Understanding Hadoop Clusters and the Network

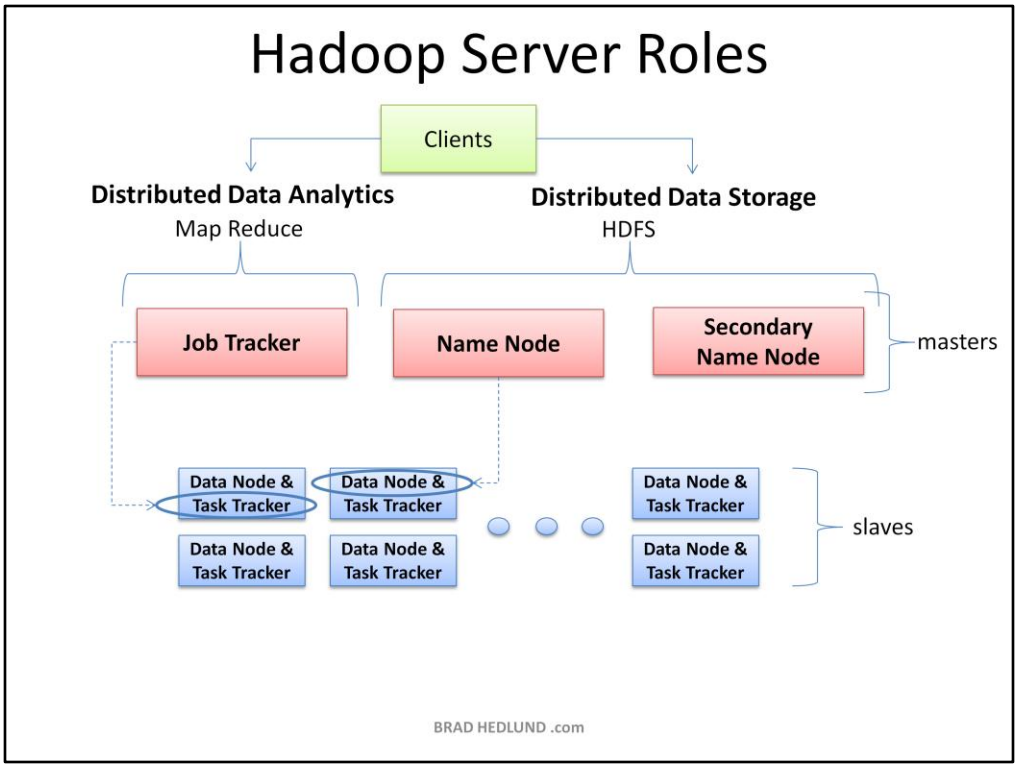
Part 1. Introduction and Overview

Brad Hedlund
http://bradhedlund.com
http://www.linkedin.com/in/bradhedlund
@bradhedlund

BRAD HEDLUND .com

http://bradhedlund.com/?p=3108

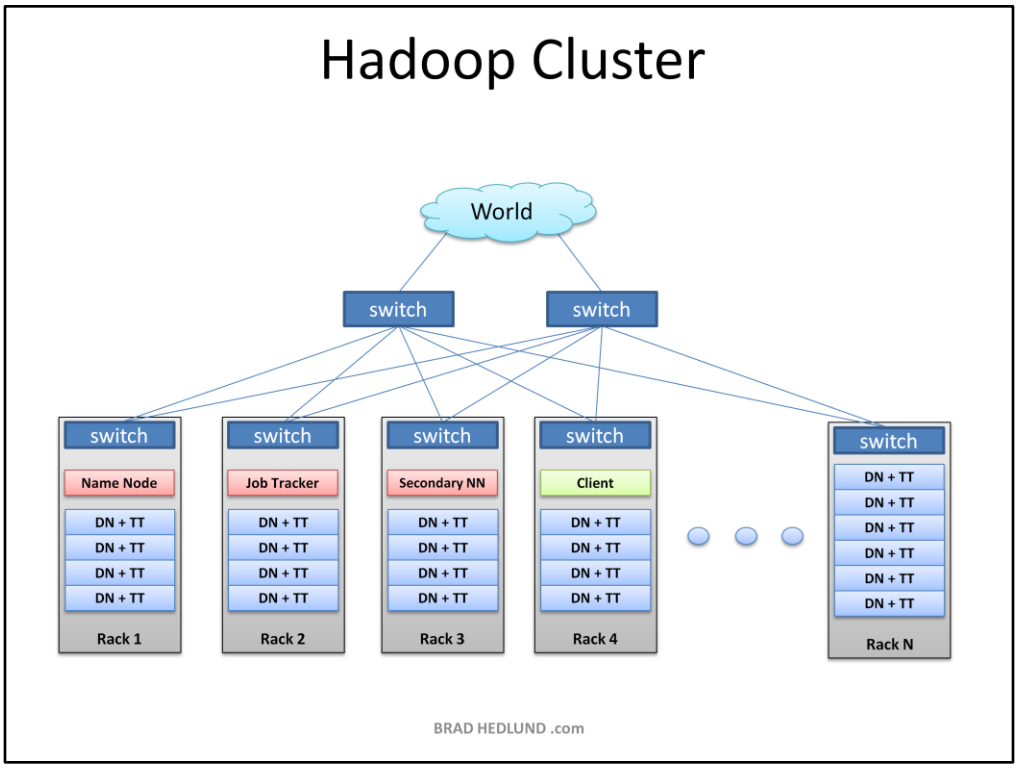
This article is Part 1 in series that will take a closer look at the architecture and methods of a Hadoop cluster, and how it relates to the network and server infrastructure. The content presented here is largely based on academic work and conversations I've had with customers running real production clusters. If you run production Hadoop clusters in your data center, I'm hoping you'll provide your valuable insight in the comments below. Subsequent articles to this will cover the server and network architecture options in closer detail. Before we do that though, lets start by learning some of the basics about how a Hadoop cluster works. OK, lets get started!



The three major categories of machine roles in a Hadoop deployment are Client machines, Masters nodes, and Slave nodes. The Master nodes oversee the two key functional pieces that make up Hadoop: storing lots of data (HDFS), and running parallel computations on all that data (Map Reduce). The Name Node oversees and coordinates the data storage function (HDFS), while the Job Tracker oversees and coordinates the parallel processing of data using Map Reduce. 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 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, 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. I have a 6-node cluster up and running in VMware Workstation on my Windows 7 laptop.



This is the typical architecture of a Hadoop cluster. You will have rack servers (not blades) populated in racks connected to a top of rack switch usually with 1 or 2 GE boned links. 10GE 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.

In this post, we are not going to discuss various detailed network design options. Let's save that for another discussion (stay tuned). First, lets understand how this application works...

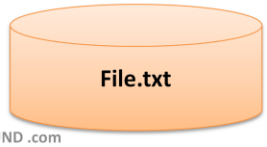
Typical Workflow

- 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 "**Fraud**" into emails sent to customer service?

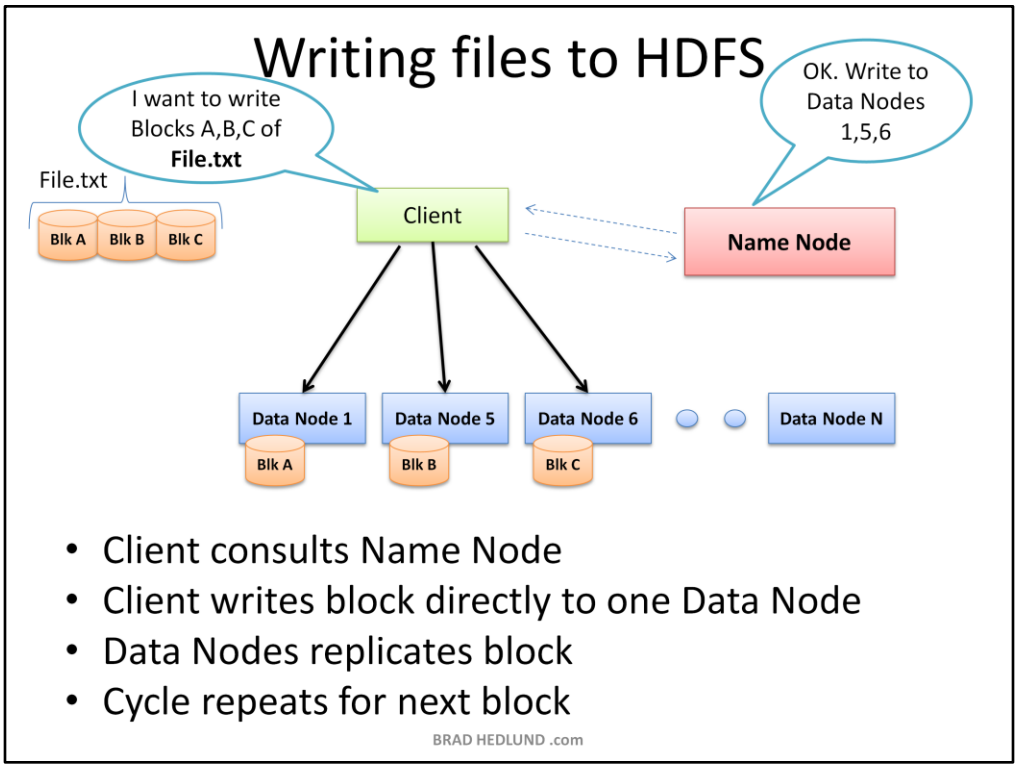
Huge file containing all emails sent to customer service



BRAD HEDLUND .com

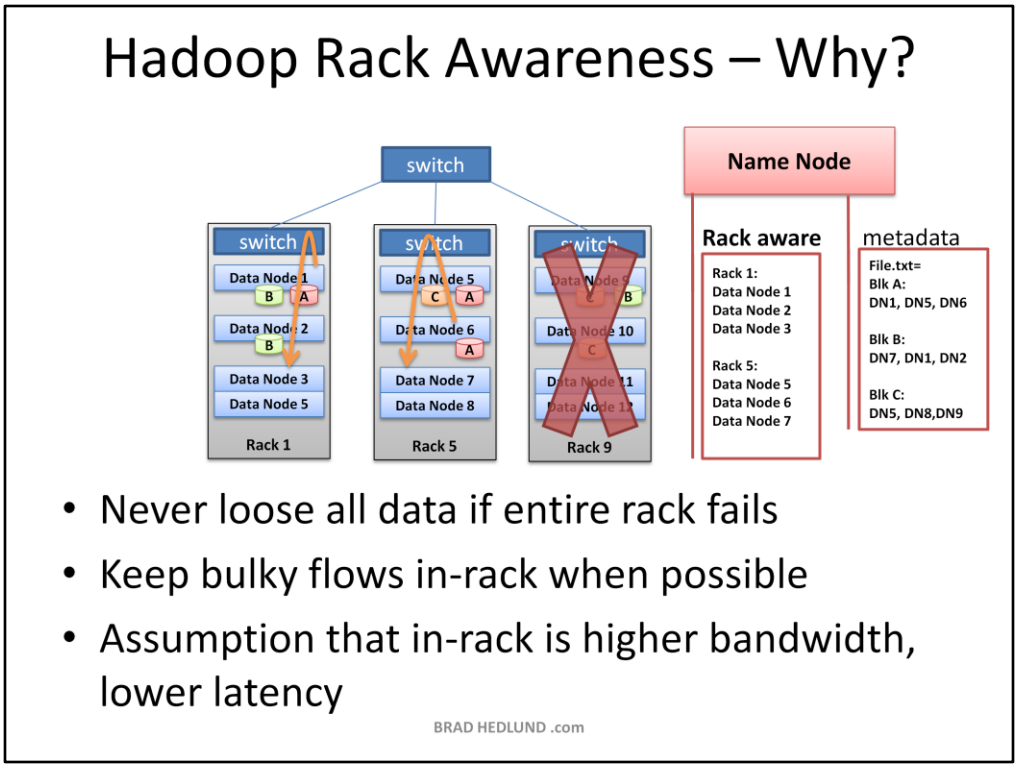
Why did Hadoop come to exist? What problem does it solve? Simply put, businesses and governments have a tremendous amount of data that needs to be analyzed and processed very quickly. If I can chop that huge chunk of data into small chunks and spread it out over many machines, and have all those machines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

In our simple example, we'll have a huge data file containing emails sent to the customer service department. I 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.



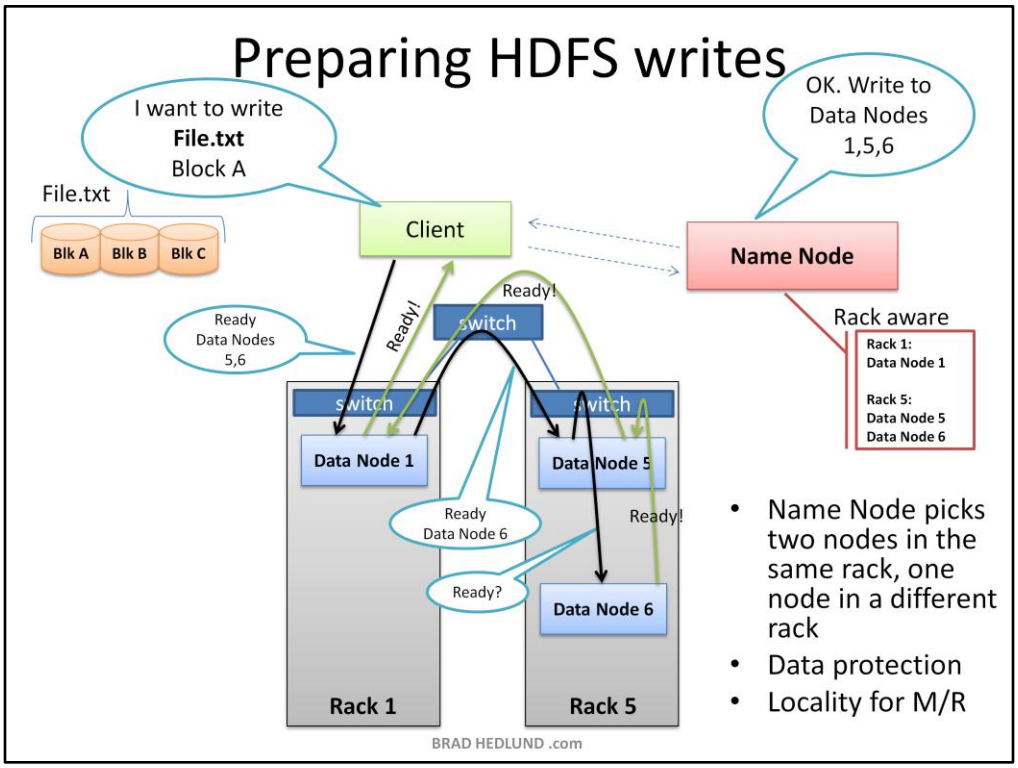
Your Hadoop cluster is useless until it has data, so we'll begin by loading our huge File.txt into the cluster for processing. The goal here is fast parallel processing of lots of data. To accomplish that I need as many machines as possible working on this data all at once. To that end, the Client is going to break the data file into smaller "Blocks", and place those blocks on different machines throughout the cluster. The more blocks I 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. 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**.

The Client breaks File.txt into (3) Blocks. For each block, the Client consults the Name Node (usually TCP 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 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 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.

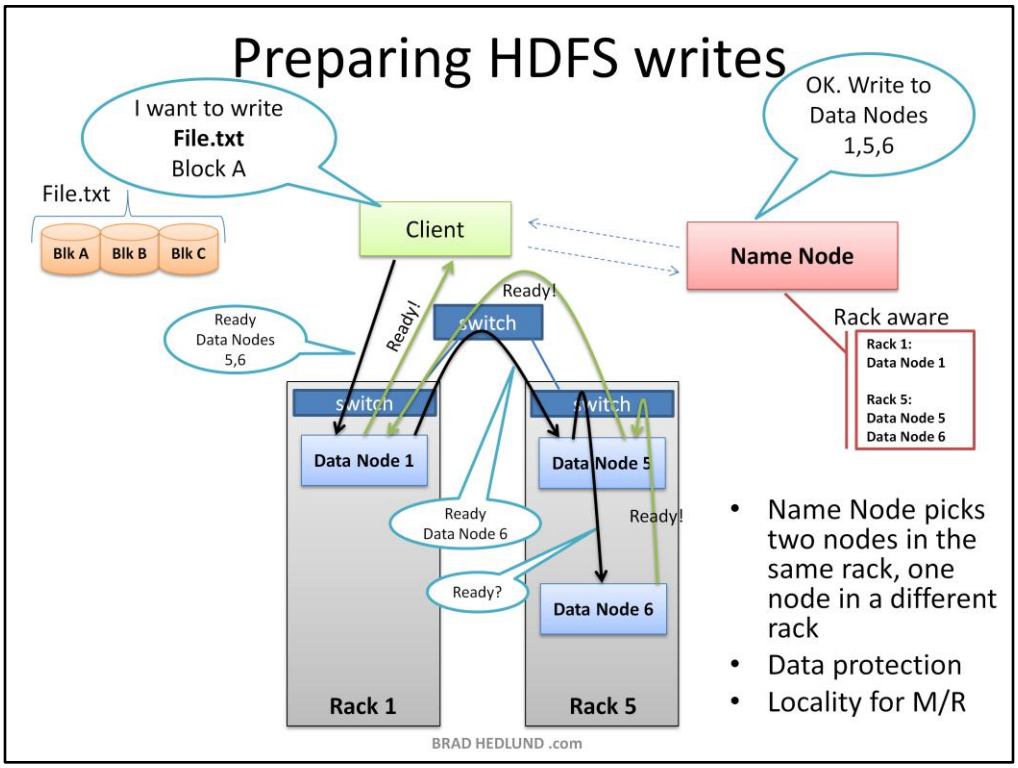
There is also an assumption that two machines in the same rack have more bandwidth and lower latency between each other than two machines in two different racks. This is true most of the time. The rack switch uplink bandwidth is usually (but not always) less than its downlink bandwidth. Furthermore, 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! Cool, right? 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.
Even more interesting would be a OpenFlow network, where the Name Node could query the OpenFlow controller about a Node's location in the topology.



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.

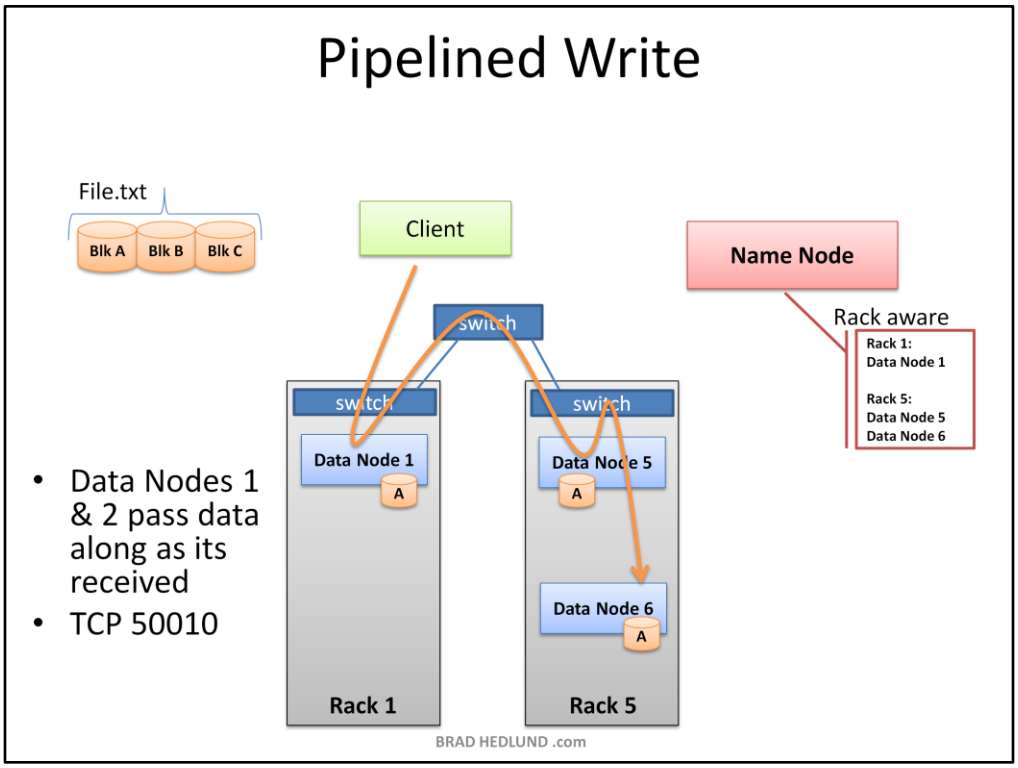
Before the Client writes "Block A" of File.txt to the cluster it wants to know that all Data Nodes which are expected to have a copy of this block are ready to receive it. It picks the first Data Node in the list for Block A (Data Node 1), opens a TCP 50010 connection and says, "Hey, get ready to receive 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.



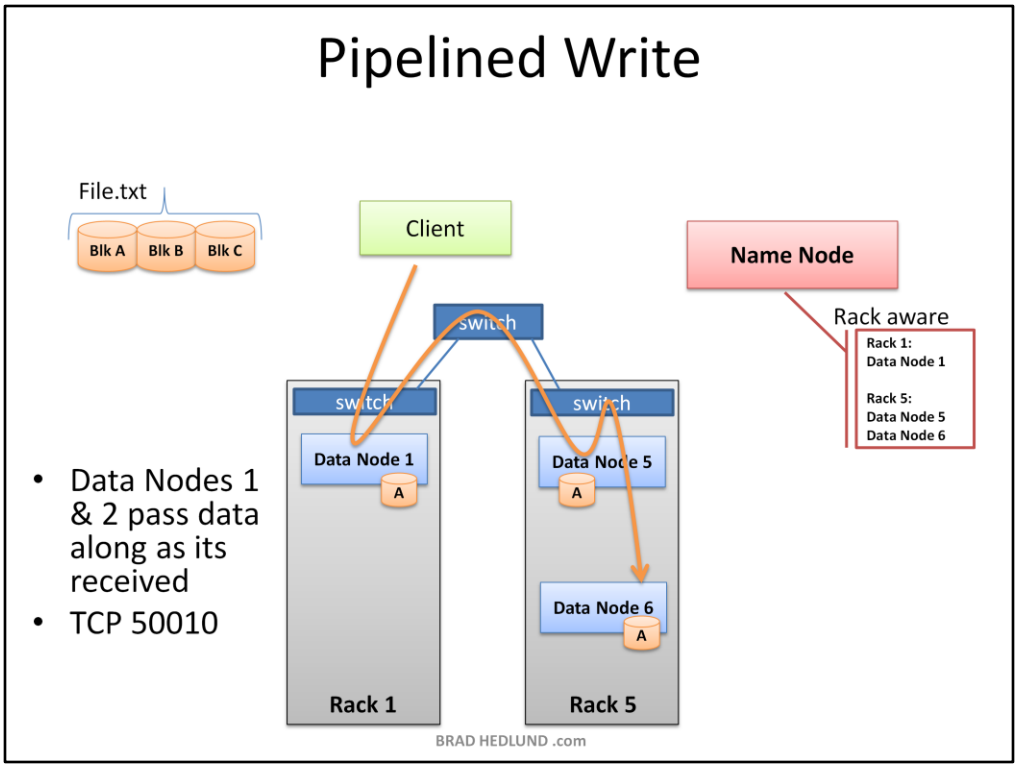
Capture Analysis:

Client tells the Name Node the directory and file (File.txt) it wishes to write to HDFS.
Name Node checks permissions and responds to Client with grant (Writeable).
Client sends Add Block request to Name Node for Block (1) of File.txt.
Name Node responds with a list of (3) Data Nodes, including IP address, port number, hostnames, rack numbers.
Client initiates TCP to Data Node (1) in the list (TCP port 50010).
Client sends Data Node (1) the list of the other (2) Data Nodes, including IP address, port number, hostname, rack #.
Data Node (1) initiates TCP to Data Node (2), handshake (TCP port 50010).
Data Node (1) provides Data Node (2) information about Data Node (3), IP address, port number, hostname, rack # - Data Node (2) ACKs.
Data Node (2) initiates TCP to Data Node (3), handshake (TCP port 50010).
Data Node (2) provides Data Node (3) some information about the Client, Data Node (3) ACKs.
Data Node (3) sends Data Node (2) a 3 Byte string of 0's (000000) – some sort of "pipeline ready" message? – Data Node (2) ACKs.
Data Node (2) sends Data Node (1) same 3 Byte string of 0's (000000) – "pipline ready"? – Data Node (1) ACKs.
Data Node (1) sends Client same 3 Byte string of 0's (000000) – "pipeline ready"? – Client ACKs.



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 three nodes.



Capture Analysis:
Client begins sending block data to Data Node (1).
Client sends Data Node (1) block data.

During that time:

Data Node (1) begins sending block data to Data Node (2).
Data Node (2) begins block data to Data Node (3).

~pipeline continues for remainder of block data~

When completed, each Data Node reports to Name Node "block received" with block info

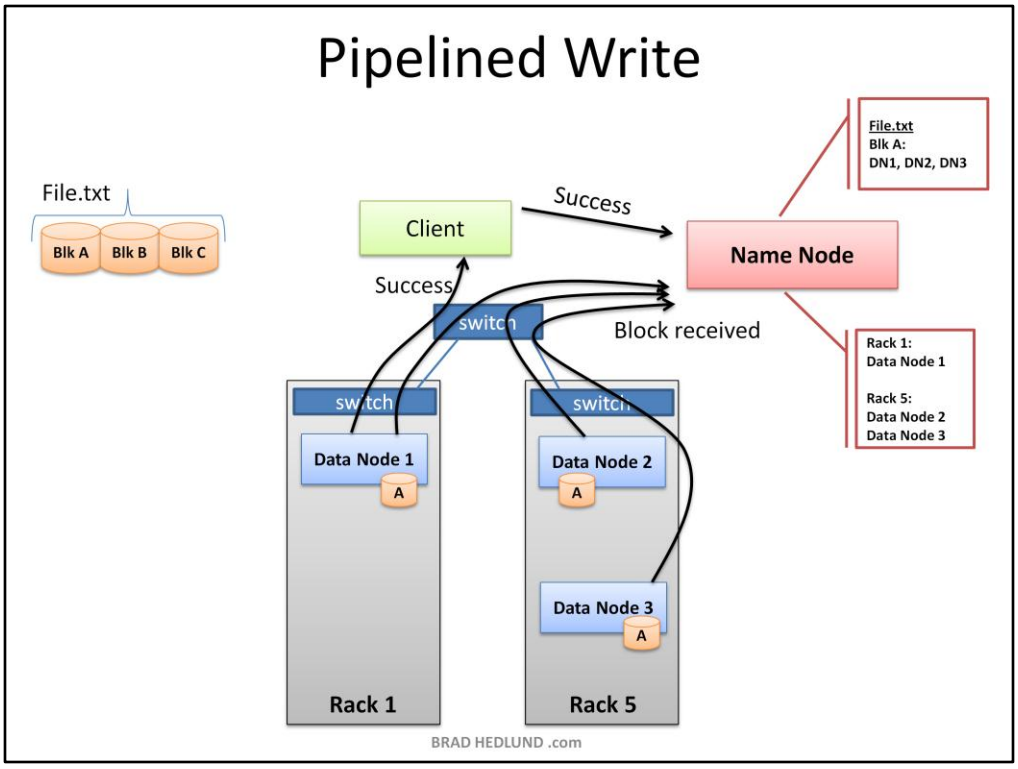
Data Node (1) sends a string of 00:00:00:00:00:00:00:05:00:00:00:00:00:00 to Client – some sort of complete signal?

Client sends a string of 00000000 to Data Node (1) – acknowledgement of the completion?

Data Node (1) closes TCP connection to Client.
Client reports "success" to Name Node.
Name Node knows (3) replicas of the block exist.

Data Nodes proceed to provide usual Block Reports and Heartbeats to Name Node

~Entire process repeats for remaining blocks of File.txt~
Next block is not transferred to HDFS until previous block successfully completes.



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 it 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.

Capture Analysis:
When completed, each Data Node reports to Name Node “block received” with block info

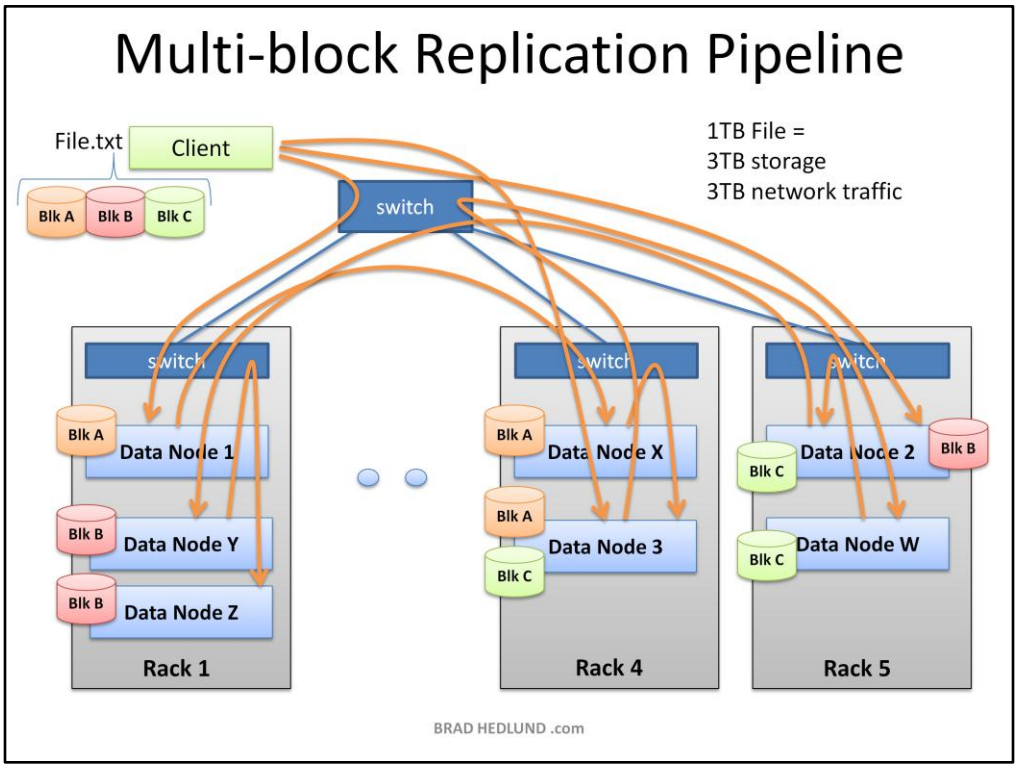
Data Node (1) sends a string of 00:00:00:00:00:00:00:05:00:00:00:00:00:00:00 to Client – some sort of complete signal?

Client sends a string of 00000000 to Data Node (1) – acknowledgement of the completion?

Data Node (1) closes TCP connection to Client.
Client reports “success” to Name Node.
Name Node knows (3) replicas of the block exist.

Data Nodes proceed to provide usual Block Reports and Heartbeats to Name Node

~Entire process repeats for remaining blocks of File.txt~
Next block is not transferred to HDFS until previous block successfully completes.



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.

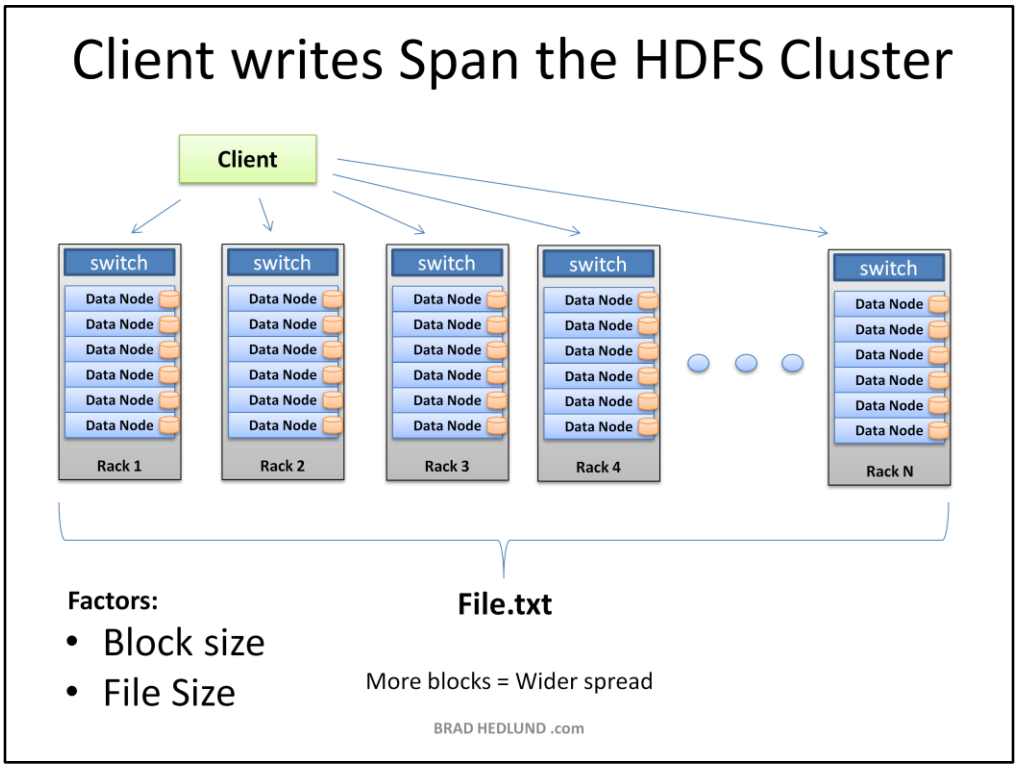
++++
Client writes first block copy to any given Data Node (1) in any given rack.

While receiving data from the Client, Data Node (1) begin sending block data to a second Data Node (2) in a different rack.

While receiving data from Data Node (1), Data Node (2) begins sending block data to a third Data Node (3) in the same rack.

Only when all three Data Nodes have successfully received the entire block, the Client will repeat the process for the subsequent block.

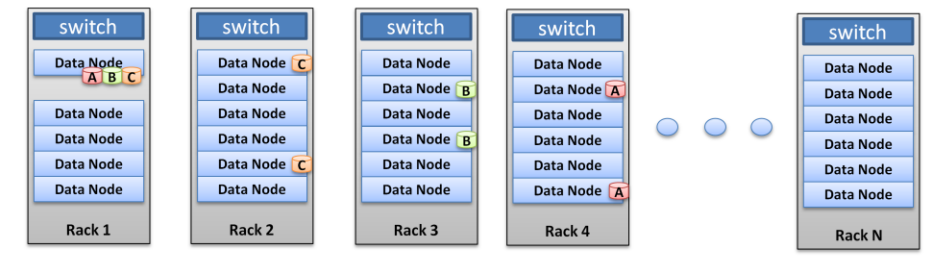
Any given block will only exist in (2) racks of the cluster. One copy in one rack, two copies in another rack.

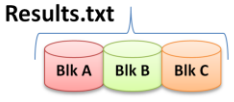


After the replication pipeline of each block is complete the file is successfully written to the cluster. As intended the file is spread in blocks across the cluster of machines, each machine having a relatively small part of the data. The more blocks that make up a file, the more machines the data can 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.

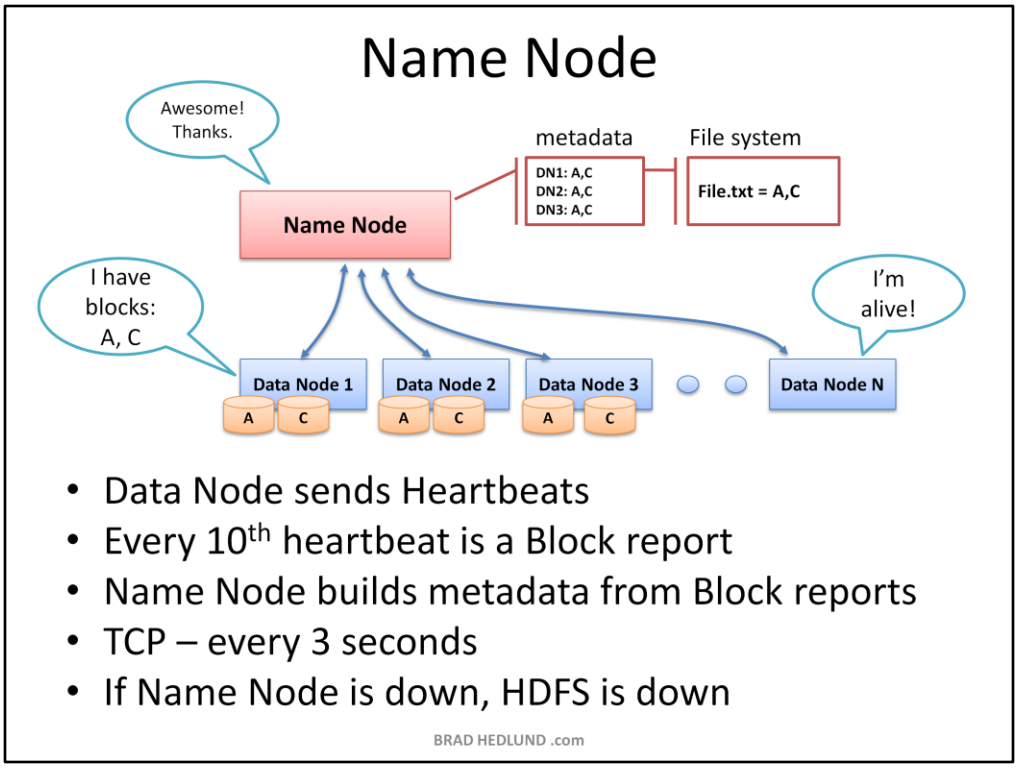
Another approach to scaling the cluster is to go **deep**. This is where you scale up the machines with more disk drives and more CPU cores. Instead of increasing the number of machines you begin to look at increasing the density of each machine. In scaling deep, you put yourself on a trajectory where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

Data Node writes span itself, and other racks





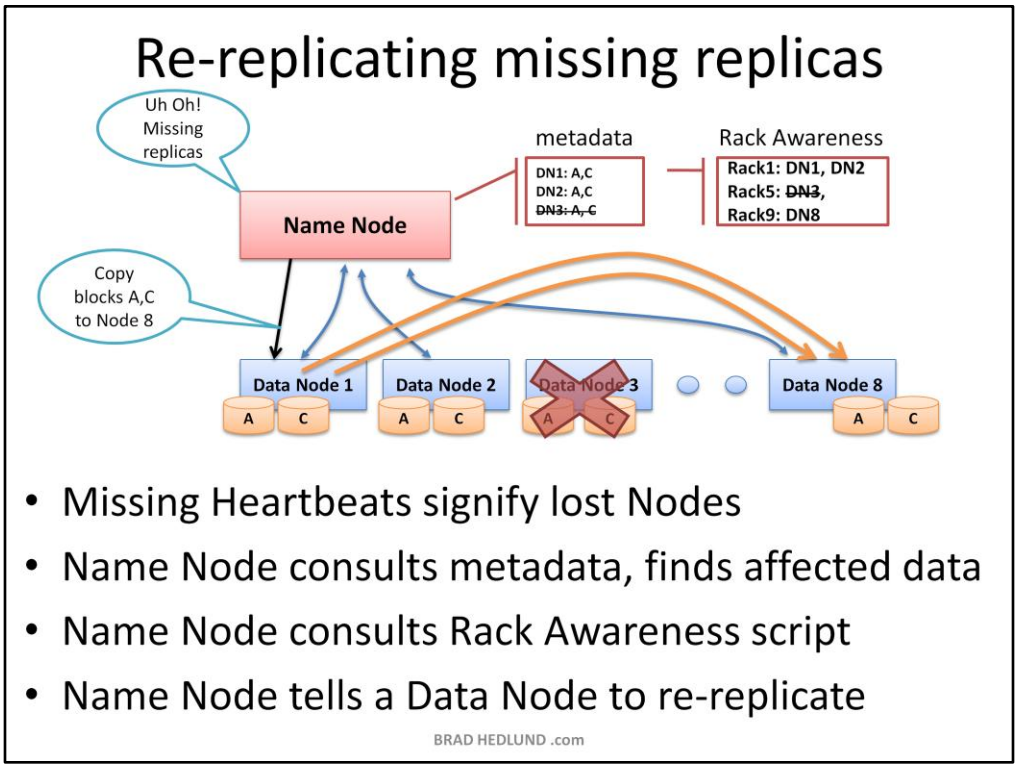
BRAD HEDLUND .com



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.

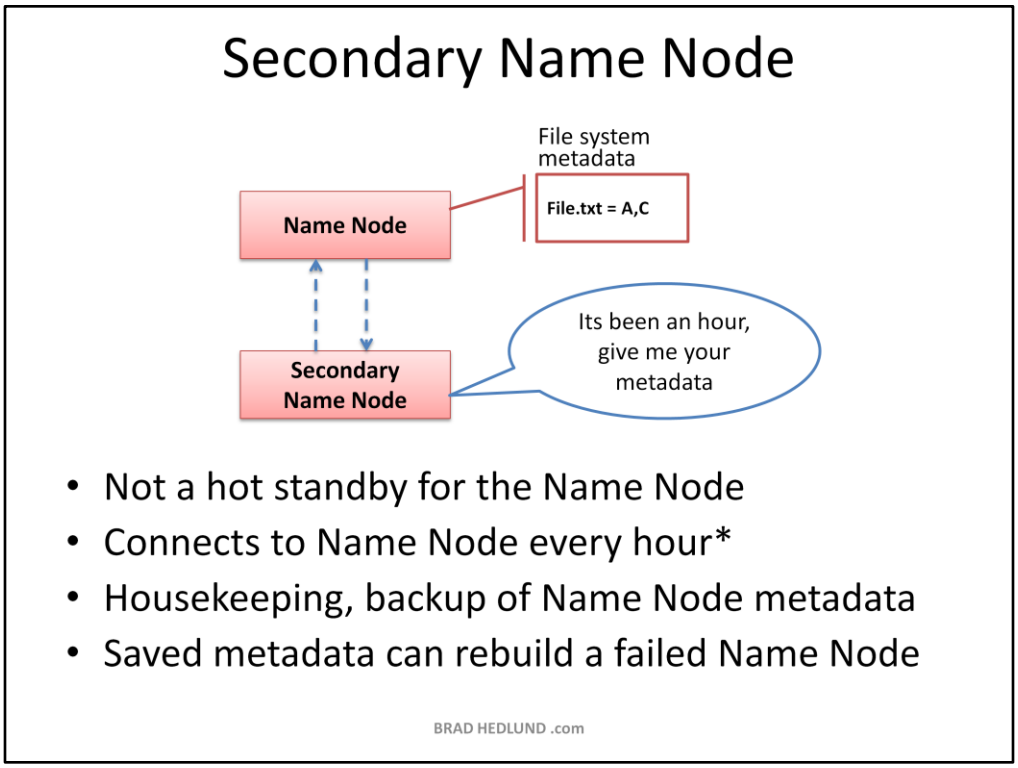
Data Nodes send heartbeats to the Name Node every 3 seconds via a TCP handshake, using the same port number defined for the Name Node daemon, usually TCP 9000. Every tenth heartbeat is a Block Report, where the Data Node tells the Name Node about all the blocks it has. The block reports allow the Name 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.
++++++
1) Data Node sends heart beat or block report
2) Name Node ACK
3) Data Node acknowledges the ACK



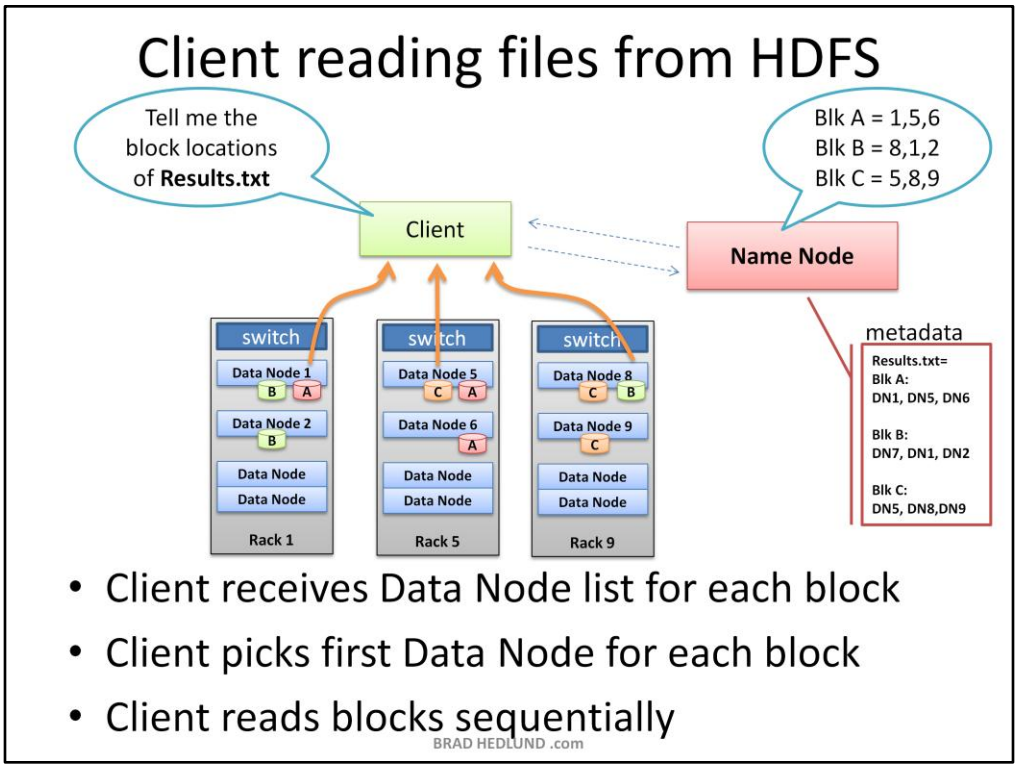
If the Name Node stops receiving heartbeats from a Data Node it presumes it to be dead and any data it had to be gone as well. Based on the block reports it had been receiving from the dead node, the Name Node knows which copies of blocks died along with the node and can make the decision to re-replicate 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.



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, ever 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.



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.

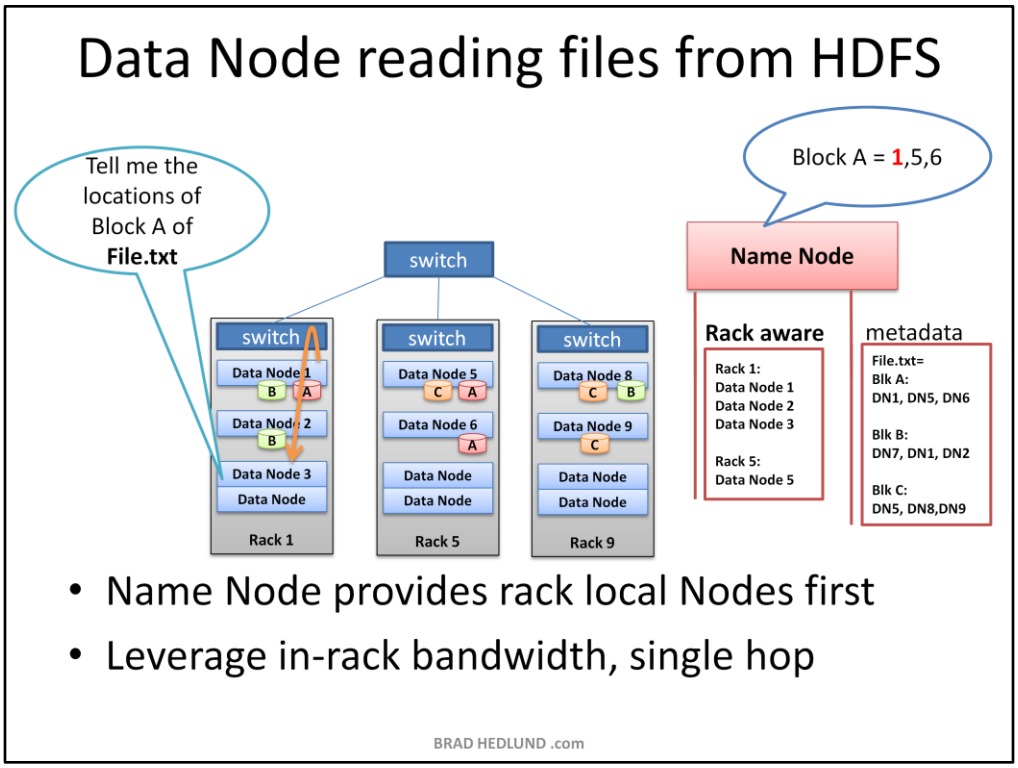
++++++
Client asks Name Node for the block locations of File.txt

Name Node provides Client a unique list of (3) Data Nodes for each block

Client chooses the first Data Node in each list for reading the block

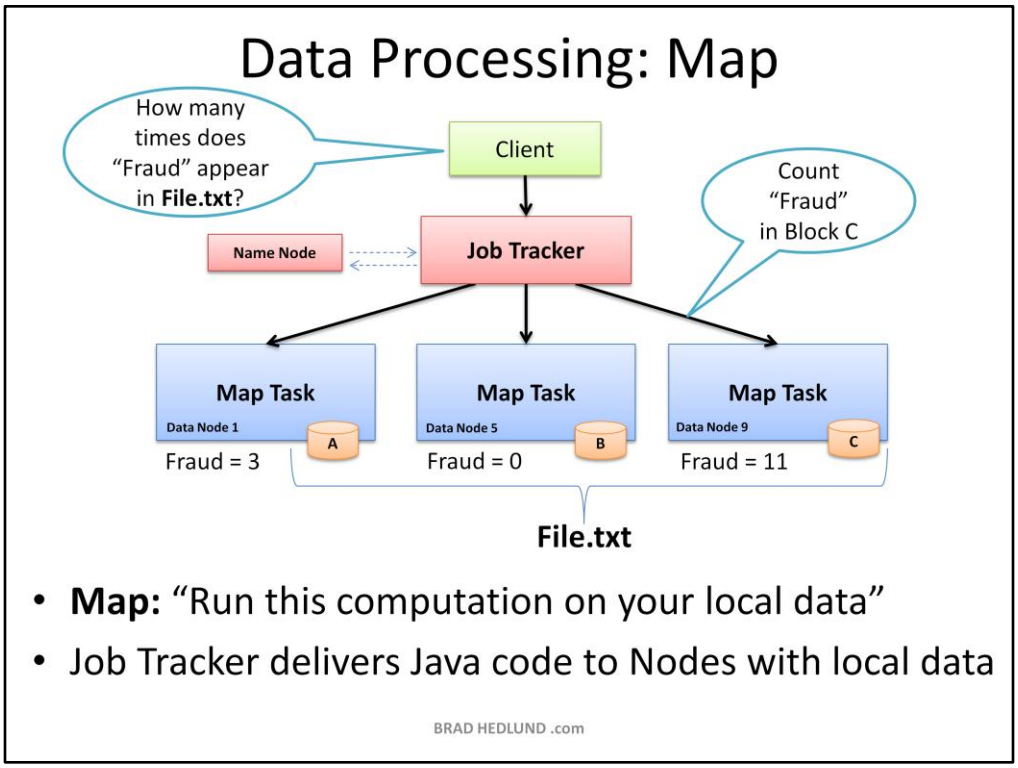
Blocks are read sequentially

Reading subsequent blocks does not begin until the previous block finishes



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.

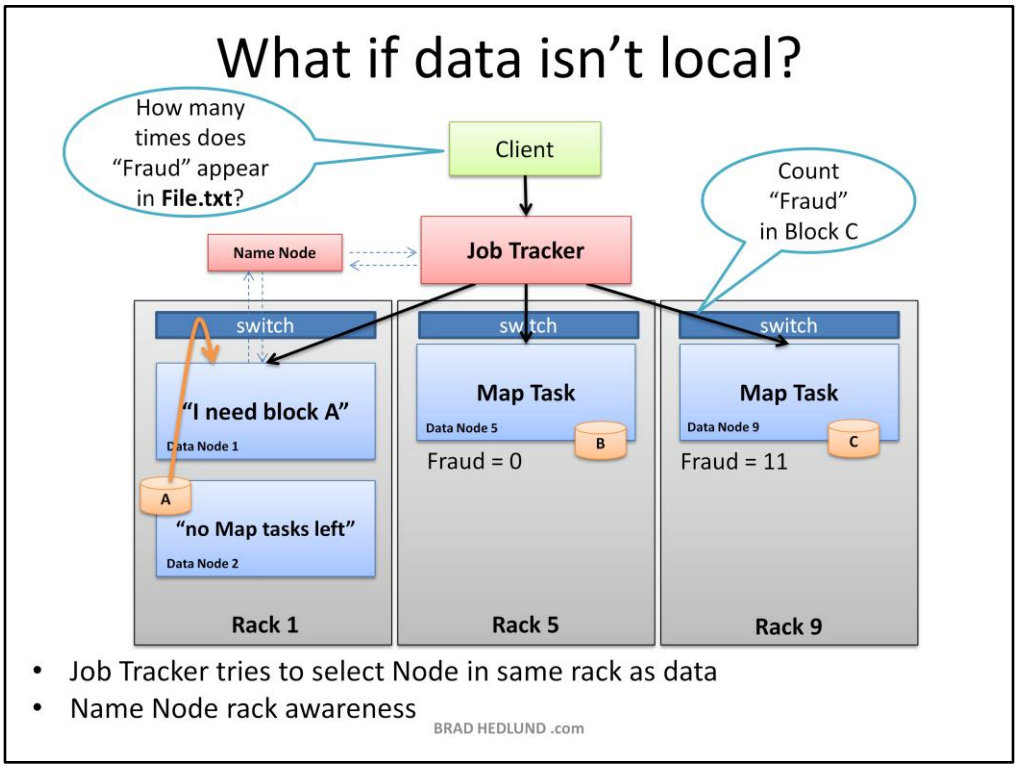


Now that File.txt is spread in small blocks across my cluster of machines I have the opportunity to provide extremely fast and efficient parallel processing of that data. The parallel processing framework included with Hadoop is called Map Reduce, named after two important steps in the model; **Map**,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

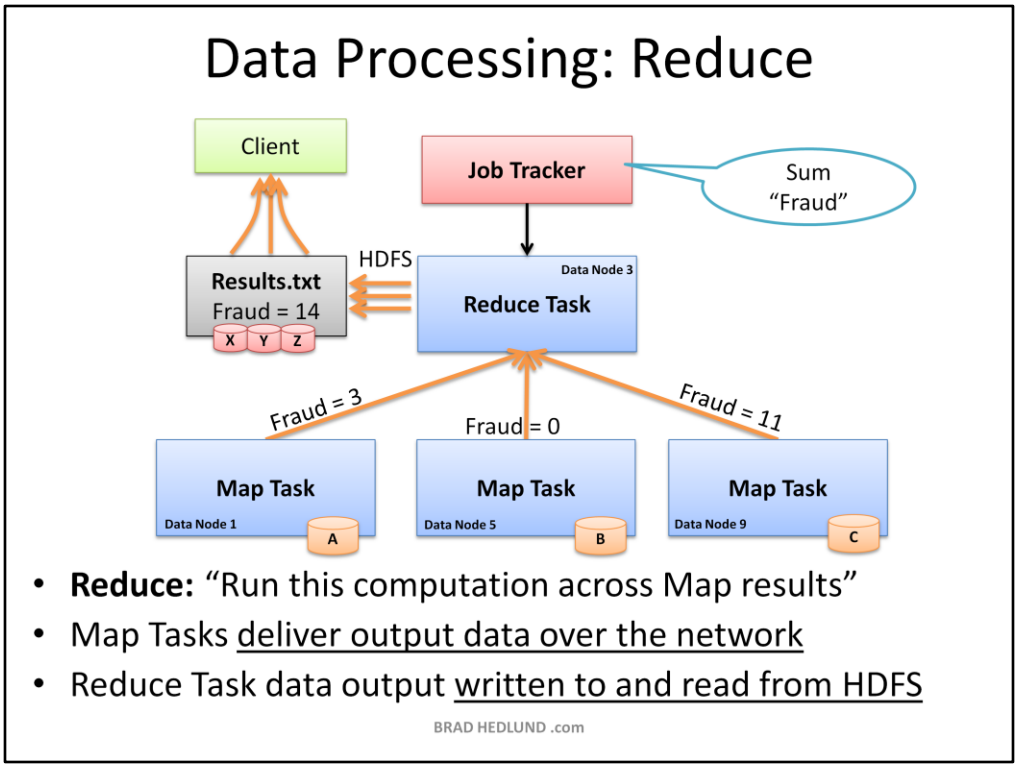
To start this process the Client machine submits the Map Reduce job to the Job Tracker, asking "How many times does Refund occur in File.txt" (paraphrasing Java code). The Job Tracker consults the Name Node to learn which Data Nodes have blocks of File.txt. The Job Tracker then provides the Task Tracker 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.



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.



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.

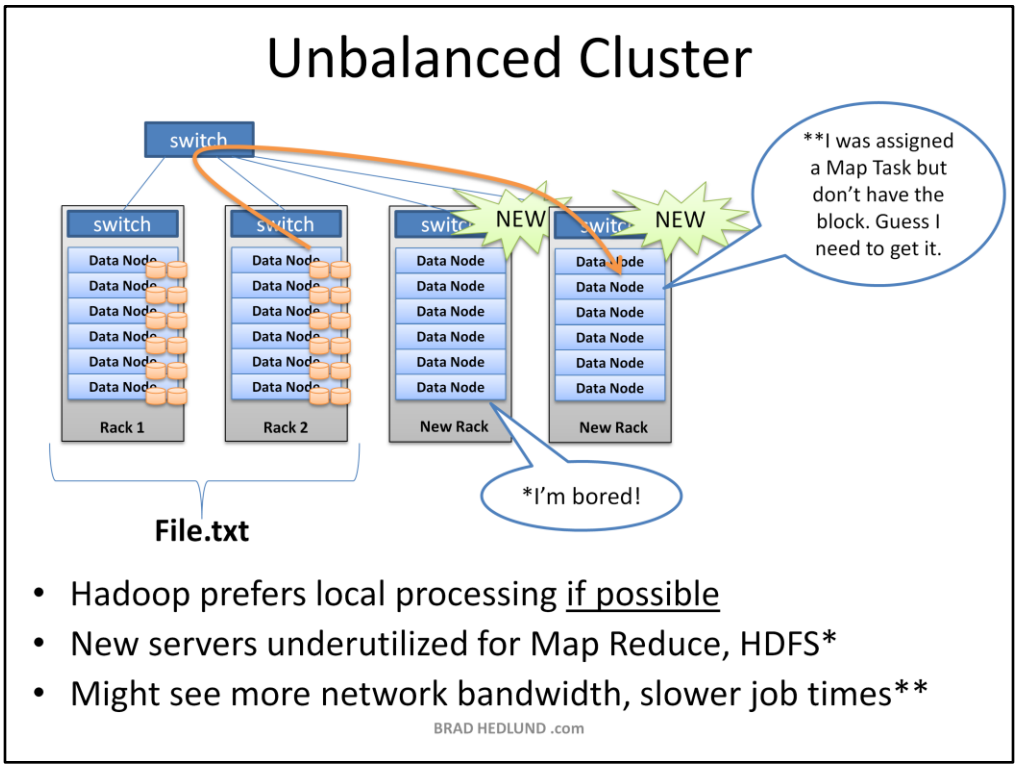
The Job Tracker starts a Reduce task on any one of the nodes in the cluster and instructs the Reduce task to go grab the intermediate data from all of the completed Map tasks. The Map tasks may respond to the Reducer almost simultaneously, resulting in a situation where you have a number of nodes sending 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 The output from the job is a file called Results.txt that is written to HDFS following all of the processes we have covered already; splitting the file up into blocks, pipeline replication of those blocks, etc. When complete, the Client machine can read the Results.txt file from HDFS, and the job is considered complete.

Our simple word count job did not result in a lot of intermediate data to transfer over the network. Other jobs however may produce a lot of intermediate data -- such as sorting

a terabyte of data. Where the output of the Map Reduce job is a new set of data equal to the size of data you started with. How much traffic you see on the network in the Map Reduce process is entirely dependent on the type job you are running at that given time.

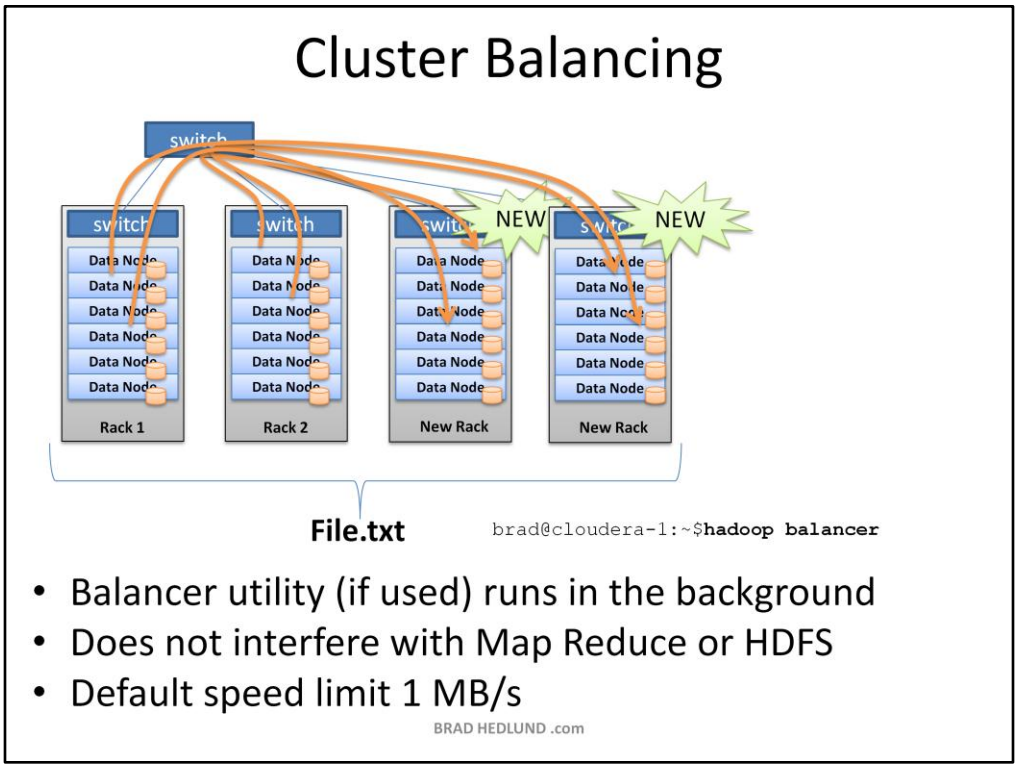
If you're a studious network administrator, you would learn more about Map Reduce and the types of jobs your cluster will be running, and how the type of job affects the traffic flows on your network. If you're a Hadoop networking rock star, you might even be able to suggest ways to better code the Map Reduce jobs so as to optimize the performance of the network, resulting in faster job completion 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 as result you may see more network traffic and slower job completion times.



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? I think so.

This material is based on studies, training from Cloudera, and observations from my own virtual Hadoop lab of six nodes. Everything discussed here is based on the latest stable release of Cloudera's CDH3 distribution of Hadoop. There are new and interesting technologies coming to Hadoop such as Hadoop on Demand (HOD) and HDFS Federations, not discussed here, but worth investigating on your own if so inclined.

Author:

Brad Hedlund
http://bradhedlund.com
brad.hedlund@gmail.com