the Job Tracker may have no other choice but to assign Map tasks on File.txt to the new servers which have no local data.

The new servers need to go grab the data over the network. As a result you may see more network traffic and slower job completion times.

Cluster Balancing



- Balancer utility (if used) runs in the background
- Does not interfere with Map Reduce or HDFS
- Default rate limit 1 MB/s

To fix the unbalanced cluster situation, Hadoop includes a nifty utility called, you guessed it, **balancer**.

Balancer looks at the difference in available storage between nodes and attempts to provide balance to a certain threshold. New nodes with lots of free disk space will be detected and balancer can begin copying block data off nodes with less available space to the new nodes.

Balancer isn't running until someone types the command at a terminal, and it stops when the terminal is canceled or closed.

The amount of network traffic balancer can use is very low, with a default setting of 1MB/s.

This setting can be changed with the **dfs.balance.bandwidthPerSec** parameter in the file **hdfs-site.xml**

The Balancer is good housekeeping for your cluster. It should definitely be used any time new machines are added, and perhaps even run once a week for good measure. Given the balancers low default bandwidth setting it can take a long time to finish its work, perhaps days or weeks.

Wouldn't it be cool if cluster balancing was a core part of Hadoop, and not just a utility?