Rebalance an HDFS Cluster

Introduction/Objective

This document describes a design for redistributing data blocks when an HDFS cluster becomes imbalanced.

Motivation

A HDFS cluster can easily become imbalanced, for example, when a new data node joins the cluster. Because it does not hold much data, any map task assigned to the machine most likely does not access local data, thus increasing the use of network bandwidth. On the other hand, when some data nodes become full, new data blocks are placed on only non-full data nodes, thus reducing their read parallelism. It is important to redistribute data blocks when imbalance occurs.

Requirements

1. Rebalancing maintains data availability guarantees in the sense that it does not cause any block loss, change the number of replicas that a block has, or reduce the number of racks that a block resides.
2. An administrator should be able to invoke and interrupt rebalancing from a command line.
3. Block moving should be throttled so that it does not saturate the network.
4. Rebalancing should not cause a name node to be too busy to serve any incoming request.

Architecture Overview

1. What's balance?

A cluster is balanced if there is no under-utilized or above-utilized data node in the cluster. A data node's utilization is defined as the percentage of its used space.

An under-utilized data node is a node whose utilization is less than (average utilization – threshold).

An over-utilized data node is a node whose utilization is greater than (average utilization +threshold).

A threshold is user configurable. It can be set or changed from a command line. A default value is a utilization of 10%.

2. When to rebalance?

An administrator issues a command to trigger rebalancing. Rebalancing automatically shuts off when the cluster is balanced or when it does not make any progress. The command is defined as below:

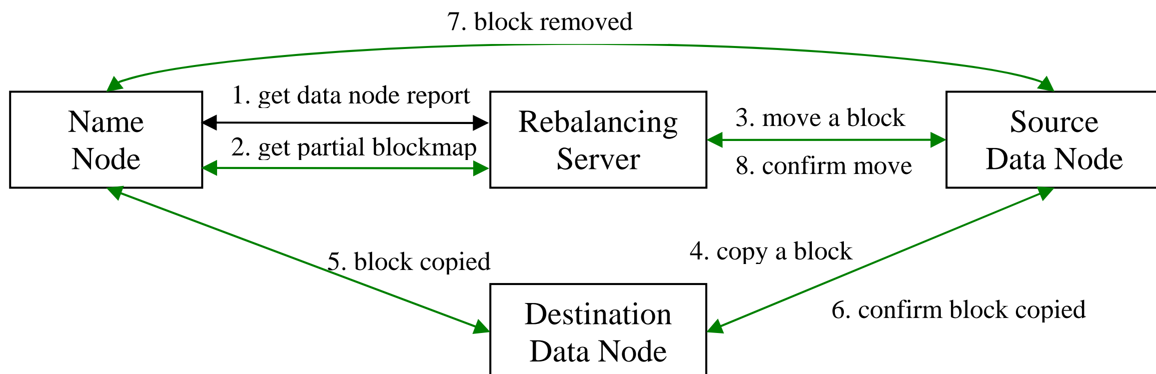
Hadoop dfsadmin rebalance [-t <threshold>*] <start>*

Start rebalancing and change the default threshold if the option –t is set.

An administrator can simply press Ctrl-C to stop the rebalancing process.

3. How to rebalance?

To prevent the rebalancing from making the name node too busy to serve incoming requests, all the rebalancing decisions are made at a separate process, a rebalancing server at the machine where the command is issued. This implies that a rebalancing command should run at a machine that has fast connection to the DFS cluster.



Picture 1: Interactions among rebalancing server, name node, and data nodes

The rebalancing server makes rebalancing decision iteration by iteration. At each iteration the major goal is to make every over/under-utilized data node less imbalanced rather than reducing the number of imbalanced data nodes. In this way rebalancing is able to maximize the use of the network bandwidth. Picture 1 shows the interactions among a rebalancing server, a name node, and data nodes. Green lines represent new protocols for rebalancing. The numbers show the communication order. The rebalancing server first asks the name node for a data node report (step 1). After choosing all source and destination data nodes, it asks the name node for each source's partial blocks map (step 2). It then decides the blocks to move from sources to destinations. For each block, it instructs the block's source to move it to its destination (step 3) and the source instructs the destination to copy the block (step 4). After the copy is done, the destination notifies

the name node (step 5) and the source (step 6). The source deletes the block and informs the name node (step 7) and the rebalancing server (step 8) of the removal.

Architecture Detail

Rebalancing Server Design

1. A rebalancing command starts a Rebalancing Server at the user's machine.
2. It asks the name node for a data node report.
   - construct the network topology and calculate average utilization
   - construct an over-utilized data node list, an under-utilized data node list, an above-average utilized data node list, and a below-average utilized data node list
   - print the utilization variance on the screen as a progress report.
   - exit if the cluster is balanced or no progress has been made for three iterations; otherwise do the following:
   - go through the over/under-utilized data node list and make rebalancing decisions
     a. choose source data nodes & destination data nodes. The selection criteria are:
     - For an under-utilized data node, source data nodes are chosen randomly from data nodes in the following priority:
       (a). Over-utilized nodes in the same rack
       (b). Above-average-utilized nodes in the same rack
       (c). Over-utilized nodes in the other racks
       (d). Above-average-utilized nodes in the other racks
     - For an over-utilized data node, destination data nodes are randomly chosen from data nodes in the following priority:
       (a). Under-utilized nodes in the same rack
       (b). Below-average-utilized data nodes in the same rack
       (c). Under-utilized nodes in other racks
       (d). Below-average-utilized nodes in the other racks

     b. decide the number of bytes (not the blocks) to move from a source to a destination.
     - if the source is over-utilized, the size is equal to size of the threshold; if the source is above-average-utilized, the size is equal to (its capacity-average capacity)
     - if the destination is under-utilized, the size is equal to the threshold size; if the destination is below-average-utilized, the size is equal to (average capacity-its capacity)
3. The rebalancing server asks the name node for a partial blocks map for each source. Each request is in the form of <source, 1.5*size>.
4. The name node randomly chooses a certain number of blocks so that the total size of the chosen blocks is equal to the requested size. It sends all blocks back to the rebalancing server.
5. The rebalancing server keeps on choosing blocks to move for each source data node until the total number of chosen blocks is equal to the size decided previously or all the received blocks have been examined.

a. The block selection should not violate requirement 1. The selection criteria is that:
    - The block should not have other replicas on the rack that the destination resides if the source & the destination are on different racks;
    - The destination should not hold a replica of the block;
    - Disallow more than one replicas of the block to move concurrently;

b. Block move tasks are put to a pendingMove queue and moving requests are sent to source data nodes.

c. The pendingMove queue is indexed by sources, destinations, and blocks. It makes sure that both a source and a destination's queue size is at most 5, thus limiting the number of concurrent block transfers is 5 per node. It also makes sure that at any given time, only one replica of a block is transferring.

d. A task is removed from pendingMove when it receives a confirmation from the source or when the timer is expired.

6. When all blocks are chosen and the pendingMove queue becomes empty, repeat step 2.

Data Node Support for Rebalancing

1. When a data node receives an OP_MOVE command, it sends an OP_COPY command to transfer the block to the destination & removes the block from itself after the copy is confirmed.
2. After the destination data nodes copied the block, it informs the name node that a block is copied. The name node updates its block map and marks the block that any of its excessive replicas should not be removed. After the source data node removes the block, it sends a confirmation back to the name node. The name node then removes the source from its blocks map, removes the mark, and triggers deletion if the block has excessive replicas. If no deletion confirmation is received after a timeout period, the name node clears the mark.
3. Each data node has a limited bandwidth for rebalancing. The default value for the bandwidth is 5MB/s.
4. Throttling is done at both source & destination sides. Each data node limits maximum number of concurrent data transfers (including both sending and receiving) for the rebalancing purpose to be 5. In the worst case, each data transfer has a limited bandwidth of 1MB/s.
5. Each sender & receiver has a Throttler. The primary method of the class is throttle(int numOfBytes). The parameter numOfBytes indicates the total number of bytes that the caller has sent or received since the last throttle is called. The method calculates the caller's I/O rate. If the rate is faster than the bandwidth limit, it sleeps to slow down the data transfer. After it wakes up, it adjusts its bandwidth limit if the number of concurrent data transfers is changed. Throttle() is called periodically whenever a chunk is sent or received.

Rebalancing protocols

All new protocols to support rebalancing are marked as green in picture 1.

1. Name node protocol

A rebalancing server shares part of the name node work load, acting as a secondary name node. So a new interface, NamenodeProtocol, is created to support any secondary name node's communications with the name node.

```
/*****************************************************************************************
 * Protocol that a secondary NameNode uses to communicate with the NameNode.
 * It's used to get part of the name node state
 *****************************************************************************************/
interface NamenodeProtocol extends VersionedProtocol {
  public static final long versionID = 0L;

  /** Get a list of blocks belonged to <code>datanode</code>
    * whose total size is equal to <code>size</code>
   * @param datanode   a data node
   * @param size       requested size
   * @return           a list of blocks
   */
  public LocatedBlocks getBlocks(DatanodeInfo datanode, long size)
  throws IOException;                                          // Step 2 in Picture 1
}
```

2. Changes to DatanodeProtocol

```
  /**
   * Allows a DataNode to tell the NameNode about blocks recently moved to this datanode. If
   * disableRemove is set, namenode marks that any excessive replicas of this block is not
   * allowed to be deleted.
   */
  public void blockCopied(DatanodeRegistration registration, Block blocks[],
                          boolean disableRemove) throws IOException;      // Step 5 in Picture 1
  /**
   * Allows a DataNode to tell the NameNode about recently-removed blocks.
   * If enableRemove is set, namenode removes the disableRemove mark on the block and
   * handles excessive replicas if there are any.
   */
  public void blockRemoved(DatanodeRegistration registration, Block blocks[],
                          Boolean enableRemove) throws IOException;       // Step 7 in Picture 1
```

3. Communications with Data nodes

Two rebalancing related operations will be added to communication protocols that define communications to data nodes. OP_MOVE is to send to a source that requests the source to move a block to another data node, while OP_COPY is to send to a destination that requests the destination to receive a block. Data nodes also send reply when operations are finished. The message formats are defined below:

OP_MOVE request:	OP_MOVE	BlkID (long)	DstNode (DatanodeInfo)	//step 3 in Picture 1
OP_COPY request:	OP_COPY	BlkID (long)	data	//step 4 in Picture 1
OP reply:	Operation status (short)			//steps 6&8 in Picture 1

Table 1 Formats of Rebalancing Messages sent to Data Nodes

Discussion of possible race conditions

Race Conditions	**How to handle it**
Source data node does not have the block when OP_MOVE is received	Return OP_FAILURE to the rebalancing server.
Source data node receives an invalidate block command while waiting for the reply from OP_COPY	Reject the Invalidate command
Source data node receives an OP_READ command while waiting for the reply from OP_COPY	Allow read. But need to wait to delete the block until read is done.
Destination data node already has the block when OP_COPY is received.	Return OP_FAILURE to the source and source aborts the move.
Destination data node receives an OP_WRITE or another OP_COPY when OP_COPY is in progress.	Reject the write command/the late coming OP_WRITE.
Source sends its block report after it receives confirmation from destination but before the block is removed from its own disk	Not a problem. The block report is processed before the block removed confirmation.
Destination sends its block report right before it copies the block to its disk	Not a problem. The block report is processed before the block copied confirmation.
Rebalancing server's request to name node failed	The rebalancing server exits
destination's confirmation to name node failed	Abort block move
Source's confirmation to name node failed	Continue
Any request to data node failed	Abort the block move

Table 2 Race Conditions and their solutions

Test Plan

A rebalancing server moves blocks around in a DFS cluster in order to rebalance it. All tests should make sure that:
REQ1. Rebalancing does not cause the loss of a block;
REQ2. Rebalancing does not change the number of replicas that a block had;
REQ3. Rebalancing does not decrease the number of racks that a block had.
Besides the algorithm should try best to satisfy the following but provides no guarantee.
REQ4. Rebalancing process makes the cluster to be less and less imbalanced.

Id	Test Case	Expected Behavior
1	Bring up a one-node dfs cluster. Set files' replication factor to be 1 and fill up the node to 50% full. Then add an empty data node.	REQ1, 4 (1 implies 2 and 3)
2	Bring up a dfs cluster with nodes A and B. Set files' replication factor to be 2 and fill up the cluster to 50% full. Then add an empty data node C. All three nodes are on the same rack.	REQ1, 2, 4 (2 implies 3)
3	The same as test case 2 except that A, B, and C are on different racks.	REQ1, 2, 3, 4
4	a. The same as test case 3 except that interrupt rebalancing.	REQ1, 2, 3
	b. Restart rebalancing until it is done.	REQ1, 2, 3, 4
5	The same as test case 3 except that shut down namenode while rebalancing.	REQ1, 2, 3
6	The same as test case 3 except that writing while rebalancing.	REQ1, 2, 3
7	The same as test case 3 except that deleting while rebalancing	REQ1, 2, 3
8	The same as test case 3 except that writing & deleting while rebalancing	REQ1, 2, 3
9	Bring up a 3-node cluster. All nodes are on different racks. Set Files' replication factor to be 2. Fill 2 nodes to its 20% capacity and fill the other to its 50% capacity.	REQ1, 2, 3, 4
10	The same as test case 9 except that fill 1 node to its 30% capacity, 1 node to 50%, the other to 80%.	REQ1, 2, 3, 4
11	Bring up a dfs cluster with 3 nodes, 2 of which are on rack A and another node is on rack B. Set replication factor to be 2 and fill up the cluster to its 20% capacity and add one empty node to rack A.	No block is moved.
12	The same as test case 10 except that the new node is added to rack B	REQ1, 2, 3, 4
13	Bring up a 4-node cluster. Nodes A, B are on rack 1 and Nodes C, D are on rack 2. Set replication factor to be 3 and fill Node A to 60%, B to 30%, C to 50%, and D to 20%.	REQ1, 2, 3, 4
14	Scalability test: populate a 800-node cluster a. Run rebalancing after one node is added. b. Run rebalancing after 40 nodes are added.	REQ1, 2, 3, 4

Table 3 Test Cases and their expected behaviors

Above tests are run with a threshold of 10%. Test case 3 is automatic but all the other cases are manual tests.