**1. Career with Hadoop:**

**According to a Forbes report of 2015, about 90% of global organizations report medium to high levels of investment in big data analytics, and about a third call their investments “very significant.” Most importantly, about two-thirds of respondents report that big data and analytics initiatives have had a significant, measurable impact on revenues.**

Resources to Help :

* How Big data training can change your organization
* Learn How Big Data has Impacted players like Sears, JP Morgan, etc.

Hadoop skills are in demand – this is an undeniable fact! Hence, there is an urgent need for IT professionals to keep themselves in trend with Hadoop and Big Data technologies.

Resources to Help :

* What are the skills taught in Big Data and Hadoop Course ?
* To create your own Use-Case and implement Hadoop, join Big Data and Hadoop course now!

Apache Hadoop provides you with means to ramp up your career and gives you the following advantages:

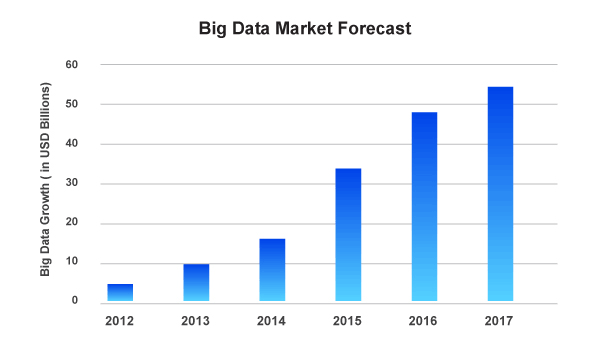
* Accelerated career growth.
* Increased pay package due to Hadoop skill.

**2. More Job Opportunities with Apache Hadoop:**

Looking at the Big Data market forecast, it looks promising and the upward trend will keep progressing with time. Hence, the job trend or Market is not a short lived phenomenon as Big Data and its technologies are here to stay. Hadoop has the potential to improve job prospects whether you are a fresher or an experienced professional.

Resources to Help :

* Gartner predicts how Big Data can lead economic growth
* Be a part of the Big Data Revolution!

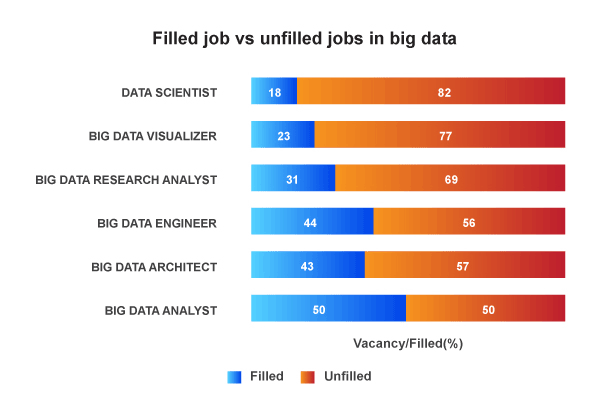
[](http://cdn.edureka.co/blog/wp-content/uploads/2014/03/slide-21.png)

**A research report by Avendus Capital estimates that the IT market for big data in India is hovering around $1.15 billion as 2015 comes to an end. This contributed to one fifth of India’s KPO market worth $5.6 billion. Also, The Hindu predicts that by end of 2018, India alone will face a shortage of close to two lakh Data Scientists. This presents a tremendous career and growth opportunity.**

This skill gap in Big Data can be bridged through comprehensive learning of Apache Hadoop that enables professionals and freshers alike, to add the valuable Big Data skills to their profile.

Resources to Help :

* Hadoop Developer job responsibilities and skills
* Want to master these skills? Get your seat reserved now!

[](http://cdn.edureka.co/blog/wp-content/uploads/2014/03/sad.png)

This is a perfect opportunity to take advantage of this positive trend and reap its benefits through appropriate learning of Hadoop.

Resources to Help :

* Hadoop Learner’s Profile
* Don’t trust anyone. Be the Judge. Watch sample class recording

**3. Look who is employing:**

* Start Learning

LinkedIn is the best place to get information on the number of existing Hadoop professional. The above info graph talks about the top companies employing Hadoop professionals and who is leading of them all. Yahoo! happens to be leading in this race.

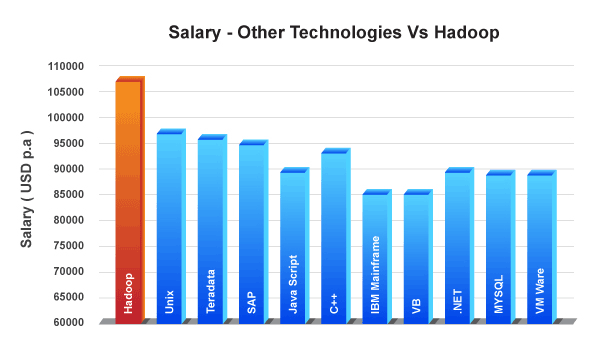
**4. Big Data and Hadoop equal Big Bucks!**

Dice has quoted, “**Technology professionals should be volunteering for Big Data projects, which makes them more valuable to their current employer and more marketable to other employers.**”

Get started with Big Data and Hadoop !

You can watch the sample class recording for Edureka’s Big Data and Hadoop course here:

“Companies are betting big that harnessing data can play a major role in their competitive plans, and that is leading to high pay for critical skills,” said Shravan Goli, president of Dice, in a statement.

[](http://cdn.edureka.co/blog/wp-content/uploads/2014/03/slide-51.png)

Alice Hill, managing director of Dice, tells *Data Informed*, that the postings for Hadoop jobs has gone up by 64%**,** compared to last year. And that **Hadoop** **is the leader in the Big Data category** of job postings. According to Dice, Hadoop pros made an average of **$108,669** in 2013, which is slightly above the **$106,542** average for Big Data jobs.

**5. Top Hadoop Technology Companies:**

[](http://cdn.edureka.co/blog/wp-content/uploads/2014/03/Top-Hadoop-Companies_Final.png)

**Ideally What should be the Block size to get maximum performance from Hadoop cluster?  
What are the effects when I increase / decrease block size ?  
What are factors I should consider when I change block size ?  
Is there any thumb rule to start with ?**

**Answer to this question is ”it depends on Input Data”.**

**There is no as such rule set by hadoop to bound user with certain block size. Usually, it depends on the input data. If you want to maximize throughput for a very large input file, using very large blocks (may be 128MB or even 256MB) is best. But on the other hand for smaller files, using a smaller block size is better.**

**So we are talking about larger file large block & smaller file small blocks. In Industry we can get files of different sizes & we can have files with different block sizes on the same file system. So inorder to overcome that situation ”dfs.block.size” parameter can be used when the file is written. It will help you in overriding default block size written in hdfs-site.xml**

**There are few points to keep in mind for taking decision on what should be block size, each things has its own pros & cons:-**

**1. Most obviously, a file will have fewer blocks if the block size is larger. This can potentially make it possible for client to read/write more data without interacting with the Namenode which saves time.**

**2. Having larger blocks also reduces the metadata size of the Namenode, reducing Namenode load.**

**3. With fewer blocks, the file may potentially be stored on fewer nodes in total, this can reduce total throughput for parallel access**

**4. Having fewer & larger blocks, also means longer tasks which in turn may not gain maximum parallelism**

**5. Also while larger block is being processed and some failure occur more work need to be done.**

# Monitoring the Health and Status of Services

From the **Services** page, you can:

* Monitor the health and status of the services running on your clusters.
* Manage the services and roles in your clusters.
* Add new services.
* Access the client configuration files generated by Cloudera Manager that enable Hadoop client users to work with the HDFS, MapReduce, HBase, and YARN services you added. (Note that these configuration files are normally deployed automatically when you install your cluster or add a service).
* View the Maintenance Mode status of your cluster.
* Install an additional cluster. After initial installation, you can use the **Add Cluster** wizard to add and configure an additional cluster. See Managing Multiple Clusters and Adding a Cluster for more information on this topic.

You can also pull down a menu from an individual service name to go directly to one of the tabs for that service – to its status, instances, commands, configuration, audits, or charts tabs.

## **Service Health and Status**

To view the status of your services, click the **Services** tab and select **All Services**. The Services page opens and displays an overview of the service instances currently installed on your cluster.

For each service instance, this page shows:

* The type of service
* The service status (for example, Started)
* The overall health of the service
* The type and number of the roles that have been configured for that service instance.

http://www.cloudera.com/documentation/archive/manager/4-x/4-5-2/static/note.jpg  **Note**:

By default, the All Services page shows the **current** state of the services in your cluster. By moving the Time Marker ( images/image7.jpeg), you can see what the status was at any point in the past. When you are looking at the past, the **Actions** menus and most other commands are disabled, and Role Counts information may not be accurate. Click the Current Time button ( images/image10.jpeg) to return to the current time.

See Selecting the Time Range for details of how time range selection works in Cloudera Manager.

## **Add a Service**

After initial installation, you can use the **Add a Service** wizard to add and configure (but not start) new service instances. The **Add a Service...** command is found under the cluster **Actions** menu for the cluster where you want to add the service.

The cluster **Actions** menu, and thus the **Add a Service...** command, is not available if you are viewing status for a point of time in the past.

See Adding Services for more information on this topic.

## **View the URLs of the Client Configuration Files**

To allow Hadoop client users to work with the HDFS, MapReduce, YARN and HBase services you created, Cloudera Manager generates client configuration files that contain the relevant configuration files with the settings from your services. These files are deployed automatically by Cloudera Manager based on the services you have installed, when you add a service, or when you add a Gateway role on a host.

You can download and distribute these client configuration files manually to the users of a service, if necessary.

The **Client Configuration URLs** command on the cluster **Actions** menu opens a pop-up that displays links to the client configuration zip files created for the services installed in your cluster. You can download these zip files by clicking the link.

The **Client Configuration URLs** button is not available if you are viewing status for a point of time in the past.

See Deploying Client Configuration Files for more information on this topic.

## **View the Health and Status of a Service Instance or Role Instance**

**To see the status of a service instance:**

* Click the link in the **Name** column,   OR
* Click the health status associated with the instance,   OR
* From the **Services** tab, select the service instance you want to see.

This will open the **Status** page where you can view a variety of information about a service and its performance. See Viewing Service Status for details. **To see the status of a role instance:**

* Click the role instance under the **Role Counts** column.

If there is just one instance of this role, this opens the **Status** tab for the role instance.

If there are multiple instances of a role, clicking the role link under **Role Counts** will open the **Instances** tab for the service, showing instances of the role type you have selected. See Viewing Status for a Role Instance for details.

If you are viewing a past point in time, the Role Count links will be greyed out, but still functional. Their behavior will depend on whether historical data is available for the role instance.

## **Viewing the Maintenance Mode Status of a Cluster**

* Click the **View Maintenance Mode Status** button to view the status of your cluster in terms of which components (service, roles or hosts) are in maintenance mode.

This pops up a dialog box that shows the components in your cluster that are in maintenance mode, and indicates which are in effective maintenance mode as well as those that have been placed into maintenance mode explicitly. (See Maintenance Mode for an explanation of explicit maintenance mode and effective maintenance mode.)

From this dialog box you can select any of the components shown there, and remove them from maintenance mode.

If individual services are in maintenance mode, you will see the maintenance mode icon next to the **Actions** button for that service.

The **View Maintenance Mode Status** button is not available if you are viewing status for a point of time in the past.

## **The Actions Menus**

There are two **Actions** menu available on the **All Services** page: one for the cluster, and one for each service.

**Actions for a Cluster**

There are multiple actions you can take at a cluster level:

* Stop, Start, or Restart all the services in the cluster
* Deploy the client configurations onto the appropriate nodes of the cluster, or view the client configuration file URLs.
* Upgrade the cluster
* Rename the cluster
* Enter or exit Maintenance Mode for the cluster.

### Actions for a Service

There is an **Actions** menu associated with each service instance installed on your cluster. From the **Actions** menu for a service you can:

* Stop, start, restart, or delete the service (the available actions depend on the current status of the associated service – for example, you cannot Start a Started service).
* Change the display name of a service
* Enter or exit Maintenance Mode for the service.

These actions are covered in the Services Configuration section of this document:

* Starting, Stopping, and Restarting Services
* Deleting Service Instances and Role Instances
* Renaming a Service to change its display name

***8.2. Architecture***

***8.2.1. NameNode***

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file), and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of blocks to DataNodes. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

***8.2.2. Image and Journal***

The inodes and the list of blocks that define the metadata of the name system are called the image. NameNode keeps the entire namespace image in RAM. The persistent record of the image stored in the NameNode's local native filesystem is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native filesystem. The location of block replicas are not part of the persistent checkpoint.

Each client-initiated transaction is recorded in the journal, and the journal file is flushed and synced before the acknowledgment is sent to the client. The checkpoint file is never changed by the NameNode; a new file is written when a checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal. A new checkpoint and an empty journal are written back to the storage directories before the NameNode starts serving clients.

For improved durability, redundant copies of the checkpoint and journal are typically stored on multiple independent local volumes and at remote NFS servers. The first choice prevents loss from a single volume failure, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize this process, the NameNode batches multiple transactions. When one of the NameNode's threads initiates a flush-and-sync operation, all the transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

***8.2.3. DataNodes***

Each block replica on a DataNode is represented by two files in the local native filesystem. The first file contains the data itself and the second file records the block's metadata including checksums for the data and the generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional filesystems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode, the DataNode automatically shuts down.

The namespace ID is assigned to the filesystem instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus protecting the integrity of the filesystem. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.

A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block ID, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster.

During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's block allocation and load balancing decisions.

The NameNode does not directly send requests to DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, and shut down the node.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

***8.2.4. HDFS Client***

User applications access the filesystem using the HDFS client, a library that exports the HDFS filesystem interface.

Like most conventional filesystems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application does not need to know that filesystem metadata and storage are on different servers, or that blocks have multiple replicas.

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. The list is sorted by the network topology distance from the client. The client contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Choice of DataNodes for each block is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 8.1.

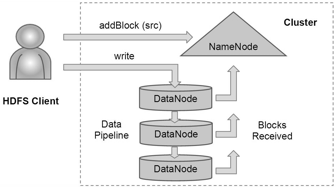


Figure 8.1: HDFS Client Creates a New File

Unlike conventional filesystems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves tolerance against faults and increases read bandwidth.

***8.2.5. CheckpointNode***

The NameNode in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a CheckpointNode or a BackupNode. The role is specified at the node startup.

The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode.

Creating periodic checkpoints is one way to protect the filesystem metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal are unavailable. Creating a checkpoint also lets the NameNode truncate the journal when the new checkpoint is uploaded to the NameNode. HDFS clusters run for prolonged periods of time without restarts during which the journal constantly grows. If the journal grows very large, the probability of loss or corruption of the journal file increases. Also, a very large journal extends the time required to restart the NameNode. For a large cluster, it takes an hour to process a week-long journal. Good practice is to create a daily checkpoint.

***8.2.6. BackupNode***

A recently introduced feature of HDFS is the BackupNode. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an in-memory, up-to-date image of the filesystem namespace that is always synchronized with the state of the NameNode.

The BackupNode accepts the journal stream of namespace transactions from the active NameNode, saves them in journal on its own storage directories, and applies these transactions to its own namespace image in memory. The NameNode treats the BackupNode as a journal store the same way as it treats journal files in its storage directories. If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state.

The BackupNode can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace into its local storage directories.

The BackupNode can be viewed as a read-only NameNode. It contains all filesystem metadata information except for block locations. It can perform all operations of the regular NameNode that do not involve modification of the namespace or knowledge of block locations. Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility of persisting the namespace state to the BackupNode.

***8.2.7. Upgrades and Filesystem Snapshots***

During software upgrades the possibility of corrupting the filesystem due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades.

The snapshot mechanism lets administrators persistently save the current state of the filesystem, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged.

During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the directories containing the data files as this would require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot; for snapshots created during upgrade, this finalizes the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The layout version identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.

***8.3. File I/O Operations and Replica Management***

Of course, the whole point of a filesystem is to store data in files. To understand how HDFS does this, we must look at how reading and writing works, and how blocks are managed.

***8.3.1. File Read and Write***

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgment for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.

After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the hflush operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all DataNodes in the pipeline acknowledge the successful transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

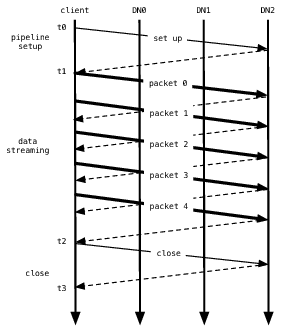


Figure 8.2: Data Pipeline While Writing a Block

If no error occurs, block construction goes through three stages as shown in Figure 8.2 illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t0 to t1 is the pipeline setup stage. The interval t1 to t2 is the data streaming stage, where t1 is the time when the first data packet gets sent and t2 is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits packet 2. The hflush indication travels with the packet data and is not a separate operation. The final interval t2 to t3 is the pipeline close stage for this block.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.

***8.3.2. Block Placement***

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. Figure 8.3 describes a cluster with two racks, each of which contains three nodes.

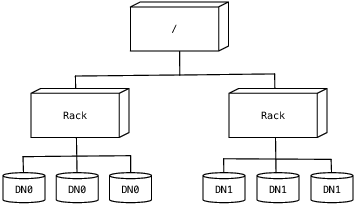


Figure 8.3: Cluster Topology

HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing the distances to their closest common ancestor. A shorter distance between two nodes means greater bandwidth they can use to transfer data.

HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs the configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test alternate policies that are optimal for their applications.

The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located. The second and the third replicas are placed on two different nodes in a different rack. The rest are placed on random nodes with restrictions that no more than one replica is placed at any one node and no more than two replicas are placed in the same rack, if possible. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack.

After all target nodes are selected, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order.

This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

***8.3.3. Replication Management***

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.

When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas.

The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as mis-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the over-replication policy prefers not to reduce the number of racks.

***8.3.4. Balancer***

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes with a lot of free storage. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster.

The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node3 differs from the utilization of the whole cluster4 by no more than the threshold value.

The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.

The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A.

A configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

***8.3.5. Block Scanner***

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.

The verification time of each block is stored in a human-readable log file. At any time there are up to two files in the top-level DataNode directory, the current and previous logs. New verification times are appended to the current file. Correspondingly, each DataNode has an in-memory scanning list ordered by the replica's verification time.

Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

***8.3.6. Decommissioning***

The cluster administrator specifies list of nodes to be decommissioned. Once a DataNode is marked for decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

***8.3.7. Inter-Cluster Data Copy***

When working with large datasets, copying data into and out of a HDFS cluster is daunting. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination filesystem. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

***8.4. Practice at Yahoo!***

Large HDFS clusters at Yahoo! include about 4000 nodes. A typical cluster node has two quad core Xeon processors running at 2.5 GHz, 4–12 directly attached SATA drives (holding two terabytes each), 24 Gbyte of RAM, and a 1-gigabit Ethernet connection. Seventy percent of the disk space is allocated to HDFS. The remainder is reserved for the operating system (Red Hat Linux), logs, and space to spill the output of map tasks (MapReduce intermediate data are not stored in HDFS).

Forty nodes in a single rack share an IP switch. The rack switches are connected to each of eight core switches. The core switches provide connectivity between racks and to out-of-cluster resources. For each cluster, the NameNode and the BackupNode hosts are specially provisioned with up to 64 GB RAM; application tasks are never assigned to those hosts. In total, a cluster of 4000 nodes has 11 PB (petabytes; 1000 terabytes) of storage available as blocks that are replicated three times yielding a net 3.7 PB of storage for user applications. Over the years that HDFS has been in use, the hosts selected as cluster nodes have benefited from improved technologies. New cluster nodes always have faster processors, bigger disks and larger RAM. Slower, smaller nodes are retired or relegated to clusters reserved for development and testing of Hadoop.

On an example large cluster (4000 nodes), there are about 65 million files and 80 million blocks. As each block typically is replicated three times, every data node hosts 60 000 block replicas. Each day, user applications will create two million new files on the cluster. The 40 000 nodes in Hadoop clusters at Yahoo! provide 40 PB of on-line data storage.

Becoming a key component of Yahoo!'s technology suite meant tackling technical problems that are the difference between being a research project and being the custodian of many petabytes of corporate data. Foremost are issues of robustness and durability of data. But also important are economical performance, provisions for resource sharing among members of the user community, and ease of administration by the system operators.

***8.4.1. Durability of Data***

Replication of data three times is a robust guard against loss of data due to uncorrelated node failures. It is unlikely Yahoo! has ever lost a block in this way; for a large cluster, the probability of losing a block during one year is less than 0.005. The key understanding is that about 0.8 percent of nodes fail each month. (Even if the node is eventually recovered, no effort is taken to recover data it may have hosted.) So for the sample large cluster as described above, a node or two is lost each day. That same cluster will re-create the 60 000 block replicas hosted on a failed node in about two minutes: re-replication is fast because it is a parallel problem that scales with the size of the cluster. The probability of several nodes failing within two minutes such that all replicas of some block are lost is indeed small.

Correlated failure of nodes is a different threat. The most commonly observed fault in this regard is the failure of a rack or core switch. HDFS can tolerate losing a rack switch (each block has a replica on some other rack). Some failures of a core switch can effectively disconnect a slice of the cluster from multiple racks, in which case it is probable that some blocks will become unavailable. In either case, repairing the switch restores unavailable replicas to the cluster. Another kind of correlated failure is the accidental or deliberate loss of electrical power to the cluster. If the loss of power spans racks, it is likely that some blocks will become unavailable. But restoring power may not be a remedy because one-half to one percent of the nodes will not survive a full power-on restart. Statistically, and in practice, a large cluster will lose a handful of blocks during a power-on restart.

In addition to total failures of nodes, stored data can be corrupted or lost. The block scanner scans all blocks in a large cluster each fortnight and finds about 20 bad replicas in the process. Bad replicas are replaced as they are discovered.

***8.4.2. Features for Sharing HDFS***

As the use of HDFS has grown, the filesystem itself has had to introduce means to share the resource among a large number of diverse users. The first such feature was a permissions framework closely modeled on the Unix permissions scheme for file and directories. In this framework, files and directories have separate access permissions for the owner, for other members of the user group associated with the file or directory, and for all other users. The principle differences between Unix (POSIX) and HDFS are that ordinary files in HDFS have neither execute permissions nor sticky bits.

In the earlier version of HDFS, user identity was weak: you were who your host said you are. When accessing HDFS, the application client simply queries the local operating system for user identity and group membership. In the new framework, the application client must present to the name system credentials obtained from a trusted source. Different credential administrations are possible; the initial implementation uses Kerberos. The user application can use the same framework to confirm that the name system also has a trustworthy identity. And the name system also can demand credentials from each of the data nodes participating in the cluster.

The total space available for data storage is set by the number of data nodes and the storage provisioned for each node. Early experience with HDFS demonstrated a need for some means to enforce the resource allocation policy across user communities. Not only must fairness of sharing be enforced, but when a user application might involve thousands of hosts writing data, protection against applications inadvertently exhausting resources is also important. For HDFS, because the system metadata are always in RAM, the size of the namespace (number of files and directories) is also a finite resource. To manage storage and namespace resources, each directory may be assigned a quota for the total space occupied by files in the sub-tree of the namespace beginning at that directory. A separate quota may also be set for the total number of files and directories in the sub-tree.

While the architecture of HDFS presumes most applications will stream large data sets as input, the MapReduce programming framework can have a tendency to generate many small output files (one from each reduce task) further stressing the namespace resource. As a convenience, a directory sub-tree can be collapsed into a single Hadoop Archive file. A HAR file is similar to a familiar tar, JAR, or Zip file, but filesystem operations can address the individual files within the archive, and a HAR file can be used transparently as the input to a MapReduce job.

***8.4.3. Scaling and HDFS Federation***

Scalability of the NameNode has been a key struggle [Shv10]. Because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap limits the number of files and also the number of blocks addressable. This also limits the total cluster storage that can be supported by the NameNode. Users are encouraged to create larger files, but this has not happened since it would require changes in application behavior. Furthermore, we are seeing new classes of applications for HDFS that need to store a large number of small files. Quotas were added to manage the usage, and an archive tool has been provided, but these do not fundamentally address the scalability problem.

A new feature allows multiple independent namespaces (and NameNodes) to share the physical storage within a cluster. Namespaces use blocks grouped under a Block Pool. Block pools are analogous to logical units (LUNs) in a SAN storage system and a namespace with its pool of blocks is analogous to a filesystem volume.

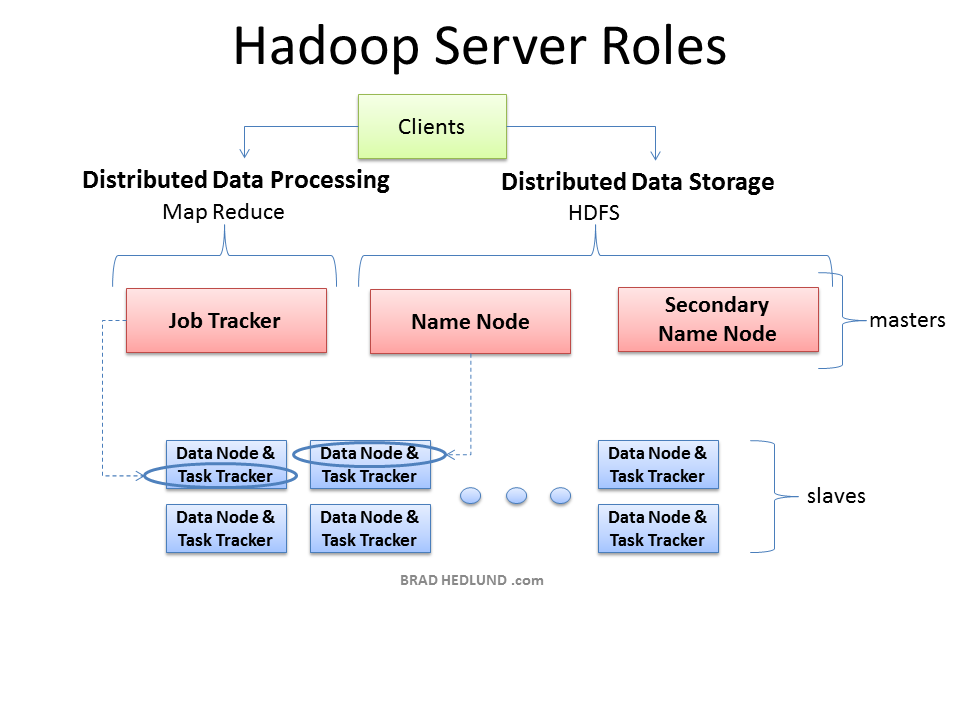
This approach offers a number of advantages besides scalability: it can isolate namespaces of different applications improving the overall availability of the cluster. Block pool abstraction allows other services to use the block storage with perhaps a different namespace structure. We plan to explore other approaches to scaling such as storing only partial namespace in memory, and truly distributed implementation of the NameNode.

Applications prefer to continue using a single namespace. Namespaces can be mounted to create such a unified view. A client-side mount table provide an efficient way to do that, compared to a server-side mount table: it avoids an RPC to the central mount table and is also tolerant of its failure. The simplest approach is to have shared cluster-wide namespace; this can be achieved by giving the same client-side mount table to each client of the cluster. Client-side mount tables also allow applications to create a private namespace view. This is analogous to the per-process namespaces that are used to deal with remote execution in distributed systems [PPT+93, Rad94, RP93].

# Understanding Hadoop Clusters and the Network

Sep 10, 2011 • Brad Hedlund

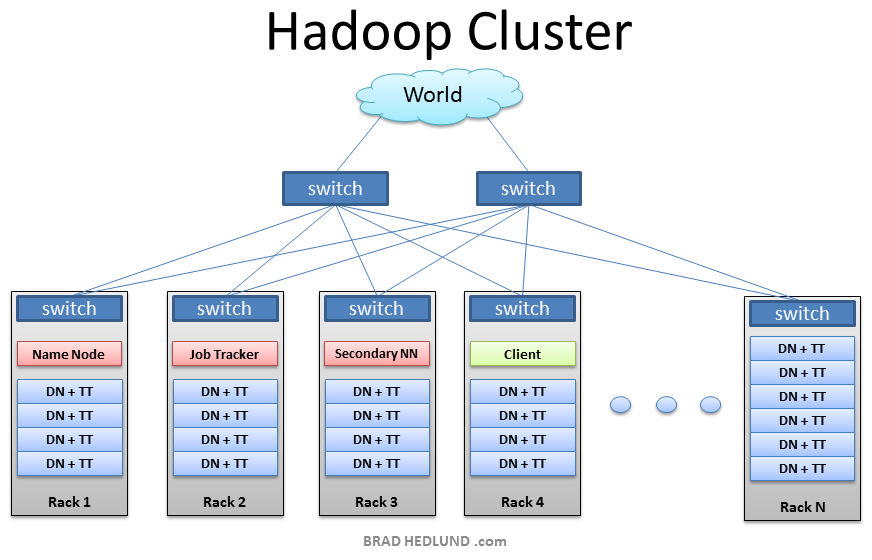
This article is Part 1 in series that will take a closer look at the architecture and methods of a Hadoop cluster, and how it relates to the network and server infrastructure. The content presented here is largely based on academic work and conversations I’ve had with customers running real production 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, let’s get started!



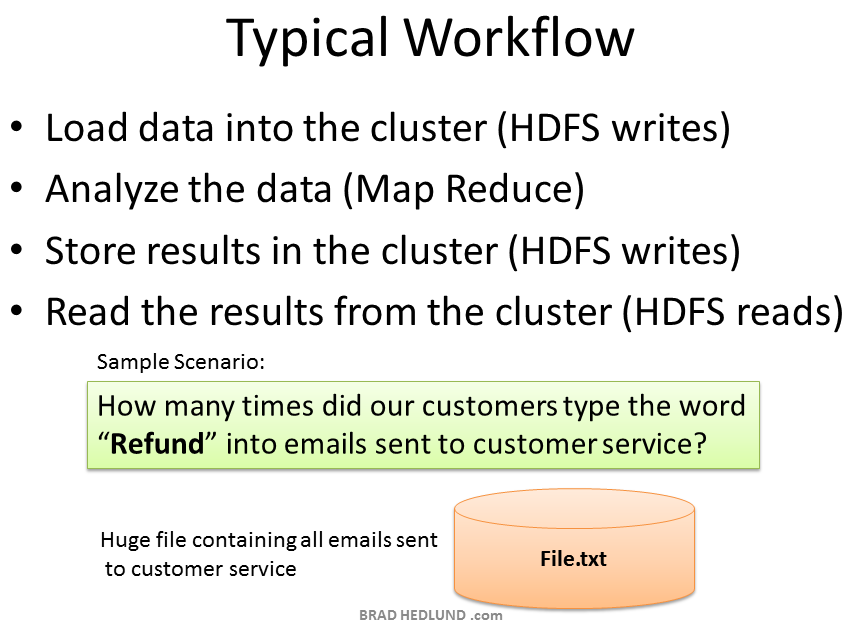
The three major categories of machine roles in a Hadoop deployment are Client machines, Masters nodes, and Slave nodes. The Master nodes oversee the two key functional pieces that make up Hadoop: storing lots of data (HDFS), and running parallel computations on all that data (Map Reduce). The Name Node 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.

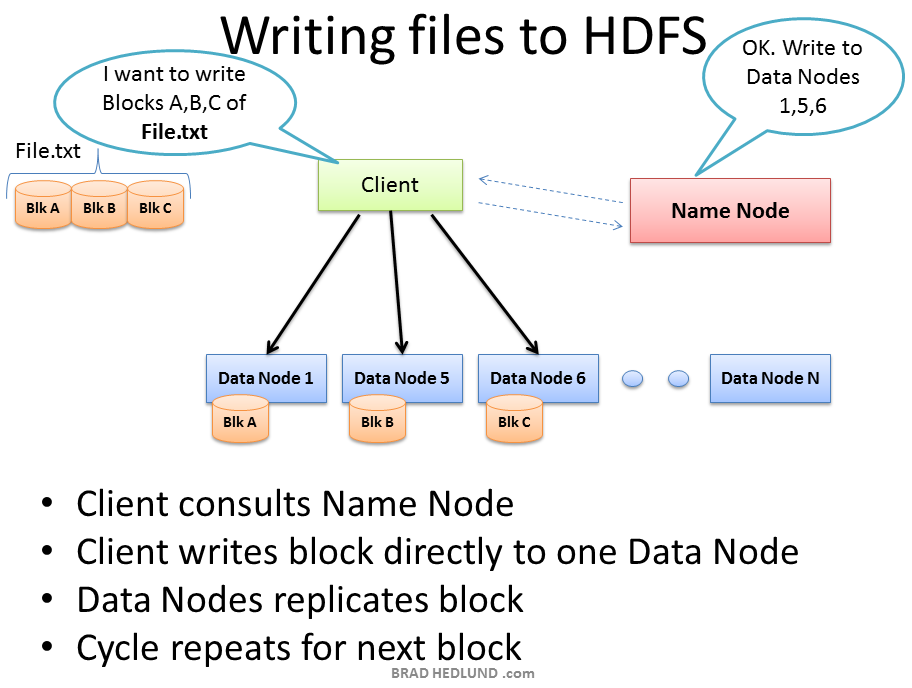
In real production clusters there is no server virtualization, no hypervisor layer. That would only amount to unnecessary overhead impeding performance. Hadoop runs best on Linux machines, working directly with the underlying hardware. That said, Hadoop does work in a virtual machine. That’s a great 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…

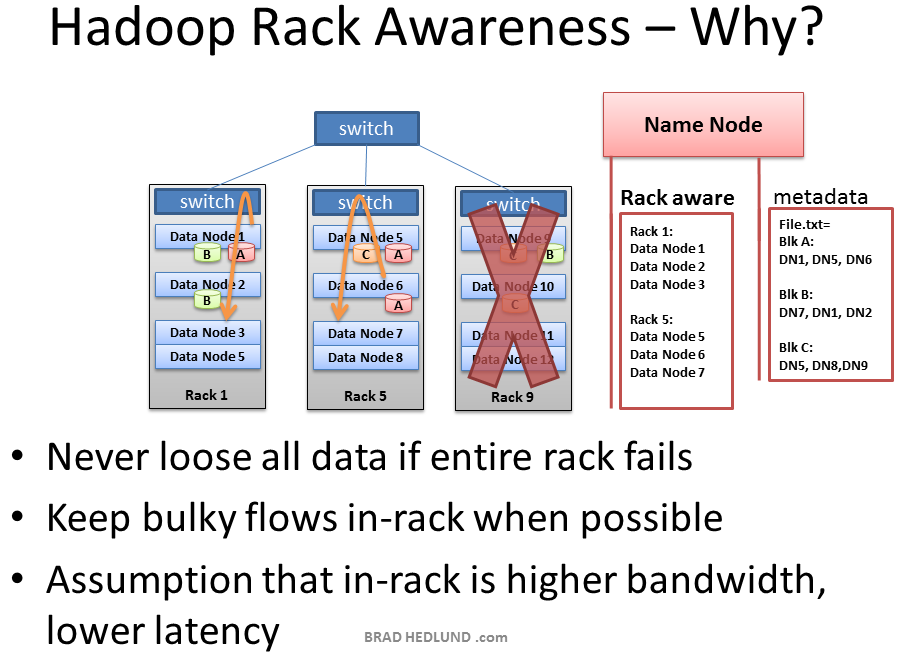


Why did Hadoop come to exist? What problem does it solve? Simply put, businesses and governments have a tremendous amount of data that needs to be analyzed and processed very quickly. If I can chop that huge chunk of data into small chunks and spread it out over many machines, and have all those machines 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).

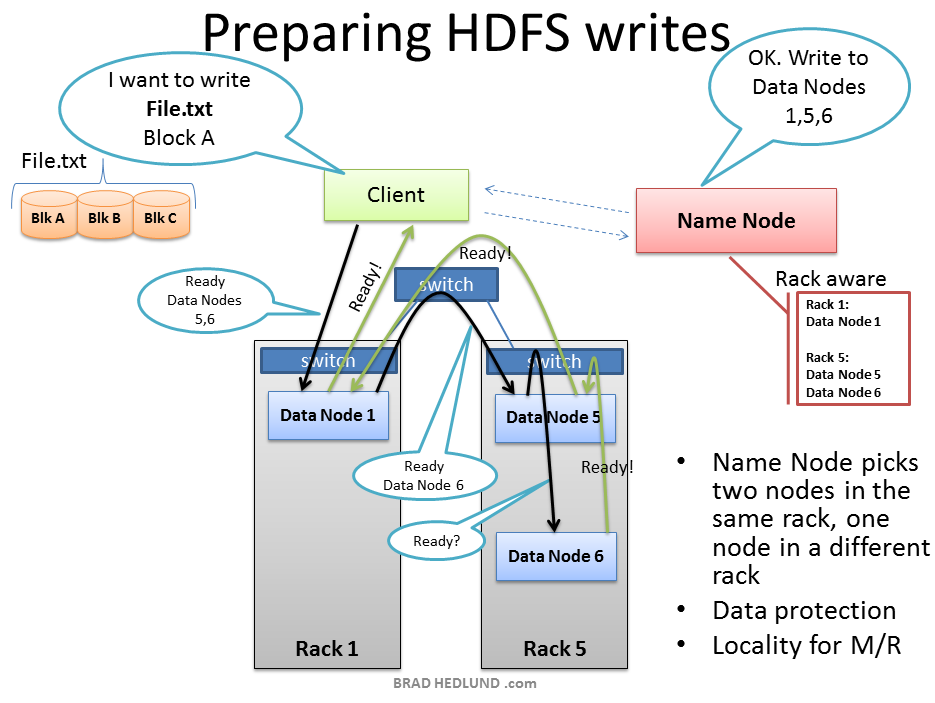


Hadoop has the concept of “Rack Awareness”. As the Hadoop administrator you can **manually** define the rack number of each slave Data Node in your cluster. Why would you go through the trouble of doing this? There are two key reasons for this: Data loss prevention, and network performance. Remember 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.

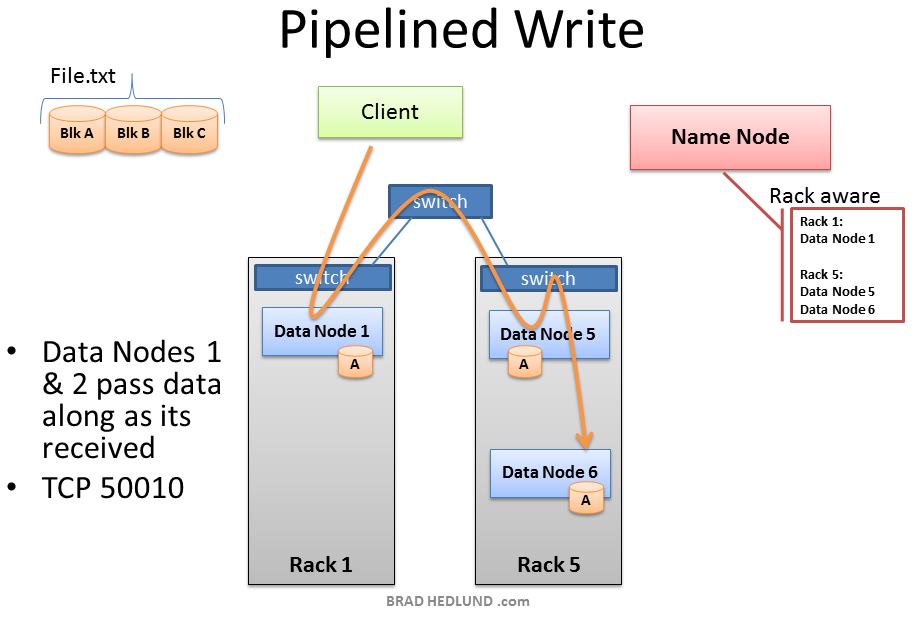
Even more interesting would be a OpenFlow network, where the Name Node could query the OpenFlow controller about a Node’s location in the topology.



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.

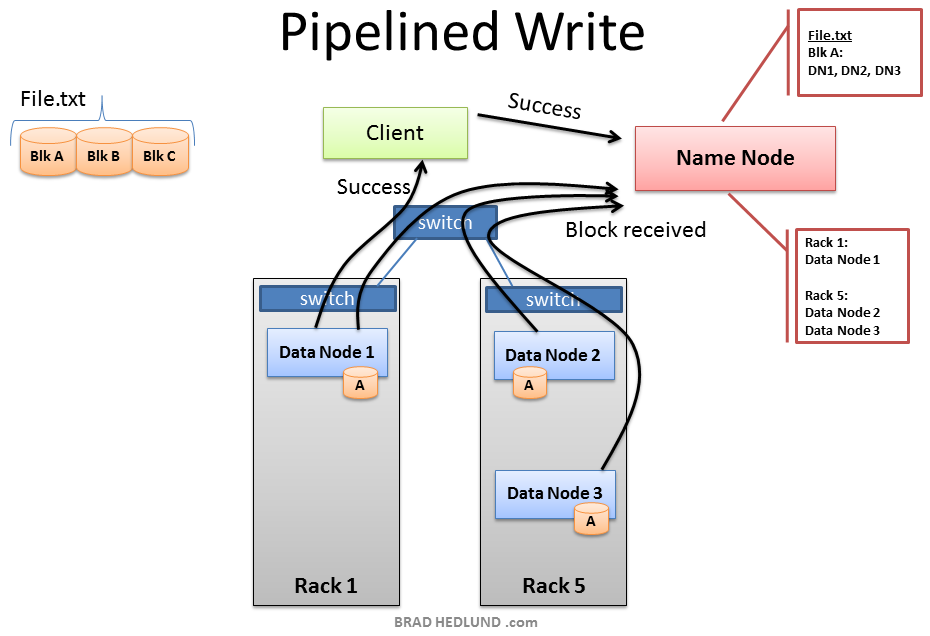
Before the Client writes “Block A” of File.txt to the cluster it wants to know that all Data Nodes which are expected to have a copy of this block are ready to receive it. It picks the first Data Node in the list for Block A (Data Node 1), opens a TCP 50010 connection and says, “Hey, get ready to receive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

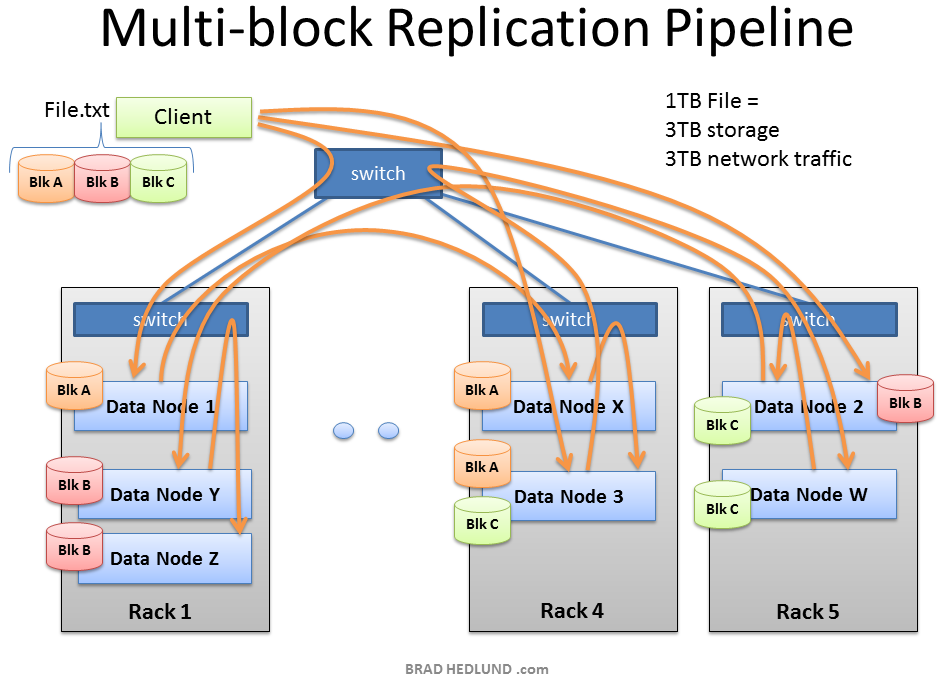


As data for each block is written into the cluster a replication pipeline is created between the (3) Data Nodes (or however many you have configured in dfs.replication). This means that as a Data Node is receiving block data it will at the same time push a copy of that data to the next Node in the pipeline.

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.

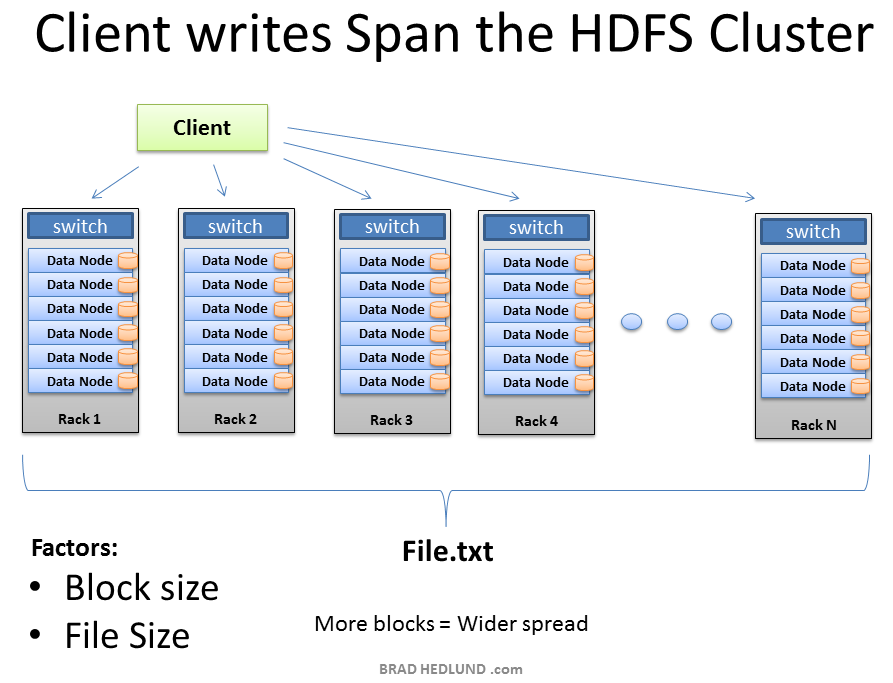


When all three Nodes have successfully received the block they will send a “Block Received” report to the Name Node. They will also send “Success” messages back up the pipeline and close down the TCP sessions. The Client receives a success message and tells the Name Node the block was successfully written. 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.



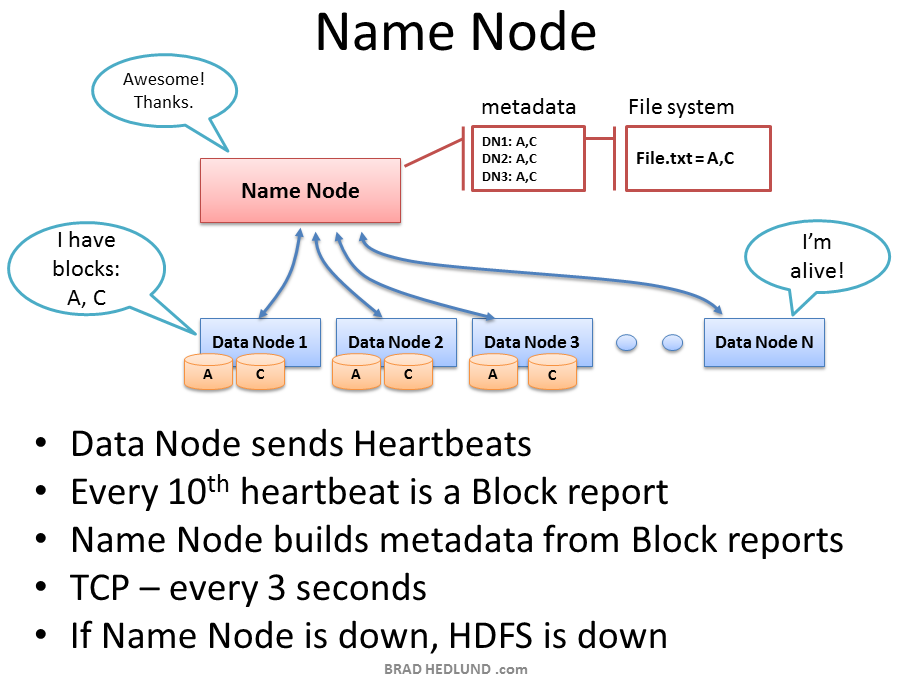
As the subsequent blocks of File.txt are written, the initial node in the pipeline will vary for each block, spreading around the hot spots of in-rack and cross-rack traffic for replication.

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.



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.

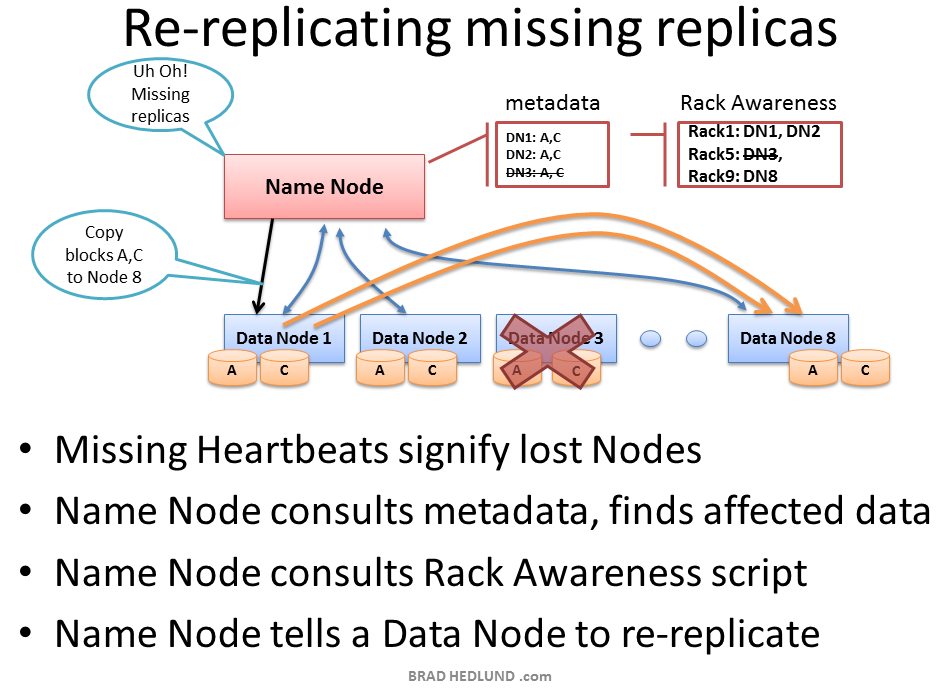
Another approach to scaling the cluster is to go **deep**. This is where you scale up the machines with more disk drives and more CPU cores. Instead of increasing the number of machines you begin to look at increasing the density of each machine. In scaling deep, you put yourself on a trajectory where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



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.

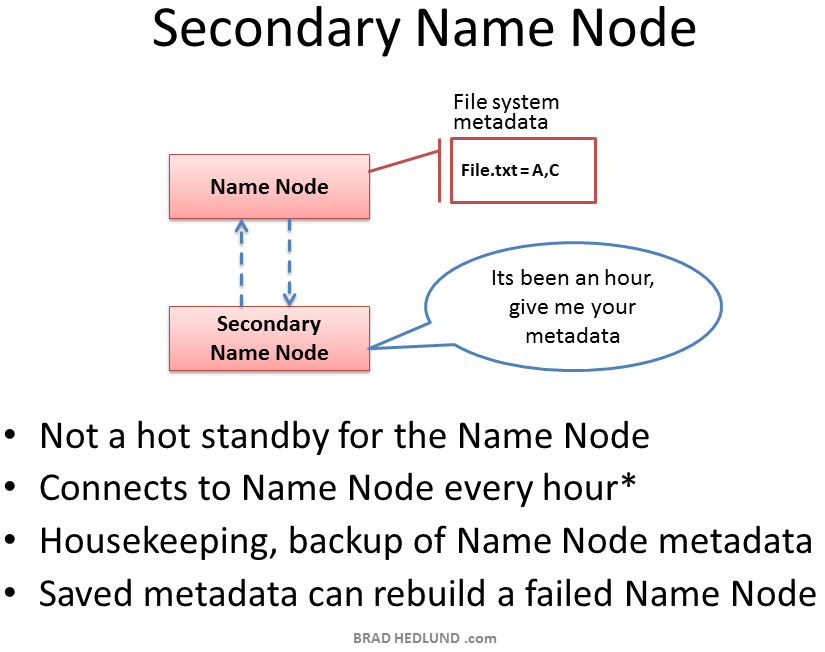
Data Nodes send heartbeats to the Name Node every 3 seconds via a TCP handshake, using the same port number defined for the Name Node daemon, usually TCP 9000. Every tenth heartbeat is a Block Report, where the Data Node tells the Name Node about all the blocks it has. The block reports allow the Name 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.



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.

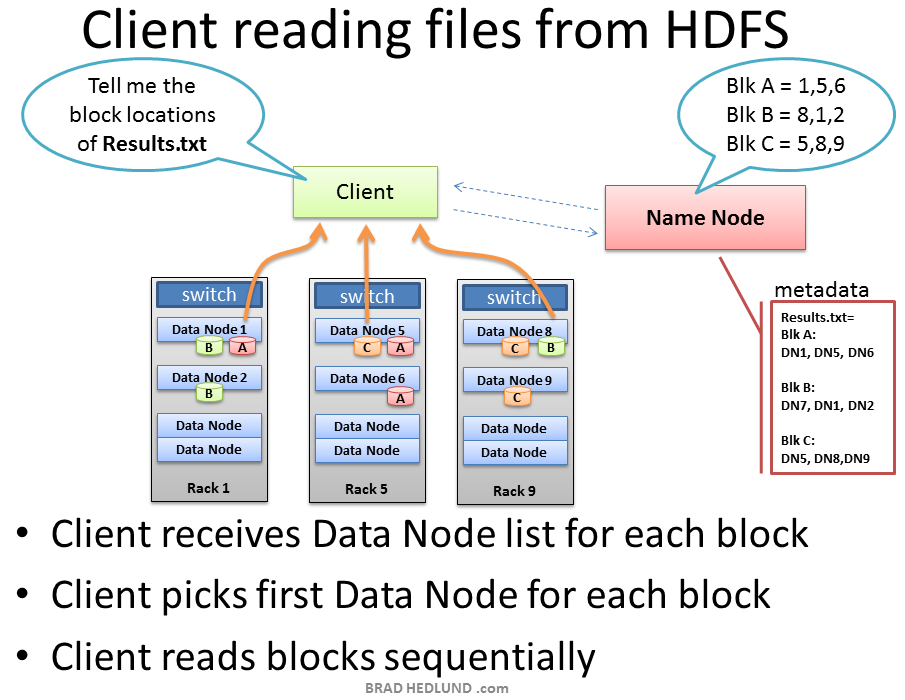
Consider the scenario where an entire rack of servers falls off the network, perhaps because of a rack switch failure, or power failure. The Name Node would begin instructing the remaining nodes in the cluster to re-replicate all of the data blocks lost in that rack. If each server in that rack had a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



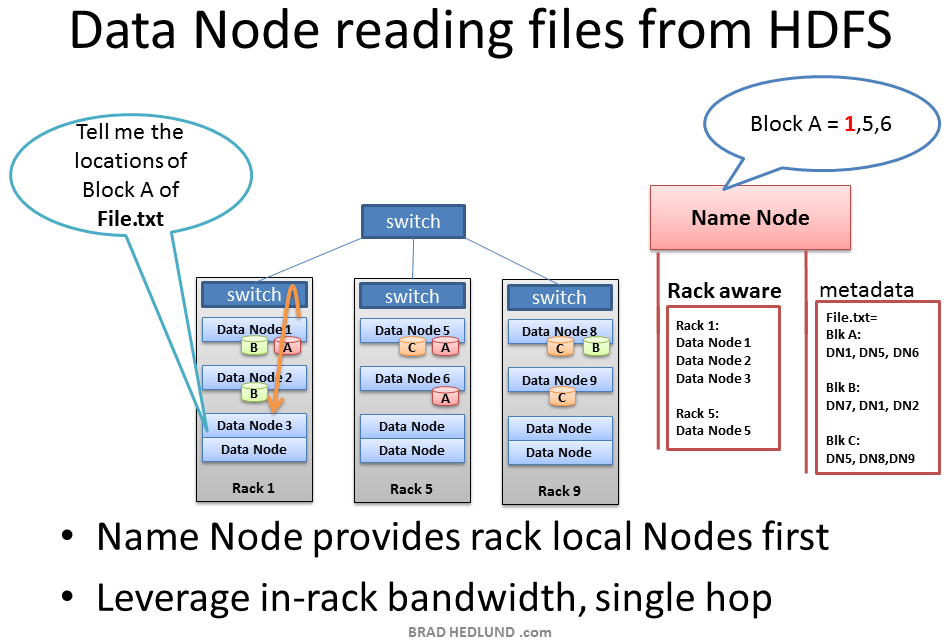
Hadoop has server role called the Secondary Name Node. A common misconception is that this role provides a high availability backup for the Name Node. This is not the case.

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.

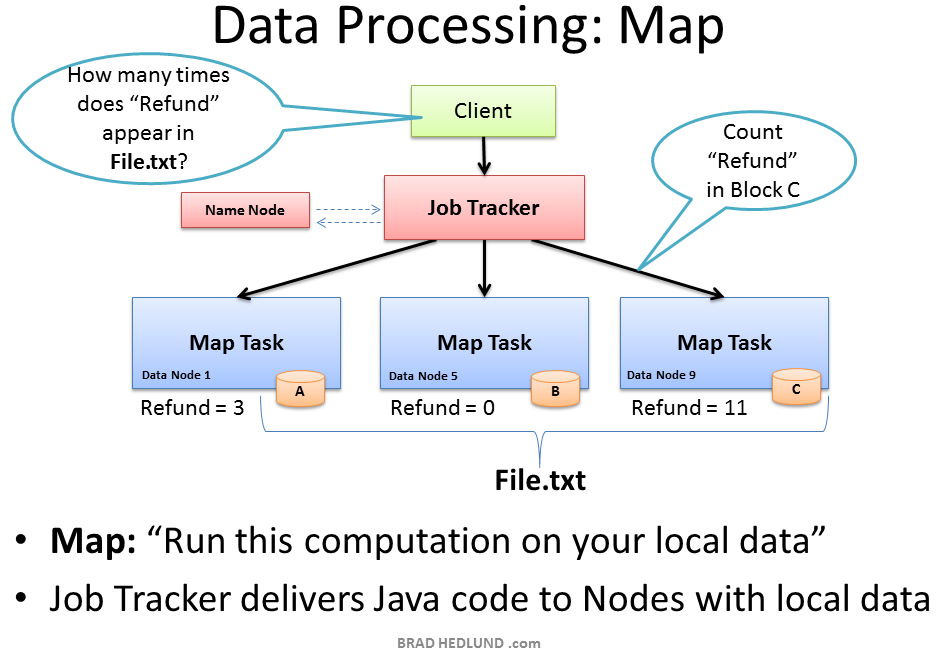


When a Client wants to retrieve a file from HDFS, perhaps the output of a job, it again consults the Name Node and asks for the block locations of the file. The Name Node returns a list of each Data Node holding a block, for each block. The Client picks a Data Node from each block list and reads one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



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.

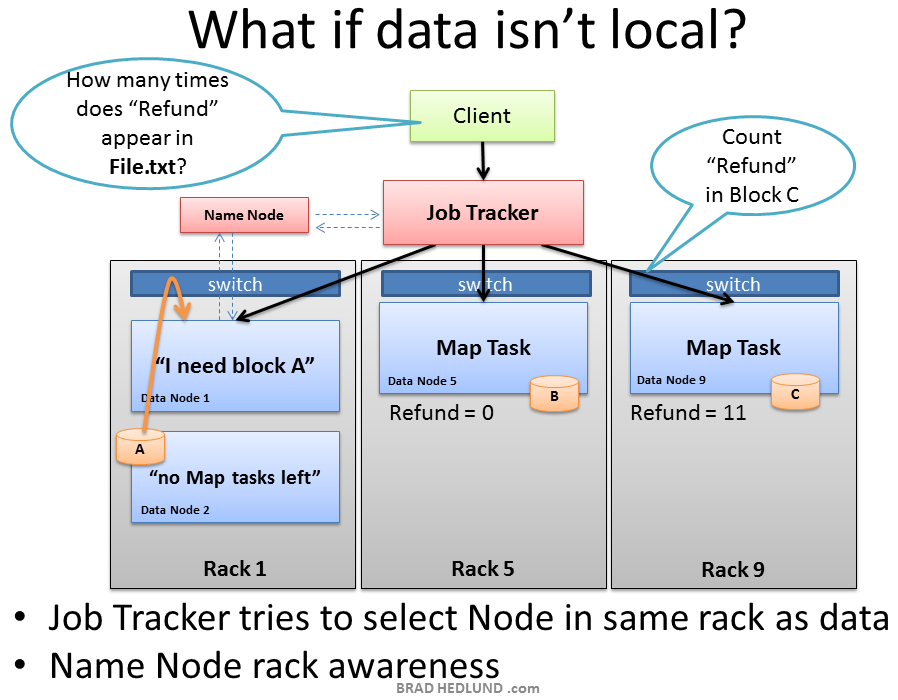


Now that File.txt is spread in small blocks across my cluster of machines I have the opportunity to provide extremely fast and efficient parallel processing of that data. The parallel processing framework included with Hadoop is called Map Reduce, named after two important steps in the model; **Map**,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

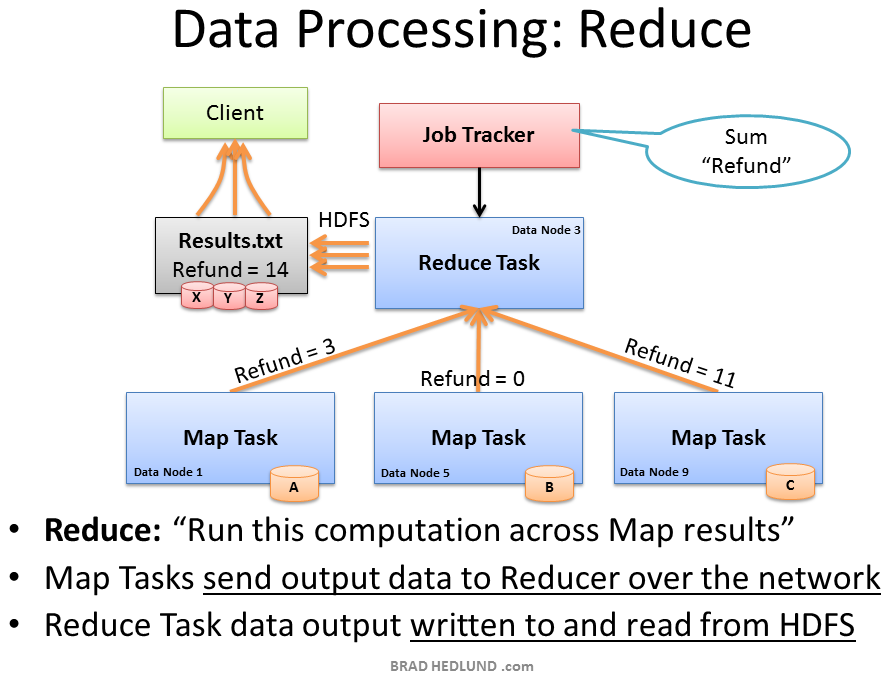
The first step is the Map process. This is where we simultaneously ask our machines to run a computation on their local block of data. In this case we are asking our machines to count the number of occurrences of the word “Refund” in the data blocks of File.txt.

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.

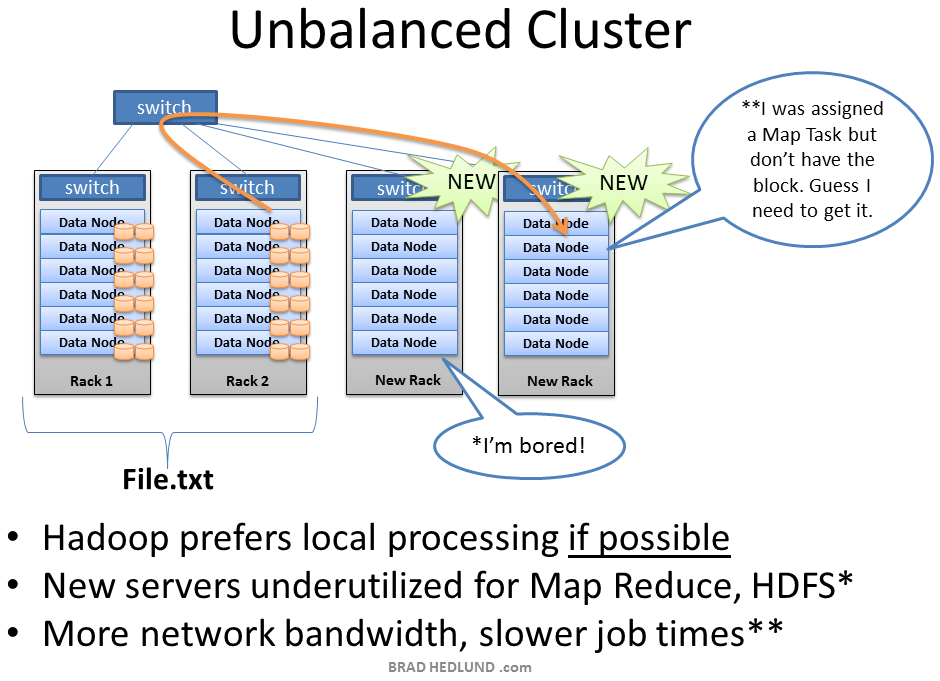
The Job Tracker starts a Reduce task on any one of the nodes in the cluster and instructs the Reduce task to go grab the intermediate data from all of the completed Map tasks. The Map tasks may respond to the Reducer almost simultaneously, resulting in a situation where you have a number of nodes sending TCP data to a single node, all at once. This traffic condition is often referred to as TCP Incast or “fan-in”. For networks handling lots of Incast conditions, it’s 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.

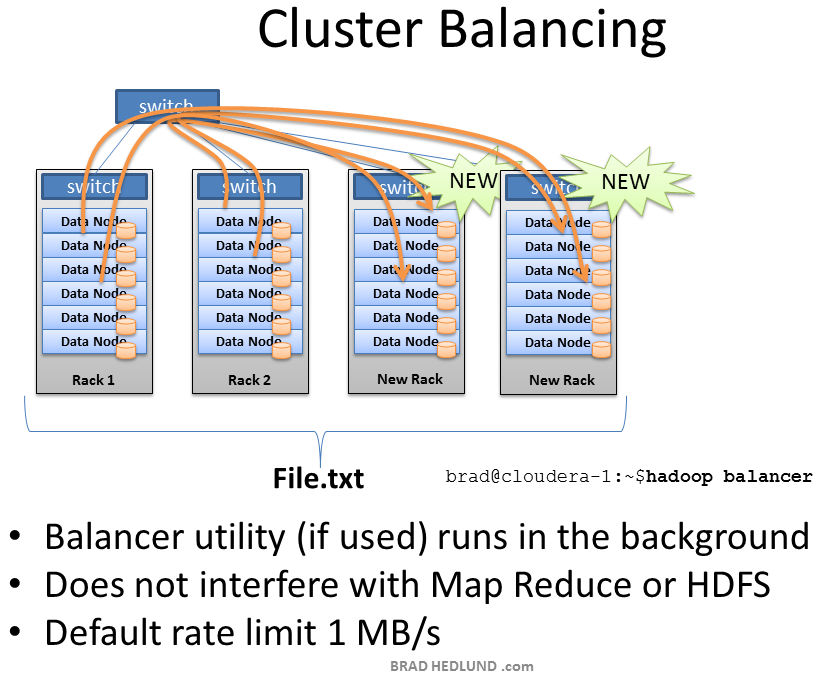
If you’re a studious network administrator, you would learn more about Map Reduce and the types of jobs your cluster will be running, and how the type of job affects the traffic flows on your network. If you’re a Hadoop networking rock star, you might even be able to suggest ways to better code the Map 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**

### DFSAdmin Command

The bin/hdfs dfsadmin command supports a few HDFS administration related operations. The bin/hdfs dfsadmin -help command lists all the commands currently supported. For e.g.:

* -report: reports basic statistics of HDFS. Some of this information is also available on the NameNode front page.
* -safemode: though usually not required, an administrator can manually enter or leave Safemode.
* -finalizeUpgrade: removes previous backup of the cluster made during last upgrade.
* -refreshNodes: Updates the namenode with the set of datanodes allowed to connect to the namenode. Namenodes re-read datanode hostnames in the file defined by dfs.hosts, dfs.hosts.exclude Hosts defined in dfs.hosts are the datanodes that are part of the cluster. If there are entries in dfs.hosts, only the hosts in it are allowed to register with the namenode. Entries in dfs.hosts.exclude are datanodes that need to be decommissioned. Datanodes complete decommissioning when all the replicas from them are replicated to other datanodes. Decommissioned nodes are not automatically shutdown and are not chosen for writing for new replicas.
* -printTopology : Print the topology of the cluster. Display a tree of racks and datanodes attached to the tracks as viewed by the NameNode.

For command usage, see dfsadmin.

## Secondary NameNode

The NameNode stores modifications to the file system as a log appended to a native file system file, edits. When a NameNode starts up, it reads HDFS state from an image file, fsimage, and then applies edits from the edits log file. It then writes new HDFS state to the fsimage and starts normal operation with an empty edits file. Since NameNode merges fsimage and edits files only during start up, the edits log file could get very large over time on a busy cluster. Another side effect of a larger edits file is that next restart of NameNode takes longer.

The secondary NameNode merges the fsimage and the edits log files periodically and keeps edits log size within a limit. It is usually run on a different machine than the primary NameNode since its memory requirements are on the same order as the primary NameNode.

The start of the checkpoint process on the secondary NameNode is controlled by two configuration parameters.

* dfs.namenode.checkpoint.period, set to 1 hour by default, specifies the maximum delay between two consecutive checkpoints, and
* dfs.namenode.checkpoint.txns, set to 1 million by default, defines the number of uncheckpointed transactions on the NameNode which will force an urgent checkpoint, even if the checkpoint period has not been reached.

The secondary NameNode stores the latest checkpoint in a directory which is structured the same way as the primary NameNode’s directory. So that the check pointed image is always ready to be read by the primary NameNode if necessary.

For command usage, see secondarynamenode.

## Checkpoint Node

NameNode persists its namespace using two files: fsimage, which is the latest checkpoint of the namespace and edits, a journal (log) of changes to the namespace since the checkpoint. When a NameNode starts up, it merges the fsimage and edits journal to provide an up-to-date view of the file system metadata. The NameNode then overwrites fsimage with the new HDFS state and begins a new edits journal.

The Checkpoint node periodically creates checkpoints of the namespace. It downloads fsimage and edits from the active NameNode, merges them locally, and uploads the new image back to the active NameNode. The Checkpoint node usually runs on a different machine than the NameNode since its memory requirements are on the same order as the NameNode. The Checkpoint node is started by bin/hdfs namenode -checkpoint on the node specified in the configuration file.

The location of the Checkpoint (or Backup) node and its accompanying web interface are configured via the dfs.namenode.backup.address and dfs.namenode.backup.http-address configuration variables.

The start of the checkpoint process on the Checkpoint node is controlled by two configuration parameters.

* dfs.namenode.checkpoint.period, set to 1 hour by default, specifies the maximum delay between two consecutive checkpoints
* dfs.namenode.checkpoint.txns, set to 1 million by default, defines the number of uncheckpointed transactions on the NameNode which will force an urgent checkpoint, even if the checkpoint period has not been reached.

The Checkpoint node stores the latest checkpoint in a directory that is structured the same as the NameNode’s directory. This allows the checkpointed image to be always available for reading by the NameNode if necessary. See Import checkpoint.

Multiple checkpoint nodes may be specified in the cluster configuration file.

For command usage, see namenode.

## Backup Node

The Backup node provides the same checkpointing functionality as the Checkpoint node, as well as maintaining an in-memory, up-to-date copy of the file system namespace that is always synchronized with the active NameNode state. Along with accepting a journal stream of file system edits from the NameNode and persisting this to disk, the Backup node also applies those edits into its own copy of the namespace in memory, thus creating a backup of the namespace.

The Backup node does not need to download fsimage and edits files from the active NameNode in order to create a checkpoint, as would be required with a Checkpoint node or Secondary NameNode, since it already has an up-to-date state of the namespace state in memory. The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local fsimage file and reset edits.

As the Backup node maintains a copy of the namespace in memory, its RAM requirements are the same as the NameNode.

The NameNode supports one Backup node at a time. No Checkpoint nodes may be registered if a Backup node is in use. Using multiple Backup nodes concurrently will be supported in the future.

The Backup node is configured in the same manner as the Checkpoint node. It is started with bin/hdfs namenode -backup.

The location of the Backup (or Checkpoint) node and its accompanying web interface are configured via the dfs.namenode.backup.address and dfs.namenode.backup.http-address configuration variables.

Use of a Backup node provides the option of running the NameNode with no persistent storage, delegating all responsibility for persisting the state of the namespace to the Backup node. To do this, start the NameNode with the -importCheckpoint option, along with specifying no persistent storage directories of type edits dfs.namenode.edits.dir for the NameNode configuration.

For a complete discussion of the motivation behind the creation of the Backup node and Checkpoint node, see HADOOP-4539. For command usage, see namenode.

## Import Checkpoint

The latest checkpoint can be imported to the NameNode if all other copies of the image and the edits files are lost. In order to do that one should:

* Create an empty directory specified in the dfs.namenode.name.dir configuration variable;
* Specify the location of the checkpoint directory in the configuration variable dfs.namenode.checkpoint.dir;
* and start the NameNode with -importCheckpoint option.

The NameNode will upload the checkpoint from the dfs.namenode.checkpoint.dir directory and then save it to the NameNode directory(s) set in dfs.namenode.name.dir. The NameNode will fail if a legal image is contained in dfs.namenode.name.dir. The NameNode verifies that the image in dfs.namenode.checkpoint.dir is consistent, but does not modify it in any way.

For command usage, see namenode.

## Balancer

HDFS data might not always be be placed uniformly across the DataNode. One common reason is addition of new DataNodes to an existing cluster. While placing new blocks (data for a file is stored as a series of blocks), NameNode considers various parameters before choosing the DataNodes to receive these blocks. Some of the considerations are:

* Policy to keep one of the replicas of a block on the same node as the node that is writing the block.
* Need to spread different replicas of a block across the racks so that cluster can survive loss of whole rack.
* One of the replicas is usually placed on the same rack as the node writing to the file so that cross-rack network I/O is reduced.
* Spread HDFS data uniformly across the DataNodes in the cluster.

Due to multiple competing considerations, data might not be uniformly placed across the DataNodes. HDFS provides a tool for administrators that analyzes block placement and rebalanaces data across the DataNode. A brief administrator’s guide for balancer is available at HADOOP-1652.

For command usage, see balancer.

## Rack Awareness

Typically large Hadoop clusters are arranged in racks and network traffic between different nodes with in the same rack is much more desirable than network traffic across the racks. In addition NameNode tries to place replicas of block on multiple racks for improved fault tolerance. Hadoop lets the cluster administrators decide which rack a node belongs to through configuration variable net.topology.script.file.name. When this script is configured, each node runs the script to determine its rack id. A default installation assumes all the nodes belong to the same rack. This feature and configuration is further described in PDF attached to HADOOP-692.

## Safemode

During start up the NameNode loads the file system state from the fsimage and the edits log file. It then waits for DataNodes to report their blocks so that it does not prematurely start replicating the blocks though enough replicas already exist in the cluster. During this time NameNode stays in Safemode. Safemode for the NameNode is essentially a read-only mode for the HDFS cluster, where it does not allow any modifications to file system or blocks. Normally the NameNode leaves Safemode automatically after the DataNodes have reported that most file system blocks are available. If required, HDFS could be placed in Safemode explicitly using bin/hdfs dfsadmin -safemode command. NameNode front page shows whether Safemode is on or off. A more detailed description and configuration is maintained as JavaDoc for setSafeMode().

## fsck

HDFS supports the fsck command to check for various inconsistencies. It it is designed for reporting problems with various files, for example, missing blocks for a file or under-replicated blocks. Unlike a traditional fsck utility for native file systems, this command does not correct the errors it detects. Normally NameNode automatically corrects most of the recoverable failures. By default fsck ignores open files but provides an option to select all files during reporting. The HDFS fsck command is not a Hadoop shell command. It can be run as bin/hdfs fsck. For command usage, see fsck. fsck can be run on the whole file system or on a subset of files.

## fetchdt

HDFS supports the fetchdt command to fetch Delegation Token and store it in a file on the local system. This token can be later used to access secure server (NameNode for example) from a non secure client. Utility uses either RPC or HTTPS (over Kerberos) to get the token, and thus requires kerberos tickets to be present before the run (run kinit to get the tickets). The HDFS fetchdt command is not a Hadoop shell command. It can be run as bin/hdfs fetchdt DTfile. After you got the token you can run an HDFS command without having Kerberos tickets, by pointing HADOOP\_TOKEN\_FILE\_LOCATION environmental variable to the delegation token file. For command usage, see fetchdt command.

## Recovery Mode

Typically, you will configure multiple metadata storage locations. Then, if one storage location is corrupt, you can read the metadata from one of the other storage locations.

However, what can you do if the only storage locations available are corrupt? In this case, there is a special NameNode startup mode called Recovery mode that may allow you to recover most of your data.

You can start the NameNode in recovery mode like so: namenode -recover

When in recovery mode, the NameNode will interactively prompt you at the command line about possible courses of action you can take to recover your data.

If you don’t want to be prompted, you can give the -force option. This option will force recovery mode to always select the first choice. Normally, this will be the most reasonable choice.

Because Recovery mode can cause you to lose data, you should always back up your edit log and fsimage before using it.

## Upgrade and Rollback

When Hadoop is upgraded on an existing cluster, as with any software upgrade, it is possible there are new bugs or incompatible changes that affect existing applications and were not discovered earlier. In any non-trivial HDFS installation, it is not an option to loose any data, let alone to restart HDFS from scratch. HDFS allows administrators to go back to earlier version of Hadoop and rollback the cluster to the state it was in before the upgrade. HDFS upgrade is described in more detail in Hadoop Upgrade Wiki page. HDFS can have one such backup at a time. Before upgrading, administrators need to remove existing backup using bin/hadoop dfsadmin -finalizeUpgrade command. The following briefly describes the typical upgrade procedure:

* Before upgrading Hadoop software, finalize if there an existing backup. dfsadmin -upgradeProgress status can tell if the cluster needs to be finalized.
* Stop the cluster and distribute new version of Hadoop.
* Run the new version with -upgrade option (bin/start-dfs.sh -upgrade).
* Most of the time, cluster works just fine. Once the new HDFS is considered working well (may be after a few days of operation), finalize the upgrade. Note that until the cluster is finalized, deleting the files that existed before the upgrade does not free up real disk space on the DataNodes.
* If there is a need to move back to the old version,
  + stop the cluster and distribute earlier version of Hadoop.
  + run the rollback command on the namenode (bin/hdfs namenode -rollback).
  + start the cluster with rollback option. (sbin/start-dfs.sh -rollback).

When upgrading to a new version of HDFS, it is necessary to rename or delete any paths that are reserved in the new version of HDFS. If the NameNode encounters a reserved path during upgrade, it will print an error like the following:

/.reserved is a reserved path and .snapshot is a reserved path component in this version of HDFS. Please rollback and delete or rename this path, or upgrade with the -renameReserved [key-value pairs] option to automatically rename these paths during upgrade.

Specifying -upgrade -renameReserved [optional key-value pairs] causes the NameNode to automatically rename any reserved paths found during startup. For example, to rename all paths named .snapshot to .my-snapshot and .reserved to .my-reserved, a user would specify -upgrade -renameReserved .snapshot=.my-snapshot,.reserved=.my-reserved.

If no key-value pairs are specified with -renameReserved, the NameNode will then suffix reserved paths with .<LAYOUT-VERSION>.UPGRADE\_RENAMED, e.g. .snapshot.-51.UPGRADE\_RENAMED.

There are some caveats to this renaming process. It’s recommended, if possible, to first hdfs dfsadmin -saveNamespace before upgrading. This is because data inconsistency can result if an edit log operation refers to the destination of an automatically renamed file.

## DataNode Hot Swap Drive

Datanode supports hot swappable drives. The user can add or replace HDFS data volumes without shutting down the DataNode. The following briefly describes the typical hot swapping drive procedure:

* If there are new storage directories, the user should format them and mount them appropriately.
* The user updates the DataNode configuration dfs.datanode.data.dir to reflect the data volume directories that will be actively in use.
* The user runs dfsadmin -reconfig datanode HOST:PORT start to start the reconfiguration process. The user can use dfsadmin -reconfig datanode HOST:PORT status to query the running status of the reconfiguration task.
* Once the reconfiguration task has completed, the user can safely umount the removed data volume directories and physically remove the disks.

## File Permissions and Security

The file permissions are designed to be similar to file permissions on other familiar platforms like Linux. Currently, security is limited to simple file permissions. The user that starts NameNode is treated as the superuser for HDFS. Future versions of HDFS will support network authentication protocols like Kerberos for user authentication and encryption of data transfers. The details are discussed in the Permissions Guide.

# Secondary Namenode - What it really do?

Dec 31, 2013

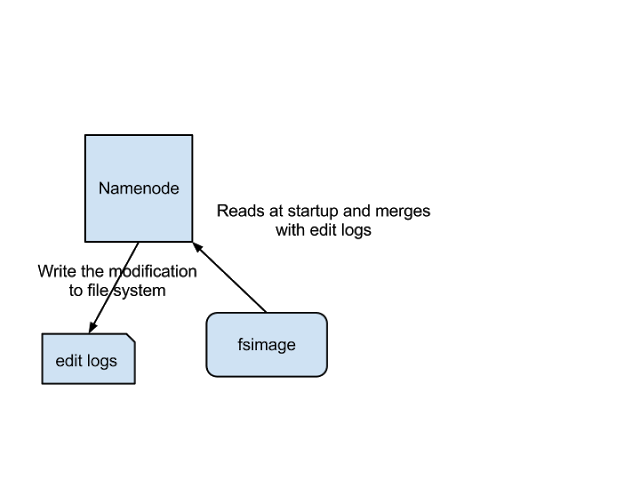
hadoop

Secondary Namenode is one of the poorly named component in Hadoop. By its name, it gives a sense that its a backup for the Namenode.But in reality its not. Lot of beginners in Hadoop get confused about what exactly SecondaryNamenode does and why its present in HDFS.So in this blog post I try to explain the role of secondary namenode in HDFS.

By its name, you may assume that it has something to do with Namenode and you are right. So before we dig into Secondary Namenode lets see what exactly Namenode does.

### Namenode

Namenode holds the meta data for the HDFS like Namespace information, block information etc. When in use, all this information is stored in main memory. But these information also stored in disk for persistence storage.



The above image shows how Name Node stores information in disk.  
Two different files are

1. fsimage - Its the snapshot of the filesystem when namenode started
2. Edit logs - Its the sequence of changes made to the filesystem after namenode started

Only in the restart of namenode , edit logs are applied to fsimage to get the latest snapshot of the file system. But namenode restart are rare in production clusters which means edit logs can grow very large for the clusters where namenode runs for a long period of time. The following issues we will encounter in this situation.

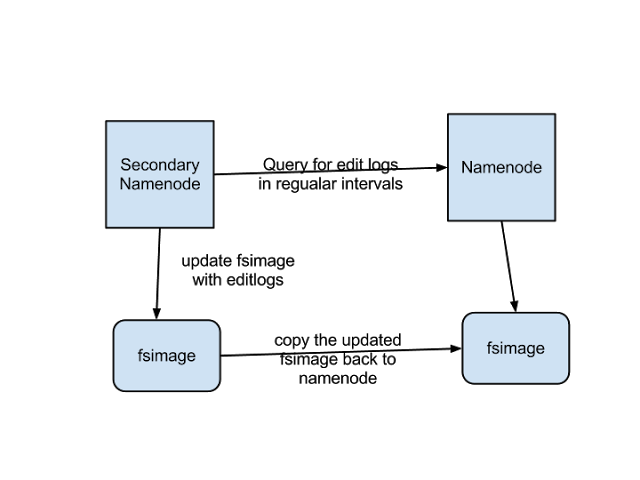
1. Editlog become very large , which will be challenging to manage it
2. Namenode restart takes long time because lot of changes has to be merged
3. In the case of crash, we will lost huge amount of metadata since fsimage is very old

So to overcome this issues we need a mechanism which will help us reduce the edit log size which is manageable and have up to date fsimage ,so that load on namenode reduces . It’s very similar to Windows Restore point, which will allow us to take snapshot of the OS so that if something goes wrong , we can fallback to the last restore point.

So now we understood NameNode functionality and challenges to keep the meta data up to date.So what is this all have to with Seconadary Namenode?

### Secondary Namenode

Secondary Namenode helps to overcome the above issues by taking over responsibility of merging editlogs with fsimage from the namenode.



The above figure shows the working of Secondary Namenode

1. It gets the edit logs from the namenode in regular intervals and applies to fsimage
2. Once it has new fsimage, it copies back to namenode
3. Namenode will use this fsimage for the next restart,which will reduce the startup time

Secondary Namenode whole purpose is to have a checkpoint in HDFS. Its just a helper node for namenode.That’s why it also known as checkpoint node inside the community.

So we now understood all Secondary Namenode does puts a checkpoint in filesystem which will help Namenode to function better. Its not the replacement or backup for the Namenode. So from now on make a habit of calling it as a checkpoint node.