Scaling Namespaces and Optimizing Data Storage

Date of Publish: 2018-07-12

Contents

Introduction

Hadoop Distributed File System (HDFS) is a Java-based file system that provides scalable and reliable data storage. An HDFS cluster contains a NameNode to manage the cluster namespace and DataNodes to store data.

Overview of Apache HDFS

Hadoop Distributed File System (HDFS) is a Java-based file system for storing large volumes of data. Designed to span large clusters of commodity servers, HDFS provides scalable and reliable data storage.

HDFS and Yet Another Resource Navigator (YARN) form the data management layer of Apache Hadoop. YARN provides the resource management while HDFS provides the storage.

HDFS is a scalable, fault-tolerant, distributed storage system that works closely with a wide variety of concurrent data access applications. By distributing storage and computation across many servers, the combined storage resource grows linearly with demand.

Components of an HDFS cluster

An HDFS cluster contains the following main components: a NameNode and DataNodes.

The NameNode manages the cluster metadata that includes file and directory structures, permissions, modifications, and disk space quotas. The file content is split into multiple data blocks, with each block replicated at multiple DataNodes.

The NameNode actively monitors the number of replicas of a block. In addition, the NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.

Benefits of HDFS

HDFS provides the following benefits as a result of which data is stored efficiently and is highly available in the cluster:

- Rack awareness: A node’s physical location is considered when allocating storage and scheduling tasks.
- Minimal data motion: Hadoop moves compute processes to the data on HDFS. Processing tasks can occur on the physical node where the data resides. This significantly reduces network I/O and provides very high aggregate bandwidth.
- Utilities: Dynamically diagnose the health of the file system and rebalance the data on different nodes.
- Version rollback: Allows operators to perform a rollback to the previous version of HDFS after an upgrade, in case of human or systemic errors.
- Standby NameNode: Provides redundancy and supports high availability (HA).
- Operability: HDFS requires minimal operator intervention, allowing a single operator to maintain a cluster of thousands of nodes

Related Information

Apache Hadoop HDFS

Scaling namespaces

You can configure an HDFS federation to use multiple NameNodes and namespaces in a single cluster. By using multiple NameNodes, an HDFS federation can horizontally scale namespaces in a cluster. You can use ViewFs with a federation to create personalized namespace views.

Scaling a cluster using HDFS federation

An HDFS federation scales a cluster horizontally by providing support for multiple independent NameNodes and namespaces, with the DataNodes available as common block storage for all the NameNodes. The support for multiple namespaces improves cluster scalability and provides isolation in a multitenanted environment.

The earlier HDFS configurations without support for federations can be constrained by a single namespace, and consequently, a single NameNode for the entire cluster. In this non-federated environment, the NameNode stores the entire file system metadata in memory. This limits the number of blocks, files, and directories supported on the file system to what can be accommodated in the memory of a single NameNode. In addition, file system operations are limited to the throughput of a single NameNode. These issues of scalability and performance are addressed through an HDFS federation.

In order to scale the name service horizontally, a federation uses multiple independent NameNodes and namespaces. The NameNodes are federated; that is, the NameNodes are independent and do not require coordination with one another. A shared pool of DataNodes is used as common storage for blocks by all the NameNodes. Each DataNode registers with all the NameNodes in the cluster. DataNodes send periodic heartbeats and block reports. They also handle commands from the NameNodes.

Note: You can use ViewFs to create personalized namespace views. ViewFs is analogous to client side mount tables in UNIX or Linux systems.

Related Concepts

Using ViewFs to manage multiple namespaces

Related Information

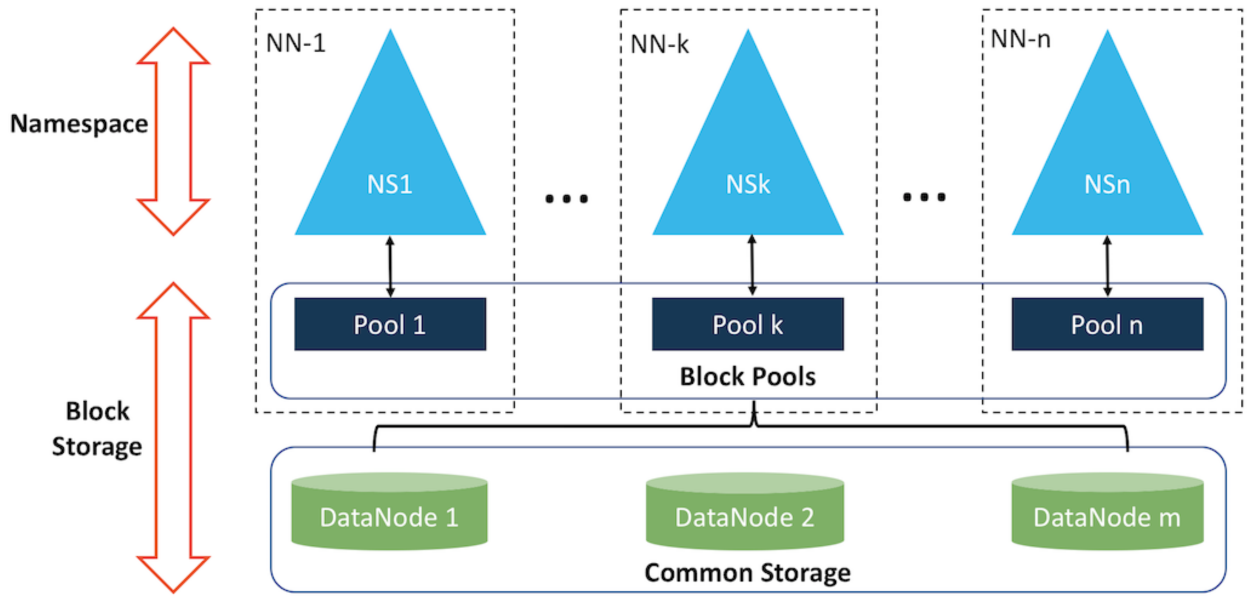
An Introduction to HDFS Federation

Federation terminology

A block pool is a set of data blocks that belongs to a single namespace. A namespace and its block pool together form a namespace volume.

Block pool

A block pool is a set of blocks that belong to a single namespace. DataNodes store blocks for all the block pools in the cluster. Each block pool is managed independently. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. A NameNode failure does not prevent the DataNode from serving other NameNodes in the cluster.



Namespace volume

A namespace and its block pool together are called namespace volume. A namespace volume is a self-contained unit of management. When a NameNode or a namespace is deleted, the corresponding block pool at the DataNodes is deleted.

Cluster ID

A Cluster ID is an identifier for all the nodes in the cluster. When a NameNode is formatted, this identifier is either provided or automatically generated. The same Cluster ID is used for formatting the other NameNodes into the cluster.

Benefits of an HDFS Federation

HDFS federation provides namespace scalability, performance advantage because of scaling read/write throughput, and isolation of applications and users in a multitenanted environment.

- Namespace Scalability: Federation adds namespace horizontal scaling. Large deployments or deployments using lot of small files benefit from namespace scaling by allowing more NameNodes to be added to the cluster.
- Performance: File system throughput is not limited by a single NameNode. Adding more NameNodes to the cluster scales the file system read/write throughput.
- Isolation: A single NameNode does not offer namespace isolation in a multi-user environment. By using multiple NameNodes, different categories of applications and users can be isolated to different namespaces.

Configure an HDFS federation

All the nodes in a federation share a common set of configuration files. To support the common configuration, you must configure a NameService ID for all the NameNodes that you want to include in the federation.

Before you begin

- Ensure that you are configuring the federation during a cluster maintenance window.
- Verify that you have configured HA for all the NameNodes that you want to include in the federation. In addition, ensure that you have configured the value of dfs.nameservices for the NameNodes in hdfs-site.xml.

About this task

A federation allows you to add new NameService IDs to a cluster. Each NameService denotes a new filesystem namespace. You can configure a maximum of four namespaces in a federated environment.

An active NameNode and its standby belong to the NameService ID. To support the common configuration, you must suffix the parameters for the various NameNodes in a federation with the NameService ID.

For example, if you define ns2 as the NameService ID for a federation, then you must add ns2 as a suffix to parameters such as dfs.namenode.rpc-address, dfs.namenode.http-address, and dfs.namenode.secondaryhttp-address.

Note: This task explains how you can configure an HDFS federation using the command line interface. For information about using Ambari to configure a federation, see the topic *Configure HDFS Federation* in the Ambari documentation.

Procedure

1. Verify whether the newly added namespaces are added to the dfs.internal.nameservices parameter in hdfs-site.xml.

   The particular parameter lists all the namespaces that belong to the local cluster.
2. Add the following configuration parameters suffixed with the correct NameService ID for the active and standby NameNodes in the hdfs-site.xml.

Daemon	Configuration Parameter
NameNode	• dfs.namenode.rpc-address • dfs.namenode.http-address • dfs.namenode.https-address • dfs.namenode.servicerpc-address • dfs.namenode.keytab.file • dfs.namenode.name.dir • dfs.namenode.checkpoint.dir • dfs.namenode.checkpoint.edits.dir **Note:** The parameters dfs.namenode.http-address and dfs.namenode.https-address

Daemon	Configuration Parameter
	are optional depending on the http policy configured. In addition, the parameters dfs.namenode.checkpoint.dir and dfs.namenode.checkpoint.edits.dir are optional.

3. Propagate the configuration file updates to all the nodes in the cluster.

Example

The following example shows the configuration for two NameNodes in a federation:

```
<configuration>
  <property>
    <name>dfs.nameservices</name>
    <value>ns1,ns2</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns1</name>
    <value>nn-host1:rpc-port</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns1</name>
    <value>nn-host1:http-port</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns2</name>
    <value>nn-host2:rpc-port</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns2</name>
    <value>nn-host2:http-port</value>
  </property>

  .... Other common configuration ...
</configuration>
```

What to do next

Format every new NameNode that you want to include in the federation.

Format NameNodes

To add a NameNode HA to a federation, you must format each new active NameNode that you want to add to the federation. In addition, you must bootstrap the standby NameNode.

Before you begin

You must have configured the parameters for the NameNodes.

About this task

Note: Ensure that you format only those NameNodes that are associated with newly created namespaces for the federation. Formatting any NameNode with already existing namespaces could result in data loss.

Procedure

1. Format the active NameNode by specifying the Cluster ID.

The Cluster ID must be the same as that of the existing namespaces.

```
hdfs namenode -format [-clusterId <cluster_id>]
```

2. Bootstrap the standby NameNode as specified.

```
hdfs namenode -bootstrapStandby
```

Add a NameNode to an existing HDFS cluster

Adding a NameNode HA to an existing cluster with federated NameNodes requires updating the cluster configuration, propagating the update to all the nodes in the cluster, starting the new NameNodes, and refreshing the DataNodes to identify the new NameNodes.

Procedure

1. Add dfs.nameservices to hdfs-site.xml of the NameNodes that you want to include in the federation.
2. For the active NameNode and its corresponding standby node, update the configuration with the NameService ID suffix.
3. Add the configuration for the new NameNodes to the cluster configuration file.

   For more information about steps 1, 2, and 3; see .
4. Propagate the configuration file updates to all the nodes in the cluster.
5. Start the new active NameNode and its standby node.
6. Refresh the DataNodes in the cluster to identify the new NameNodes.

```
hdfs dfsadmin -refreshNamenodes <datanode_host_name>:<datanode_rpc_port>
```

Configure a federation with a cluster upgrade

Hortonworks Data Platform versions starting with 3.0.0 support HDFS federations. You can configure the federation as you upgrade your cluster.

Procedure

Use the hdfs start namenode command during the upgrade process as specified.

```
hdfs start namenode --config $HADOOP_CONF_DIR  -upgrade -clusterId
 <cluster_ID>
```

If you do not specify the Cluster ID, then a unique ID is auto-generated.

Cluster management operations

To start and stop a cluster, use the start-dfs.sh and stop-dfs.sh commands respectively. To balance data in a cluster with federated NameNodes, run the HDFS Balancer. To decommission a DataNode from a federated cluster, add the DataNode details to an exclude file and distribute the file among all the NameNodes in the cluster.

Balance data in a federation

Depending on your requirements, you can use the HDFS Balancer to balance data either at the level of the DataNodes or the block pools in a cluster with federated NameNodes.

About this task

Balancer balances only data across the cluster and not the namespace.

Procedure

- Run the Balancer using the hadoop-daemon.sh start command.

  ```
  hadoop-daemon.sh start balancer [-policy <policy>]
  ```

  Specify either of the following values for policy:

  - datanode: The default policy that balances data at the level of the DataNode.
  - blockpool: Balances data at the level of the block pool.

Related reference

Balancer commands

Decommission a DataNode from a federation

To decommission a DataNode from a federation, you must add the node to the exclude file and distribute the updated exclude file among all the NameNodes. Each NameNode decommissions its block pool. When all the NameNodes finish decommissioning a DataNode, the DataNode is considered decommissioned.

Procedure

1. In the exclude file, specify the DataNode to decommission.

   If you want to decommission multiple DataNodes, add the corresponding host names separated by a newline character to the exclude file.
2. Distribute the updated exclude file among all the NameNodes.

   distribute-exclude.sh <exclude_file>
3. Refresh all the NameNodes to identify the new exclude file.

   The following command uses HDFS configuration to determine the configured NameNodes in the cluster and refreshes them to identify the new exclude file:

   refresh-namenodes.sh

Using cluster web console to monitor a federation

The cluster web console for a federation helps you monitor the cluster. You can use any NameNode in the cluster to access the console at http://<namenode_host:port>/dfsclusterhealth.jsp.

The cluster web console provides the following information:

- A cluster summary that shows the number of files, number of blocks, total configured storage capacity, and the available and used storage for the entire cluster.
- A list of NameNodes and a summary that includes the number of files, blocks, missing blocks, and live and dead DataNodes for each NameNode. The summary also provides a link to access the web UI for each NameNode.
- The decommissioning status of DataNodes.

Using ViewFs to manage multiple namespaces

View File System (ViewFs) helps you manage clusters with multiple NameNodes; and therefore, multiple namespaces, in an HDFS federation.

ViewFs is analogous to client side mount tables in UNIX or Linux systems. You can use ViewFs to create personalized namespace views and cluster-specific common views for file and directory paths corresponding to locations on different NameNodes of the cluster. You can create a cluster-specific global namespace such that applications behave the same way in a federation as they would in a cluster without support for federated NameNodes.

Note: For information about using Ambari to configure ViewFs, see *Configure ViewFs* in the Ambari documentation.

Related Concepts

Scaling a cluster using HDFS federation

Namespace view in a non-federated environment

Each cluster in an non-federated environment contains a single NameNode that provides a single independent namespace. In addition, the cluster does not share DataNodes with other clusters.

The core-site.xml file has the fs.default.name configuration property set to the NameNode of the cluster.

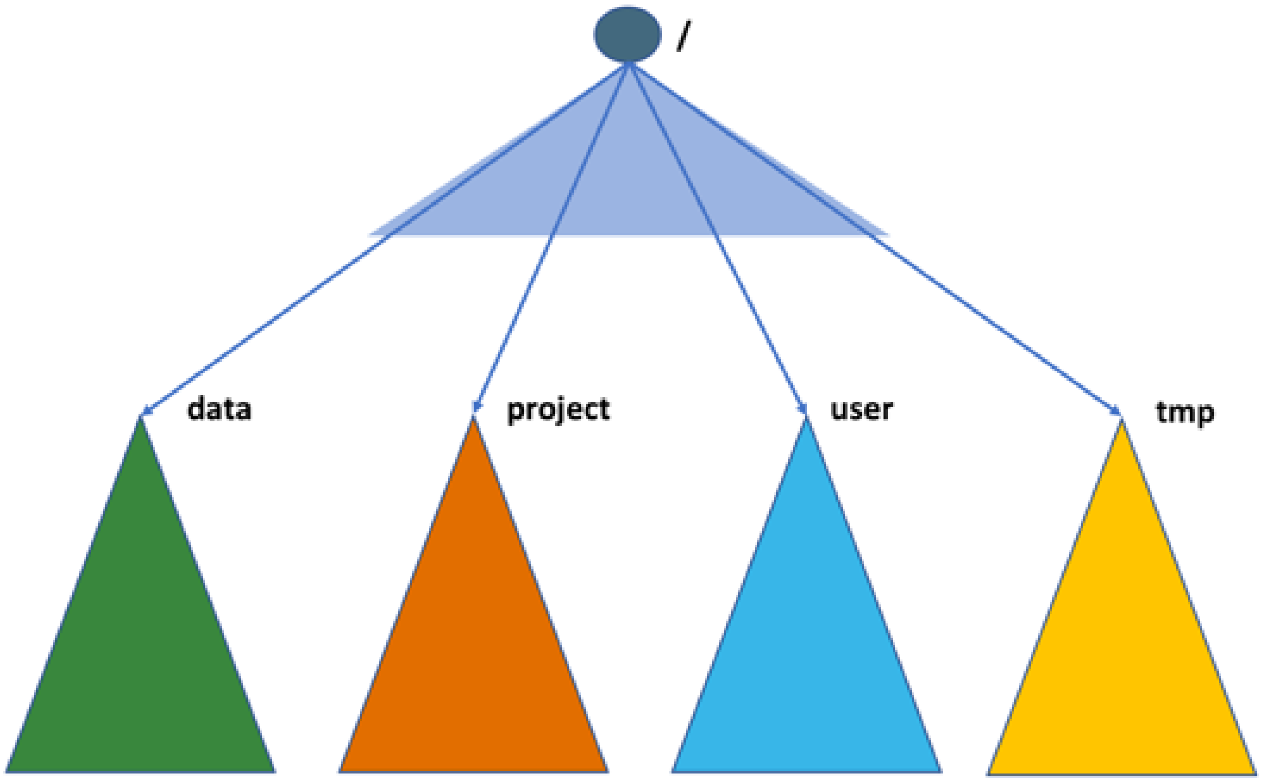
```
<property>
  <name>fs.default.name</name>
  <value>hdfs://namenodeOfClusterX:port</value>
</property>
```

Using the value of the fs.default.name configuration property, you can use path names relative to the NameNode of the cluster instead of the absolute path names. For example, you can use foo/bar to refer to hdfs://namenodeOfClusterX:port/foo/bar.

Namespace view in a federation

Each of the multiple NameNodes in a federation has its own namespace. The NameNodes share the DataNodes of the cluster. Using ViewFs, you can create a cluster namespace view that is similar to a namespace in a cluster without support for federation.

Consider a federation with namespace volumes where each volume contains data distinct from that of others. For example, a federation containing namespace volumes such as /data, /project, /user, and /tmp. Using ViewFs, you can create a cluster namespace view that defines mount points for the namespace volumes at the root level or below. The following figure shows a mount table for the namespace volumes /data, /project, /user, and /tmp:



For the cluster, you can set the default file system to the ViewFs mount table in the core-site.xml file, as shown in the following example:

```
<property>
  <name>fs.default.name</name>
  <value>viewfs://clusterX</value>
</property>
```

In this example, the authority following the viewfs:// scheme in the URI is the mount table name. For easier management of the federation, you should consider specifying the cluster name as the mount table name. In your Hadoop environment, ensure that the configuration files for all the gateway nodes contain the mount tables for all the clusters such that, for each cluster, the default file system is set to the ViewFs mount table for that cluster.

Note: After setting the default file system to the ViewFs mount table for a given cluster, you must remap any directory paths mounted earlier to point to the corresponding namespaces that contain those directories.

Related Concepts

Considerations for working with ViewFs mount table entries

Pathnames on clusters with federated and non-federated NameNodes

The pathnames of the namespace URIs used in clusters with federated and non-federated NameNodes depend on the value configured for the default file system.

In cluster without federated NameNodes, the default file system is set to the NameNode of the cluster. For example, fs.defaultFS= hdfs://nnOfThatCluster:port/. In a federation with ViewFs, the default file system for each cluster is set to the name of the cluster. For example, fs.defaultFS=viewfs://clusterX/.

The value configured for the default file system determines the pathnames of the various URIs used in clusters.

The following table contains examples of pathnames in URIs used in *clusters without federated NameNodes*:

Pathname Example	Description
/foo/bar	The relative path equivalent to hdfs://nnOfClusterX:port/foo/bar.
hdfs://nnOfClusterX:port/foo/bar	The fully qualified path to the foo/bar directory on cluster X. **Note:** You should consider using the relative path instead of the absolute path as that approach helps in moving an application and its data transparently across clusters when required.
hdfs://nnOfClusterY:port/foo/bar	The URI for referring to the foo/bar directory on cluster Y. For example, the command for copying files from one cluster to another can be as follows: `distcp hdfs://namenodeClusterY:port/pathSrc hdfs://namenodeClusterZ:port/pathDest`
hftp://nnClusterX:port/foo/bar	The URI for accessing the foo/bar directory on cluster X through the HFTP file system.
webhdfs://nnClusterX:port/foo/bar	The URI for accessing the foo/bar directory on cluster X through the WebHDFS file system.
http://namenodeClusterX:http_port/webhdfs/v1/foo/bar and http://proxyClusterX:http_port/foo/bar	The HTTP URLs respectively for accessing files through WebHDFS REST API and HDFS proxy.

The following table contains examples of pathnames in URIs used in *clusters with ViewFs*:

Pathname Example	**Description**
/foo/bar	The relative path equivalent to viewfs://clusterX/foo/bar.
viewfs://clusterX/foo/bar	The fully qualified path to the foo/bar directory on cluster X. Unlike a non-federated environment where you must specify the NameNode in the path to each namespace volume, you can use the mount table name in a federated environment to qualify the paths to all the namespace volumes of the cluster. **Note:** You should consider using the relative path instead of the absolute path as that approach helps in moving an application and its data transparently across clusters when required.

viewfs://clusterY/foo/bar	The URI for referring to the foo/bar directory on cluster Y. For example, the command for copying files from one cluster to another can be as follows:
	
	`distcp viewfs://clusterY/pathSrc`
	` viewfs://clusterZ/pathDest`

viewfs://clusterX-hftp/foo/bar	The URI for accessing the foo/bar directory on cluster X through the HFTP file system.
viewfs://clusterX-webhdfs/foo/bar	The URI for accessing the foo/bar directory on cluster X through the WebHDFS file system.
http://namenodeClusterX:http_port/webhdfs/v1/foo/bar and http://proxyClusterX:http_port/foo/bar	The HTTP URLs respectively for accessing files through WebHDFS REST API and HDFS proxy.

Note: You cannot rename files or directories across NameNodes or clusters in both federated and non-federated environments. For example, while the following renaming operation can be performed within the scope of a NameNode, the same is not possible if the source and destination paths belong to different NameNodes:

```
rename /user/joe/myStuff /data/foo/bar
```

Considerations for working with ViewFs mount table entries

After specifying the ViewFs mount table name, you must define the mount table entries to map physical locations on the federation to their corresponding mount points. You must consider factors such as data access, mount levels, and application requirements while defining ViewFs mount table entries.

- Define the ViewFs mount table entries for a cluster in a separate file and reference the file using XInclude in core-site.xml.
- For access to data across clusters, ensure that a cluster configuration contains mount table entries for all the clusters in your environment.
- For a nested directory path, you can define separate ViewFs mounts to point to the top-level directory and the sub-directories based on requirements.

  For a directory user that contains sub-directories such as joe and jane, you can define separate ViewFs mount points to /user, /user/joe, and /user/jane.
- For applications that work across clusters and store persistent file paths, consider defining mount paths of type viewfs://cluster/path.

Such a path definition insulates users from movement of data within the cluster if the data movement is transparent.

- When moving files from one NameNode to another inside a cluster, update the mount point corresponding to the files so that the mount point specifies the correct NameNode.

  Consider an example where the cluster administrator moves /user and /data, originally present on a common NameNode, to separate NameNodes. Before the movement, if the mount points for both /user and /data specified the same NameNode namenodeContainingUserAndData, the administrator must change the mount points after the data movement to specify separate NameNodes namenodeContaingUser and namenodeContainingData respectively.
- A client reads the cluster mount table when submitting a job. The XInclude in core-site.xml is expanded only at the time of job submission. Therefore, if you make any changes to the mount table entries, the jobs must be resubmitted.

Related Concepts

Namespace view in a federation

Example of ViewFs mount table entries

You can specify the ViewFs mount table entries for a cluster in a separate configuration file and reference the file using XInclude in core-site.xml.

Consider a cluster named ClusterX for which the configuration file containing ViewFs mount table entries is referenced as follows:

```
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="mountTable.xml"/>
</configuration>
```

Assume ClusterX to be a federation of three NameNodes that host different directories as specified:

- nn1-clusterx.example.com:8020: /home and /tmp
- nn2-clusterx.example.com:8020: /foo
- nn3-clusterx.example.com:8020: /bar

In addition, the home directory base path is set to /home.

The following example contains the ViewFs mount table entries for the federated cluster ClusterX:

```
<configuration>
  <property>
    <name>fs.viewfs.mounttable.ClusterX.homedir</name>
    <value>/home</value>
  </property>
  <property>
    <name>fs.viewfs.mounttable.ClusterX.link./home</name>
    <value>hdfs://nn1-clusterx.example.com:8020/home</value>
  </property>
  <property>
    <name>fs.viewfs.mounttable.ClusterX.link./tmp</name>
    <value>hdfs://nn1-clusterx.example.com:8020/tmp</value>
  </property>
  <property>
    <name>fs.viewfs.mounttable.ClusterX.link./foo</name>
    <value>hdfs://nn2-clusterx.example.com:8020/foo</value>
  </property>
  <property>
    <name>fs.viewfs.mounttable.ClusterX.link./bar</name>
    <value>hdfs://nn3-clusterx.example.com:8020/bar</value>
  </property>
```

```
</configuration>
```

Optimizing data storage

You can consider the following options to optimize data storage in HDFS clusters: balancing data across disks of a DataNode, balancing data across the DataNodes of a cluster, increasing storage space through erasure coding, applying storage policies for archiving cold data, and using codecs for compressing data.

Balancing data across disks of a DataNode

The HDFS Disk Balancer is a command line tool that evenly distributes data across all the disks of a DataNode to ensure that the disks are effectively utilized. Unlike the HDFS Balancer that balances data across DataNodes of a cluster, the Disk Balancer balances data at the level of individual DataNodes.

Disks on a DataNode can have an uneven spread of data because of reasons such as large amount of writes and deletes or disk replacement. To ensure uniformity in the distribution of data across disks of a specified DataNode, the Disk Balancer operates against the DataNode and moves data blocks from one disk to another.

For a specified DataNode, the Disk Balancer determines the ideal amount of data to store per disk based on its total capacity and used space, and computes the amount of data to redistribute between disks. The Disk Balancer captures this information about data movement across the disks of a DataNode in a plan. On executing the plan, the Disk Balancer moves the data as specified.

Plan the data movement across disks

A Disk Balancer plan identifies the amount of data that should move between the disks of a specified DataNode. The plan contains move steps, where each move step specifies the source and destination disks for data movement and the number of bytes to move.

Before you begin

You must set the dfs.disk.balancer.enabled property in hdfs-site.xml to true.

Procedure

Run the hdfs diskbalancer -plan command by specifying the path to the DataNode.

For example, hdfs diskbalancer -plan node1.mycluster.com.

The specified command generates the following two JSON documents as output in the HDFS namespace: <nodename>.before.json that indicates the state of the cluster before running Disk Balancer, and <nodename>.plan.json that details the data movement plan for the DataNode.

By default, the JSON documents are placed in the following location: /system/diskbalancer/<Creation_Timestamp>. Here, <Creation_Timestamp> indicates the folder named according to the date and time of creation of the JSON documents.

What to do next

Execute the generated plan.

Related Tasks

Execute the Disk Balancer plan

Parameters to configure the Disk Balancer

You must configure various parameters in hdfs-site.xml for effectively planning and executing the Disk Balancer.

Parameter	Description
dfs.disk.balancer.enabled	Controls whether the Disk Balancer is enabled for the cluster. The Disk Balancer executes only if this parameter is set to True. The default value is False.
dfs.disk.balancer.max.disk.throughputInMBperSec	The maximum disk bandwidth that Disk Balancer consumes while transferring data between disks. The default value is 10 MB/s.
dfs.disk.balancer.max.disk.errors	The maximum number of errors to ignore for a move operation between two disks before abandoning the move. The default value of the maximum errors to ignore is 5. For example, if a plan specifies data move operation between three pairs of disks, and if the move between the first pair encounters more than five errors, that move is abandoned and the next move between the second pair of disks starts.
dfs.disk.balancer.block.tolerance.percent	Specifies a threshold value in percentage to consider a move operation successful, and stop moving further data. For example, setting this value to 20% for an operation requiring 10GB data movement indicates that the movement will be considered successful only after 8GB of data is moved.
dfs.disk.balancer.plan.threshold.percent	The ideal storage value for a set of disks in a DataNode indicates the amount of data each disk should have for achieving perfect data distribution across those disks. The threshold percentage defines the value at which disks start participating in data redistribution or balancing operations. Minor imbalances are ignored because normal operations automatically correct some of these imbalances. The default threshold percentage for a disk is 10%; indicating that a disk is used in balancing operations only if the disk contains 10% more or less data than the ideal storage value.

Execute the Disk Balancer plan

For a DataNode, the HDFS Disk Balancer moves data according to the details specified in the JSON document generated as the Disk Balancer plan.

Before you begin

You must have generated the data movement plan for the DataNode.

Procedure

Run the hdfs diskbalancer -execute command by specifying the path to the JSON plan document.

```
hdfs diskbalancer -execute /system/diskbalancer/<Creation_Timestamp>/nodename.plan.json
```

The command reads the DataNode's address from the JSON file and executes the plan.

Related Tasks

Plan the data movement across disks

Disk Balancer commands

In addition to planning for data movement across disks and executing the plan, you can use hdfs diskbalancer sub-commands to query the status of the plan, cancel the plan, identify at a cluster level the DataNodes that require balancing, or generate a detailed report on a specific DataNode that can benefit from running the Disk Balancer.

Planning the data movement for a DataNode

Command:hdfs diskbalancer -plan <datanode>

Argument	Description
<datanode>	Fully qualified name of the DataNode for which you want to generate the plan.

hdfs diskbalancer -plan node1.mycluster.com

The following table lists the additional options that you can use with the hdfs diskbalancer -plan command.

Option	Description
-out	Specify the location within the HDFS namespace where you want to save the output JSON documents that contain the generated plans.
-bandwidth	Specify the maximum bandwidth to use for running the Disk Balancer. This option helps in minimizing the amount of data moved by the Disk Balancer on an operational DataNode. Disk Balancer uses the default bandwidth of 10 MB/s if you do not specify this value.
-thresholdPercentage	The ideal storage value for a set of disks in a DataNode indicates the amount of data each disk should have for achieving perfect data distribution across those disks. The threshold percentage defines the value at which disks start participating in data redistribution or balancing operations. Minor imbalances are ignored because normal operations automatically correct some of these imbalances. The default value of -thresholdPercentage for a disk is 10%; indicating that a disk is used in balancing operations only if the disk contains 10% more or less data than the ideal storage value.
-maxerror	Specify the number of errors to ignore for a move operation between two disks before abandoning the move. Disk Balancer uses the default if you do not specify this value.
-v	Verbose mode. Specify this option for Disk Balancer to display a summary of the plan as output.
-fs	Specify the NameNode to use. Disk Balancer uses the default NameNode from the configuration if you do not specify this value.

Executing the plan

Command:hdfs diskbalancer -execute <JSON file path>

Argument	Description
<JSON file path>	Path to the JSON document that contains the generated plan (nodename.plan.json).

hdfs diskbalancer -execute /system/diskbalancer/nodename.plan.json

Querying the current status of execution

Command:hdfs diskbalancer -query <datanode>

Argument	Description
<datanode>	Fully qualified name of the DataNode for which the plan is running.

hdfs diskbalancer -query nodename.mycluster.com

Cancelling a running plan

Commands:

hdfs diskbalancer -cancel <JSON file path>

Argument	Description
<JSON file path>	Path to the JSON document that contains the generated plan (nodename.plan.json).

hdfs diskbalancer -cancel /system/diskbalancer/nodename.plan.json

OR

hdfs diskbalancer -cancel <planID> -node <nodename>

Argument	Description
planID	ID of the plan to cancel. You can get this value from the output of the hdfs diskbalancer -query command.
nodename	The fully qualified name of the DataNode on which the plan is running.

Viewing detailed report of DataNodes that require Disk Balancer

Commands:

hdfs diskbalancer -fs http://namenode.uri -report -node <file://>

Argument	Description
<file://>	Hosts file listing the DataNodes for which you want to generate the reports.

OR

hdfs diskbalancer -fs http://namenode.uri -report -node [<DataNodeID|IP|Hostname>,...]

Argument	Description
[<DataNodeID|IP|Hostname>,...]	Specify the DataNode ID, IP address, and the host name of the DataNode for which you want to generate the report. For multiple DataNodes, provide the details using comma-separated values.

Viewing details of the top DataNodes in a cluster that require Disk Balancer

Command:hdfs diskbalancer -fs http://namenode.uri -report-node -top <topnum>

Argument	Description
<topnum>	The number of the top DataNodes that require Disk Balancer to be run.

Increasing storage capacity with HDFS erasure coding

HDFS Erasure Coding (EC) can be used to reduce the amount of storage space required for replication.

The default 3x replication scheme in HDFS adds 200% overhead in storage space and other resources such as network bandwidth. For warm and cold datasets with relatively low I/O activities, additional block replicas are rarely accessed during normal operations, but still consume the same amount of resources as the first replica.

Erasure coding provides the same level of fault-tolerance as 3x replication, but uses much less storage space. In a typical erasure coding setup, the storage overhead is not more than 50%.

In storage systems, the most notable usage of EC is in a Redundant Array of Independent Disks (RAID). RAID implements EC through striping, which divides logically sequential data such as a file into smaller units (such as a bit, byte, or block) and stores consecutive units on different disks. This unit of striping distribution is termed as a striping cell. EC uses an encoding process to calculate and store a certain number of parity cells for each stripe of original data cells. An error on any striping cell can be recovered using a decoding calculation based on the surviving data and the parity cells.

Integrating EC with HDFS can improve storage efficiency while still providing similar data durability as traditional replication-based HDFS deployments. As an example, a 3x replicated file with 6 blocks consumes 6*3 = 18 blocks of disk space. But with EC (6 data, 3 parity) deployment, the file consumes only 9 blocks of disk space.

Benefits of erasure coding

HDFS supports Erasure Coding (EC) with data striping at the directory level.

In the context of EC, striping has critical advantages.

Striping enables online EC (writing data immediately in EC format). Clients can directly write erasure-coded data as it requires only a small amount of buffering to calculate parity data. Online EC also enhances sequential I/O performance by leveraging multiple disk spindles in parallel; this is especially desirable in clusters with high-end networking.

In addition, EC with striping naturally distributes a small file to multiple DataNodes and eliminates the need to bundle multiple files into a single coding group.

In typical HDFS clusters, small files can account for over 3/4 of total storage consumption. To better support small files, HDFS supports EC with striping.

How the DataNode recovers failed erasure-coded blocks

The NameNode is responsible for tracking any missing blocks in an EC stripe. The NameNode assigns the task of recovering the blocks to the DataNodes. When a client requests for data and a block is missing, additional read requests are issued to fetch the parity blocks and decode the data.

The recovery task is passed as a heartbeat response. This process is similar to how replicated blocks are recovered after failure. The recovery task consists of the following three phases:

1. Reading the data from source nodes: Input data is read in parallel from the source nodes. Based on the EC policy, it schedules the read requests to all source targets and reads only the minimum number of input blocks for reconstruction.
2. Decoding the data and generating output: New data and parity blocks are decoded from the input data. All missing data and parity blocks are decoded together.
3. Transferring the generated data blocks to target nodes: After the completion of decoding, the recovered blocks are transferred to target DataNodes.

Erasure coding policies

To accommodate heterogeneous workloads, files and directories in an HDFS cluster are allowed to have different replication and EC policies.

Each policy is defined by the following 2 pieces of information:

- The EC Schema: Includes the numbers of data and parity blocks in an EC group (e.g., 6+3), as well as the codec algorithm (for example, Reed-Solomon).
- The size of a striping cell: Determines the granularity of striped reads and writes, including buffer sizes and encoding work.

HDP supports the Reed-Solomon Erasure Coding algorithm. The system default scheme is Reed-Solomon with 6 data blocks, 3 parity blocks, and a 1024 KB cell size (RS-6-3-1024k).

In addition, the following policies are supported: RS-3-2-1024k (Reed-Solomon with 3 data blocks, 2 parity blocks and 1024 KB cell size), RS-LEGACY-6-3-1024k, and XOR-2-1-1024k.

Limitations of erasure coding

Erasure coding works only on new data written to a directory. Files already existing in the directory configured for erasure coding continue using the default 3x replication scheme. In addition, erasure coding might impact the performance of a cluster because of consuming considerable CPU resources and network bandwidth.

Note: Given the current limitations, it is recommended that you use erasure coding only for cold data.

Effect of erasure coding on existing data

Erasure Coding (EC) is set on a per-directory basis; therefore, setting an EC policy on a new or existing directory does not affect existing data in the cluster. Unless Erasure Coding is explicitly set, the default 3x replication scheme continues to be used.

- If you set an Erasure Coding policy on a non-empty directory, the existing files are NOT converted to use Erasure Coding. The default 3x replication will still be used for these existing files.
- You can also move a file from a non-EC directory to an EC directory, or from an EC directory to a non-EC directory. This movement between EC and non-EC directories does NOT change the file's EC or replication strategy. The only way to convert an existing file from non-EC to EC is to copy the file into a directory with an EC policy. You can use distcp to copy files.

Considerations for deploying erasure coding

You must consider factors like network bisection bandwidth and fault-tolerance at the level of the racks while deploying erasure coding in your HDFS clusters.

Erasure Coding places additional demands on the cluster in terms of CPU and network.

Erasure coded files are spread across racks for fault-tolerance. This means that when reading and writing striped files, most operations are off-rack. Thus, network bisection bandwidth is very important.

For fault-tolerance at the rack level, it is also important to have at least as many racks as the configured EC stripe width. For the default EC policy of RS (6,3), this means minimally 9 racks, and around 10 or 11 to handle planned and unplanned outages. For clusters with fewer racks than the stripe width, HDFS cannot maintain fault-tolerance at the rack level, but still attempts to spread a striped file across multiple nodes to preserve fault-tolerance at the node level.

Erasure coding CLI command

Use the hdfs ec command to set erasure coding policies on directories.

```
hdfs ec [generic options]
     [-setPolicy -path <path> [-policy <policyName>] [-replicate]]
     [-getPolicy -path <path>]
     [-unsetPolicy -path <path>]
     [-listPolicies]
     [-addPolicies -policyFile <file>]
     [-listCodecs]
     [-removePolicy -policy <policyName>]
     [-enablePolicy -policy <policyName>]
     [-disablePolicy -policy <policyName>]
     [-help [cmd ...]]
```

Options:

- [-setPolicy [-p <policyName>] <path>]: Sets an EC policy on a directory at the specified path. The following EC policies are supported: RS-3-2-1024k (Reed-Solomon with 3 data blocks, 2 parity blocks and 1024 KB cell size), RS-6-3-1024k, RS-LEGACY-6-3-1024k, and XOR-2-1-1024k.

  <path>: A directory in HDFS. This is a mandatory parameter. Setting a policy only affects newly created files, and does not affect existing files.

  <policyName>: The EC policy to be used for files under the specified directory. This is an optional parameter, specified using the -p flag. If no policy is specified, the system default Erasure Coding policy is used. The default policy is RS-6-3-1024k.
- -replicate: Forces a directory to use the default 3x replication scheme.

  Note: You cannot specify -replicate and -policy <policyName> at the same time. Both the arguments are optional.
- -getPolicy -path <path>: Gets details of the EC policy of a file or directory for the specified path.
- [-unsetPolicy -path <path>]: Removes an EC policy already set by a setPolicy on a directory. This option does not work on a directory that inherits the EC policy from a parent directory. If you run this option on a directory that does not have an explicit policy set, no error is returned.
- [-addPolicies -policyFile <file>]: Adds a list of EC policies. HDFS allows you to add 64 policies in total, with the policy ID in the range of 64 to 127. Adding policies fails if there are already 64 policies.
- [-listCodecs]: Lists all the supported EC erasure coding codecs and coders in the system.
- [-removePolicy -policy <policyName>]: Removes an EC policy.
- [-enablePolicy -policy <policyName>]: Enables an EC policy.
- [-disablePolicy -policy <policyName>]: Disables an EC policy.
- [-help]: Displays help for a given command, or for all commands if none is specified.

Erasure coding background recovery work on the DataNodes can be tuned using the following configuration parameters in hdfs-site.xml.

- dfs.datanode.ec.reconstruction.stripedread.timeout.millis: Timeout for striped reads. Default value is 5000 ms.
- dfs.datanode.ec.reconstruction.threads: Number of threads used by the DataNode for the background recovery task. The default value is 8 threads.
- dfs.datanode.ec.reconstruction.stripedread.buffer.size: Buffer size for reader service. Default value is 64 KB.
- dfs.datanode.ec.reconstruction.xmits.weight: The relative weight of xmits used by the EC background recovery task when compared to replicated block recovery. The default value is 0.5.

If the parameter is set to 0, the EC task always has one xmit. The xmits of an erasure coding recovery task are calculated as the maximum value between the number of read streams and the number of write streams. For example, if an EC recovery task needs to read from six nodes and write to two nodes, the xmit value is max(6, 2) * 0.5 = 3.

Erasure coding examples

You can use the hdfs ec command with its various options to set erasure coding policies on directories.

Viewing the list of erasure coding policies

The following example shows how you can view the list of available erasure coding policies:

```
hdfs ec -listPolicies
Erasure Coding Policies:
ErasureCodingPolicy=[Name=RS-10-4-1024k, Schema=[ECSchema=[Codec=rs,
 numDataUnits=10, numParityUnits=4]], CellSize=1048576, Id=5], State=DISABLED
ErasureCodingPolicy=[Name=RS-3-2-1024k, Schema=[ECSchema=[Codec=rs,
 numDataUnits=3, numParityUnits=2]], CellSize=1048576, Id=2], State=DISABLED
ErasureCodingPolicy=[Name=RS-6-3-1024k, Schema=[ECSchema=[Codec=rs,
 numDataUnits=6, numParityUnits=3]], CellSize=1048576, Id=1], State=ENABLED
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k, Schema=[ECSchema=[Codec=rs-
legacy, numDataUnits=6, numParityUnits=3]], CellSize=1048576, Id=3],
 State=DISABLED
ErasureCodingPolicy=[Name=XOR-2-1-1024k, Schema=[ECSchema=[Codec=xor,
 numDataUnits=2, numParityUnits=1]], CellSize=1048576, Id=4], State=DISABLED
```

Enabling an erasure coding policy

In the previous example, the list of erasure coding policies indicates that RS-6-3-1024k is already enabled. If required, you can enable additional policies as mentioned in the following example:

```
hdfs ec -enablePolicy -policy RS-3-2-1024k
Erasure coding policy RS-3-2-1024k is enabled
```

Setting an erasure coding policy

The following example shows how you can set the erasure coding policy RS-6-3-1024k on a particular directory:

```
hdfs ec -setPolicy -path /data/dir1 -policy RS-6-3-1024k
Set erasure coding policy RS-6-3-1024k on /data/dir1
```

To confirm whether the specified directory has the erasure coding policy applied, run the hdfs ec -getPolicy command:

```
hdfs ec -getPolicy -path /data/dir1
RS-6-3-1024k
```

Checking the block status on an erasure-coded directory

After enabling erasure coding on a directory, you can check the block status by running the hdfs fsck command. The following example output shows the status of the erasure-coded blocks:

```
hdfs fsck /data/dir1
.
.
.
Erasure Coded Block Groups:
Total size: 434424 B
Total files: 1
Total block groups (validated): 1 (avg. block group size 434424 B)
```

```
Minimally erasure-coded block groups: 1 (100.0 %)
Over-erasure-coded block groups: 0 (0.0 %)
Under-erasure-coded block groups: 0 (0.0 %)
Unsatisfactory placement block groups: 0 (0.0 %)
Average block group size: 4.0
Missing block groups: 0
Corrupt block groups: 0
Missing internal blocks: 0 (0.0 %)
FSCK ended at Fri Mar 21 19:39:11 UTC 2018 in 1 milliseconds

The filesystem under path '/data/dir1' is HEALTHY
```

Changing the erasure coding policy

You can use the hdfs ec setPolicy command to change the erasure coding policy applied on a particular directory.

```
hdfs ec -setPolicy -path /data/dir1 -policy RS-3-2-1024k
Set erasure coding policy RS-3-2-1024k on /data/dir1
```

You can check the check the block status after applying the new policy. The following example output shows the status of the erasure-coded blocks for a directory that has the RS-3-2-1024k policy:

```
hdfs fsck /data/dir1
.
.
.
Erasure Coded Block Groups:
Total size: 68644 B
Total files: 2
Total block groups (validated): 2 (avg. block group size 34322 B)
Minimally erasure-coded block groups: 2 (100.0 %)
Over-erasure-coded block groups: 0 (0.0 %)
Under-erasure-coded block groups: 0 (0.0 %)
Unsatisfactory placement block groups: 0 (0.0 %)
Average block group size: 2.5
Missing block groups:  0
Corrupt block groups:  0
Missing internal blocks: 0 (0.0 %)
FSCK ended at Mon Apr 09 10:11:06 UTC 2018 in 3 milliseconds


The filesystem under path '/data/dir1' is HEALTHY
```

You can apply the default 3x replication policy and check the block status as specified in the following examples:

```
hdfs ec -setPolicy -path /data/dir1 -replicate
Set erasure coding policy replication on /tmp/data1/
Warning: setting erasure coding policy on a non-empty directory will not
 automatically convert existing files to replication
```

```
hdfs fsck /data/dir1
.
.
.
Erasure Coded Block Groups:
Total size: 34322 B
Total files: 1
Total block groups (validated): 1 (avg. block group size 34322 B)
Minimally erasure-coded block groups: 1 (100.0 %)
Over-erasure-coded block groups: 0 (0.0 %)
Under-erasure-coded block groups: 0 (0.0 %)
Unsatisfactory placement block groups: 0 (0.0 %)
Average block group size: 2.0
```

```
Missing block groups:  0
Corrupt block groups:  0
Missing internal blocks: 0 (0.0 %)
FSCK ended at Tue Apr 10 04:34:14 UTC 2018 in 2 milliseconds

The filesystem under path '/data/dir1' is HEALTHY
```

Increasing storage capacity with HDFS compression

Linux supports GzipCodec, DefaultCodec, BZip2Codec, LzoCodec, and SnappyCodec. Typically, GzipCodec is used for HDFS compression.

To configure data compression, you can either enable a data compression codec, for example, GZipCodec, as the default or use the codec from the command line with a one-time job.

Enable GZipCodec as the default compression codec

For the MapReduce framework, update relevant properties in core-site.xml and mapred-site.xml to enable GZipCodec as the default compression codec.

Procedure

1. Edit the core-site.xml file on the NameNode host machine.

```

<property>
  <name>io.compression.codecs</name>
  <value>org.apache.hadoop.io.compress.GzipCodec,
    org.apache.hadoop.io.compress.DefaultCodec,com.hadoop.compression.lzo.
    LzoCodec,org.apache.hadoop.io.compress.SnappyCodec</value>
  <description>A list of the compression codec classes that can be used
    for compression/decompression.</description>
</property>
```

2. Edit the mapred-site.xml file on the JobTracker host machine.

```

<property>
  <name>mapreduce.map.output.compress</name>
  <value>true</value>
</property>

<property>
  <name>mapreduce.map.output.compress.codec</name>
  <value>org.apache.hadoop.io.compress.GzipCodec</value>
</property>

<property>
  <name>mapreduce.output.fileoutputformat.compress.type</name>
  <value>BLOCK</value>
</property>
```

3. Optional: Enable the following two configuration parameters to enable job output compression. Edit the mapred-site.xml file on the Resource Manager host machine.

```

<property>
  <name>mapreduce.output.fileoutputformat.compress</name>
  <value>true</value>
</property>
```

```
<property>
  <name>mapreduce.output.fileoutputformat.compress.codec</name>
  <value>org.apache.hadoop.io.compress.GzipCodec</value>
</property>
```

4. Restart the cluster.

Use GZipCodec with a one-time job

You can configure GZipcodec to compress the output of a MapReduce job.

Procedure

To use GzipCodec with a one-time only job, add the options to configure compression for the MapReduce job and configure GZipCodec for the output of the job.

```
hadoop jar hadoop-examples-1.1.0-SNAPSHOT.jar sort sbr"-
Dmapred.compress.map.output=true"
sbr"-
Dmapred.map.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec"
sbr "-Dmapred.output.compress=true"
sbr"-
Dmapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec"sbr
 -outKey org.apache.hadoop.io.Textsbr
-outValue org.apache.hadoop.io.Text input output
```

Setting archival storage policies

Archival storage lets you store data on physical media with high storage density and low processing resources.

HDFS storage types

HDFS storage types can be used to assign data to different types of physical storage media.

The following storage types are available:

- DISK: Disk drive storage (default storage type)
- ARCHIVE: Archival storage (high storage density, low processing resources)
- SSD: Solid State Drive
- RAM_DISK: DataNode Memory

If no storage type is assigned, DISK is used as the default storage type.

HDFS storage policies

You can store data on DISK or ARCHIVE storage types using preconfigured storage policies.

The following preconfigured storage policies are available:

- HOT: Used for both storage and compute. Data that is being used for processing will stay in this policy. When a block is HOT, all replicas are stored on DISK. There is no fallback storage for creation, and ARCHIVE is used for replication fallback storage.
- WARM: Partially HOT and partially COLD. When a block is WARM, the first replica is stored on DISK, and the remaining replicas are stored on ARCHIVE. The fallback storage for both creation and replication is DISK, or ARCHIVE if DISK is unavailable.
- COLD: Used only for storage, with limited compute. Data that is no longer being used, or data that needs to be archived, is moved from HOT storage to COLD storage. When a block is COLD, all replicas are stored on ARCHIVE, and there is no fallback storage for creation or replication.

The following table summarizes these replication policies:

Policy ID	Policy Name	Replica Block Placement (for n replicas)	Fallback storage for creation	Fallback storage for replication
12	HOT (default)	Disk: n	<none>	ARCHIVE
8	WARM	Disk: 1, ARCHIVE: n-1	DISK, ARCHIVE	DISK, ARCHIVE
4	COLD	ARCHIVE: n	<none>	<none>

Configure archival storage

To configure archival storage for a DataNode, you must assign the ARCHIVE storage type to the DataNode, set storage policies, and move blocks that violate the storage policy to the appropriate storage type.

Procedure

1. Shut down the DataNode.
2. Assign the ARCHIVE Storage Type to the DataNode.

   You can use the dfs.datanode.data.dir property in the /etc/hadoop/conf/hdfs-site.xml file to assign the ARCHIVE storage type to a DataNode.

   The dfs.datanode.data.dir property determines where on the local filesystem a DataNode should store its blocks.

   If you specify a comma-delimited list of directories, data will be stored in all named directories, typically on different devices. Directories that do not exist are ignored. You can specify that each directory resides on a different type of storage: DISK, SSD, ARCHIVE, or RAM_DISK.

   To specify a DataNode as DISK storage, specify [DISK] and a local file system path. For example:

   ```
   <property>
     <name>dfs.datanode.data.dir</name>
     <value>[DISK]/grid/1/tmp/data_trunk</value>
   </property>
   ```

   To specify a DataNode as ARCHIVE storage, insert [ARCHIVE] at the beginning of the local file system path. For example:

   ```
   <property>
     <name>dfs.datanode.data.dir</name>
     <value>[ARCHIVE]/grid/1/tmp/data_trunk</value>
   </property>
   ```
3. Depending on your requirements, either set a storage policy or list the already applied storage policies on a specified file or directory.

If you want to...	Use this command...
Set a storage policy	hdfs storagepolicies -setStoragePolicy <path> <policyname>
List storage policies	hdfs storagepolicies -getStoragePolicy <path>

   Note: When you update a storage policy setting on a file or directory, the new policy is *not* automatically enforced. You must use the HDFS mover data migration tool to actually move blocks as specified by the new storage policy.
4. Start the DataNode.
5. Use the HDFS mover tool according to the specified storage policy.

The HDFS mover data migration tool scans the specified files in HDFS and verifies if the block placement satisfies the storage policy. For the blocks that violate the storage policy, the tool moves the replicas to a different storage type in order to fulfill the storage policy requirements.

Commands for configuring storage policies

Depending on your requirements, use the hdfs storagepolicy sub-commands to set storage policies or list the storage policies applied on a file or a directory.

Setting storage policies

Command: hdfs storagepolicies -setStoragePolicy <path> <policyName>

Argument	Description
<path>	The path to a directory or file.
<policyName>	The name of the storage policy.

hdfs storagepolicies -setStoragePolicy /cold1 COLD

Listing storage policies

Command: hdfs storagepolicies -getStoragePolicy <path>

Argument	Description
<path>	The path to a directory or file.

hdfs storagepolicies -getStoragePolicy /cold1

The HDFS mover command

You can use the hdfs mover command to move replicas of data blocks that violate the storage policy set on a file or a directory to a storage type that fulfills the policy requirements.

Command: hdfs mover [-p <files/dirs> | -f <local file name>]

Argument	Description
-f <local file>	Specify a local file containing a list of HDFS files or directories to migrate.
-p <files/dirs>	Specify a space-separated list of HDFS files or directories to migrate.

Note: When both -p and -f options are omitted, the default path is the root directory.

Balancing data across an HDFS cluster

The HDFS Balancer is a tool for balancing the data across the storage devices of a HDFS cluster.

You can also specify the source DataNodes, to free up the spaces in particular DataNodes. You can use a block distribution application to pin its block replicas to particular DataNodes so that the pinned replicas are not moved for cluster balancing.

Why HDFS data Becomes unbalanced

Factors such as addition of DataNodes, block allocation in HDFS, and behavior of the client application can lead to the data stored in HDFS clusters becoming unbalanced.

Addition of DataNodes

When new DataNodes are added to a cluster, newly created blocks are written to these DataNodes from time to time. The existing blocks are not moved to them without using the HDFS Balancer.

Behavior of the client application

In some cases, a client application might not write data uniformly across the DataNode machines. A client application might be skewed in writing data, and might always write to some particular machines but not others. HBase is an example of such a client application. In other cases, the client application is not skewed by design, for example, MapReduce or YARN jobs.

The data is skewed so that some of the jobs write significantly more data than others. When a Datanode receives the data directly from the client, it stores a copy to its local storage for preserving data locality. The DataNodes receiving more data generally have higher storage utilization.

Block Allocation in HDFS

HDFS uses a constraint satisfaction algorithm to allocate file blocks. Once the constraints are satisfied, HDFS allocates a block by randomly selecting a storage device from the candidate set uniformly. For large clusters, the blocks are essentially allocated randomly in a uniform distribution, provided that the client applications write data to HDFS uniformly across the DataNode machines. Uniform random allocation might not result in a uniform data distribution because of randomness. This is generally not a problem when the cluster has sufficient space. The problem becomes serious when the cluster is nearly full.

Configurations and CLI options for the HDFS Balancer

You can configure the HDFS Balancer by changing various configuration options or by using the command line.

Properties for configuring the Balancer

Depending on your requirements, you can configure various properties for the HDFS Balancer.

dfs.datanode.balance.max.concurrent.moves	Limits the maximum number of concurrent block moves that a Datanode is allowed for balancing the cluster. If you set this configuration in a Datanode, the Datanode throws an exception when the limit is exceeded. If you set this configuration in the HDFS Balancer, the HDFS Balancer schedules concurrent block movements within the specified limit. The Datanode setting and the HDFS Balancer setting can be different. As both settings impose a restriction, an effective setting is the minimum of them. It is recommended that you set this to the highest possible value in Datanodes and adjust the runtime value in the HDFS Balancer to gain the flexibility. The default value is 5. You can reconfigure without Datanode restart. Follow these steps to reconfigure a Datanode: **1.** Change the value of dfs.datanode.balance.max.concurrent.moves in the configuration xml file on the Datanode machine. **2.** Start a reconfiguration task. Use the hdfs dfsadmin -reconfig datanode <dn_addr>:<ipc_port> start command. For example, suppose a Datanode has 12 disks. You can set the configuration to 24, a small multiple of the number of disks, in the Datanodes. Setting it to a

	higher value might not be useful, and only increases disk contention. If the HDFS Balancer is running in a maintenance window, the setting in the HDFS Balancer can be the same, that is, 24, to use all the bandwidth. However, if the HDFS Balancer is running at same time as other jobs, you set it to a smaller value, for example, 5, in the HDFS Balancer so that there is bandwidth available for the other jobs.
dfs.datanode.balance.bandwidthPerSec	Limits the bandwidth in each Datanode using for balancing the cluster. Changing this configuration does not require restarting Datanodes. Use the dfsadmin -setBalancerBandwidth command. The default is 1048576 (=1MB/s).
dfs.balancer.moverThreads	Limits the number of total concurrent moves for balancing in the entire cluster. Set this property to the number of threads in the HDFS Balancer for moving blocks. Each block move requires a thread. The default is 1000.
dfs.balancer.max-size-to-move	With each iteration, the HDFS Balancer chooses DataNodes in pairs and moves data between the DataNode pairs. Limits the maximum size of data that the HDFS Balancer moves between a chosen DataNode pair. If you increase this configuration when the network and disk are not saturated, increases the data transfer between the DataNode pair in each iteration while the duration of an iteration remains about the same. The default is 10737418240 (10GB).
dfs.balancer.getBlocks.size	Specifies the total data size of the block list returned by a getBlocks(..). When the HDFS Balancer moves a certain amount of data between source and destination DataNodes, it repeatedly invokes the getBlocks(..) rpc to the Namenode to get lists of blocks from the source DataNode until the required amount of data is scheduled. The default is 2147483648 (2GB).
dfs.balancer.getBlocks.min-block-size	Specifies the minimum block size for the blocks used to balance the cluster. The default is 10485760 (10MB)
dfs.datanode.block-pinning.enabled	Specifies if block-pinning is enabled. When you create a file, a user application can specify a list of favorable DataNodes by way of the file creation API in DistributedFileSystem. The NameNode uses its best effort, allocating blocks to the favorable DataNodes. If dfs.datanode.block-pinning.enabled is set to true, if a block replica is written to a favorable DataNode, it is “pinned” to that DataNode. The pinned replicas are not moved for cluster balancing to keep them stored in the

specified favorable DataNodes. This feature is useful for block distribution aware user applications such as HBase.

The default is false.

Balancer commands

You can use various command line options with the hdfs balancer command to work with the HDFS Balancer.

Balancing Policy, Threshold, and Blockpools

[-policy <policy>]	Specifies which policy to use to determine if a cluster is balanced. The two supported policies are blockpool and datanode. Setting the policy to blockpool means that the cluster is balanced if each pool in each node is balanced while datanode means that a cluster is balanced if each DataNode is balanced. The default policy is datanode.
[-threshold <threshold>]	Specifies a number in [1.0, 100.0] representing the acceptable threshold of the percentage of storage capacity so that storage utilization outside the average +/- the threshold is considered as over/under utilized. The default threshold is 10.0.
[-blockpools <comma-separated list of blockpool ids>]	Specifies a list of block pools on which the HDFS Balancer runs. If the list is empty, the HDFS Balancer runs on all existing block pools. The default value is an empty list.

Include and Exclude Lists

[-include [-f <hosts-file> | <comma-separated list of hosts>]]	When the include list is non-empty, only the DataNodes specified in the list are balanced by the HDFS Balancer. An empty include list means including all the DataNodes in the cluster. The default value is an empty list.
[-exclude [-f <hosts-file> | <comma-separated list of hosts>]]	The DataNodes specified in the exclude list are excluded so that the HDFS Balancer does not balance those DataNodes. An empty exclude list means that no DataNodes are excluded. When a DataNode is specified in both in the include list and the exclude list, the DataNode is excluded. The default value is an empty list.

Idle-Iterations and Run During Upgrade

[-idleiterations <idleiterations>]	Specifies the number of consecutive iterations in which no blocks have been moved before the HDFS Balancer terminates with the NO_MOVE_PROGRESS exit status. Specify -1 for infinite iterations. The default is 5.

[-runDuringUpgrade]	If specified, the HDFS Balancer runs even if there is an ongoing HDFS upgrade. If not specified, the HDFS Balancer terminates with the UNFINALIZED_UPGRADE exit status. When there is no ongoing upgrade, this option has no effect. It is usually not desirable to run HDFS Balancer during upgrade. To support rollback, blocks being deleted from HDFS are moved to the internal trash directory in DataNodes and not actually deleted. Running the HDFS Balancer during upgrading cannot reduce the usage of any DataNode storage.

Source Datanodes

[-source [-f <hosts-file> | <comma-separated list of hosts>]]	Specifies the source DataNode list. The HDFS Balancer selects blocks to move from only the specified DataNodes. When the list is empty, all the DataNodes are chosen as a source. The option can be used to free up the space of some particular DataNodes in the cluster. Without the -source option, the HDFS Balancer can be inefficient in some cases. The default value is an empty list. The following table shows an example, where the average utilization is 25% so that D2 is within the 10% threshold. It is unnecessary to move any blocks from or to D2. Without specifying the source nodes, HDFS Balancer first moves blocks from D2 to D3, D4 and D5, since they are under the same rack, and then moves blocks from D1 to D2, D3, D4 and D5. By specifying D1 as the source node, HDFS Balancer directly moves blocks from D1 to D3, D4 and D5.

Table 1: Example of Utilization Movement

Datanodes (with the same capacity)	Utilization	Rack
D1	95%	A
D2	30%	B
D3, D4, and D5	0%	B

Related Tasks

Balance data in a federation

Recommended configurations for the Balancer

The HDFS Balancer can run in either Background or Fast modes. Depending on the mode in which you want the Balancer to run, you can set various properties to recommended values.

Background and Fast Modes

	HDFS Balancer runs as a background process. The cluster serves other jobs and applications at the same time.
Fast Mode	HDFS Balancer runs at maximum (fast) speed.

Table 2: DataNode Configuration Properties

Property	Default	Background Mode	Fast Mode
dfs.datanode.balance.-max.concurrent.moves	5	4 x (# of disks)	4 x (# of disks)
dfs.datanode.balance.-max.bandwidthPerSec	1048576 (1 MB)	use default	10737418240 (10 GB)

Table 3: Balancer Configuration Properties

Property	Default	Background Mode	Fast Mode
dfs.datanode.balance.-max.concurrent.moves	5	# of disks	4 x (# of disks)
dfs.balancer.-moverThreads	1000	use default	20,000
dfs.balancer.-max-size-to-move	10737418240 (10 GB)	1073741824 (1GB)	107374182400 (100 GB)
dfs.balancer.-getBlocks.min-block-size	10485760 (10 MB)	use default	104857600 (100 MB)

Cluster balancing algorithm

The HDFS Balancer runs in iterations. Each iteration contains the following four steps: storage group classification, storage group pairing, block move scheduling, and block move execution.

Storage group classification

The HDFS Balancer first invokes the getLiveDatanodeStorageReport rpc to the Namenode to the storage report for all the storages in all Datanodes. The storage report contains storage utilization information such as capacity, dfs used space, remaining space, and so forth, for each storage in each DataNode.

A Datanode can contain multiple storages and the storages can have different storage types. A storage group Gi,T is defined to be the group of all the storages with the same storage type T in Datanode i. For example, Gi,DISK is the storage group of all the DISK storages in Datanode i. For each storage type T in each DataNode i, HDFS Balancer computes Storage Group Utilization (%)

Ui,T = 100% (storage group used space)/(storage group capacity),

and Average Utilization (%)

Uavg,T = 100% * (sum of all used spaces)/(sum of all capacities).

Let # be the threshold parameter (default is 10%) and GI,T be the storage group with storage type T in DataNode I.

```
                 Over-Utilized:       {Gi,T :  Uavg,T + # < Ui,T},
Average + Threshold
 ---------------------------------------------------------------------------
                   Above-Average: {Gi,T :  Uavg,T < Ui,T <= Uavg,T + #},
Average
 --------------------------------------------------------------------------------
                   Below-Average: {Gi,T :  Uavg,T - # <= Ui,T <= Uavg,T},
Average - Threshold
 ---------------------------------------------------------------------------
                   Under-Utilized:      {Gi,T :  Ui,T < Uavg,T - # }.
```

A storage group is over-utilized or under-utilized if its utilization is larger or smaller than the difference between the average and the threshold. A storage group is above-average or below-average if its utilization is larger or smaller than average but within the threshold.

If there are no over-utilized storages and no under-utilized storages, the cluster is said to be balanced. The HDFS Balancer terminates with a SUCCESS state. Otherwise, it continues with storage group pairing.

Storage group pairing

The HDFS Balancer selects over-utilized or above-average storage as source storage, and under-utilized or below-average storage as target storage. It pairs a source storage group with a target storage group (source # target) in a priority order depending on whether or not the source and the target storage reside in the same rack.

The Balancer uses the following priority orders for pairing storage groups from the source and the target.

- Same-Rack (where the source and the target storage reside in the same rack)

  Over-Utilized # Under-Utilize

  Over-Utilized # Below-Average

  Above-Average # Under-Utilized
- Any (where the source and target storage do not reside in the same rack)

  Over-Utilized # Under-Utilized

  Over-Utilized # Below-Average

  Above-Average # Under-Utilized

Block move scheduling

For each source-target pair, the HDFS Balancer chooses block replicas from the source storage groups and schedules block moves.

A block replica in a source DataNode is a good candidate if it satisfies all of the following conditions:

- The storage type of the block replica in the source DataNode is the same as the target storage type.
- The storage type of the block replica is not already scheduled.
- The target does not already have the same block replica.
- The number of racks of the block is not reduced after the move.

Logically, the HDFS Balancer schedules a block replica to be “moved” from a source storage group to a target storage group. In practice, a block usually has multiple replicas. The block move can be done by first copying the replica from a proxy, which can be any storage group containing one of the replicas of the block, to the target storage group, and then deleting the replica in the source storage group.

After a candidate block in the source DataNode is specified, the HDFS Balancer selects a storage group containing the same replica as the proxy. The HDFS Balancer selects the closest storage group as the proxy in order to minimize the network traffic.

When it is impossible to schedule a move, the HDFS Balancer terminates with a NO_MOVE_BLOCK exit status.

Block move execution

The HDFS Balancer dispatches a scheduled block move by invoking the DataTransferProtocol.replaceBlock(..) method to the target DataNode.

The Balancer specifies the proxy, and the source as delete-hint in the method call. The target DataNode copies the replica directly from the proxy to its local storage. When the copying process has been completed, the target DataNode reports the new replica to the NameNode with the delete-hint. NameNode uses delete-hint to delete the extra replica, that is, delete the replica stored in the source.

After all block moves are dispatched, the HDFS Balancer waits until all the moves are completed. Then, the HDFS Balancer continues running a new iteration and repeats all of the steps. If the scheduled moves fail for 5 consecutive iterations, the HDFS Balancer terminates with a NO_MOVE_PROGRESS exit status.

Exit statuses for the HDFS Balancer

THe HDFS Balancer concludes a cluster balancing operation with a specific exit status that indicates whether the operation succeeded or failed, with supporting reasons.

Table 4: Exit Statuses for the HDFS Balancer

Status	Value	Description
SUCCESS	0	The cluster is balanced. There are no over or under-utilized storages, with regard to the specified threshold.
ALREADY_RUNNING	-1	Another HDFS Balancer is running.
NO_MOVE_BLOCK	-2	The HDFS Balancer is not able to schedule a move.
NO_MOVE_PROGRESS	-3	All of the scheduled moves have failed for 5 consecutive iterations.
IO_EXCEPTION	-4	An IOException occurred.
ILLEGAL_ARGUMENTS	-5	An illegal argument in the command or configuration occurred.
INTERUPTED	-6	The HDFS Balancer process was interrupted.
UNFINALIZED_UPGRADE	-7	The cluster is being upgraded.

Optimizing performance

You can consider the following options to optimize the performance of an HDFS cluster: swapping disk drives on a DataNode, caching data, configuring rack awareness, customizing HDFS, optimizing NameNode disk space with Hadoop archives, identifying slow DataNodes and improving them, optimizing small write operations by using DataNode memory as storage, and implementing short-circuit reads.

Improving performance with centralized cache management

Centralized cache management enables you to specify paths to directories that are cached by HDFS, thereby improving performance for applications that repeatedly access the same data.

Centralized cache management in HDFS is an explicit caching mechanism. The NameNode communicates with DataNodes that have the required data blocks available on disk, and instructs the DataNodes to cache the blocks in off-heap caches.

Benefits of centralized cache management in HDFS

Centralized cache management in HDFS offers many significant advantages such as explicit pinning, querying cached blocks for task placement, and improving cluster memory utilization.

- Explicit pinning prevents frequently used data from being evicted from memory. This is particularly important when the size of the working set exceeds the size of main memory, which is common for many HDFS workloads.

- Because DataNode caches are managed by the NameNode, applications can query the set of cached block locations when making task placement decisions. Co-locating a task with a cached block replica improves read performance.
- When a block has been cached by a DataNode, clients can use a more efficient zero-copy read API. Since checksum verification of cached data is done once by the DataNode, clients can incur essentially zero overhead when using this new API.
- Centralized caching can improve overall cluster memory utilization. When relying on the operating system buffer cache on each DataNode, repeated reads of a block will result in all n replicas of the block being pulled into the buffer cache. With centralized cache management, you can explicitly pin only m of the n replicas, thereby saving n-m memory.

Use cases for centralized cache management

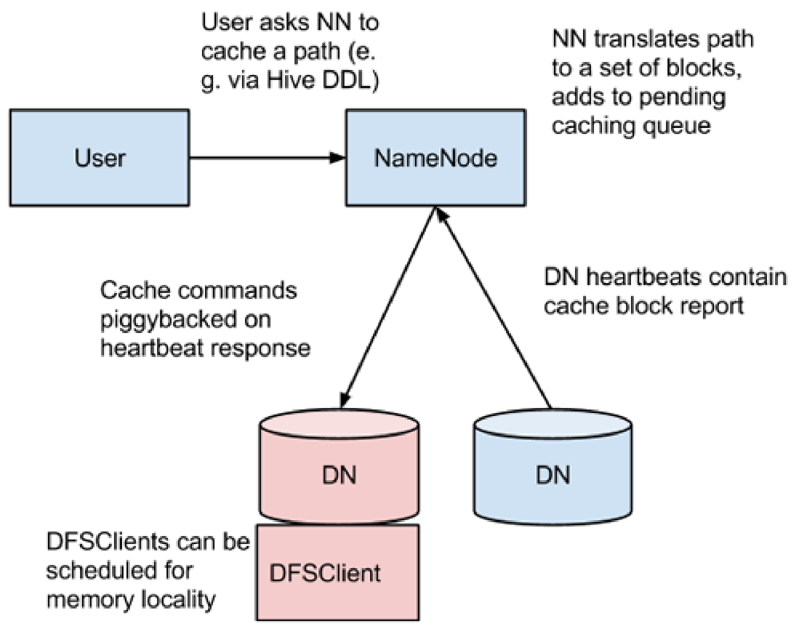
Centralized cache management is useful for files that are accessed repeatedly and for mixed workloads that have performance SLAs.

- Files that are accessed repeatedly: For example, a small fact table in Hive that is often used for joins is a good candidate for caching. Conversely, caching the input of a once-yearly reporting query is probably less useful, since the historical data might only be read once.
- Mixed workloads with performance SLAs: Caching the working set of a high priority workload ensures that it does not compete with low priority workloads for disk I/O.

Centralized cache management architecture

In a centralized cache management, the NameNode is responsible for coordinating all of the DataNode off-heap caches in the cluster. The NameNode periodically receives a cache report from each DataNode. The cache report describes all of the blocks cached on the DataNode. The NameNode manages DataNode caches by piggy-backing cache and uncache commands on the DataNode heartbeat.

The following figure illustrates the centralized cached management architecture.


User asks NN to cache a path (e. g. via Hive DDL)
NN translates path to a set of blocks, adds to pending caching queue
User
NameNode
Cache commands piggybacked on heartbeat response
DN heartbeats contain cache block report
DN
DN
DFSClients can be scheduled for memory locality
DFSClient

The NameNode queries its set of cache directives to determine which paths should be cached. Cache directives are persistently stored in the fsimage and edit logs, and can be added, removed, and modified through Java and command-line APIs. The NameNode also stores a set of cache pools, which are administrative entities used to group cache directives together for resource management, and to enforce permissions.

The NameNode periodically re-scans the namespace and active cache directives to determine which blocks need to be cached or uncached, and assigns caching work to DataNodes. Re-scans can also be triggered by user actions such as adding or removing a cache directive or removing a cache pool.

Cache blocks that are under construction, corrupt, or otherwise incomplete are not cached. If a Cache directive covers a symlink, the symlink target is not cached.

Caching can only be applied to directories and files.

Related reference

Caching terminology

Caching terminology

A cache directive defines the path to cache while a cache pool manages groups of cache directives.

Cache directive

Defines the path to be cached. Paths can point either directories or files. Directories are cached non-recursively, meaning only files in the first-level listing of the directory will be cached.

Cache directives also specify additional parameters, such as the cache replication factor and expiration time. The replication factor specifies the number of block replicas to cache. If multiple cache directives refer to the same file, the maximum cache replication factor is applied.

The expiration time is specified on the command line as a time-to-live (TTL), which represents a relative expiration time in the future. After a cache directive expires, it is no longer taken into consideration by the NameNode when making caching decisions.

Cache pool

An administrative entity that manages groups of cache cirectives. Cache pools have UNIX-like permissions that restrict which users and groups have access to the pool. Write permissions allow users to add and remove cache directives to the pool. Read permissions allow users to list the Cache Directives in a pool, as well as additional metadata. Execute permissions are unused.

Cache pools are also used for resource management. Cache pools can enforce a maximum memory limit, which restricts the aggregate number of bytes that can be cached by directives in the pool. Normally, the sum of the pool limits will approximately equal the amount of aggregate memory reserved for HDFS caching on the cluster. Cache pools also track a number of statistics to help cluster users track what is currently cached, and to determine what else should be cached.

Cache pools can also enforce a maximum time-to-live. This restricts the maximum expiration time of directives being added to the pool.

Related Concepts

Centralized cache management architecture

Properties for configuring centralized caching

You must enable JNI to use centralized caching. In addition, you must configure various properties and consider the locked memory limit for configuring centralized caching.

Native libraries

In order to lock block files into memory, the DataNode relies on native JNI code found in libhadoop.so. Be sure to enable JNI if you are using HDFS centralized cache management.

Configuration properties

Configuration properties for centralized caching are specified in the hdfs-site.xml file.

Required properties

Only the following property is required:

- dfs.datanode.max.locked.memory This property determines the maximum amount of memory (in bytes) that a DataNode will use for caching. The "locked-in-memory size" ulimit (ulimit -l) of the DataNode user also needs to be increased to exceed this parameter (for more details, see the following section on). When setting this value, remember that you will need space in memory for other things as well, such as the DataNode and application JVM heaps, and the operating system page cache. Example:

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>268435456</value>
</property>
```

Optional Properties

The following properties are not required, but can be specified for tuning.

- dfs.namenode.path.based.cache.refresh.interval.ms The NameNode will use this value as the number of milliseconds between subsequent cache path re-scans. By default, this parameter is set to 300000, which is five minutes. Example:

```
<property>
  <name>dfs.namenode.path.based.cache.refresh.interval.ms</name>
  <value>300000</value>
</property>
```

- dfs.time.between.resending.caching.directives.ms The NameNode will use this value as the number of milliseconds between resending caching directives. Example:

```
<property>
  <name>dfs.time.between.resending.caching.directives.ms</name>
  <value>300000</value>
</property>
```

- dfs.datanode.fsdatasetcache.max.threads.per.volume The DataNode will use this value as the maximum number of threads per volume to use for caching new data. By default, this parameter is set to 4. Example:

```
<property>
  <name>dfs.datanode.fsdatasetcache.max.threads.per.volume</name>
  <value>4</value>
</property>
```

- dfs.cachereport.intervalMsec The DataNode will use this value as the number of milliseconds between sending a full report of its cache state to the NameNode. By default, this parameter is set to 10000, which is 10 seconds. Example:

```
<property>
  <name>dfs.cachereport.intervalMsec</name>
  <value>10000</value>
</property>
```

- dfs.namenode.path.based.cache.block.map.allocation.percent The percentage of the Java heap that will be allocated to the cached blocks map. The cached blocks map is a hash map that uses chained hashing. Smaller maps may be accessed more slowly if the number of cached blocks is large. Larger maps will consume more memory. The default value is 0.25 percent. Example:

```
<property>
  <name>dfs.namenode.path.based.cache.block.map.allocation.percent</name>
  <value>0.25</value>
</property>
```

OS limits

If you get the error "Cannot start datanode because the configured max locked memory size...is more than the datanode's available RLIMIT_MEMLOCK ulimit," this means that the operating system is imposing a lower limit on the amount of memory that you can lock than what you have configured. To fix this, you must adjust the ulimit -l value that the DataNode runs with. This value is usually configured in /etc/security/limits.conf, but this may vary depending on what operating system and distribution you are using.

You have correctly configured this value when you can run ulimit - l from the shell and get back either a higher value than what you have configured or the string "unlimited", which indicates that there is no limit. Typically, ulimit -l returns the memory lock limit in kilobytes (KB), but dfs.datanode.max.locked.memory must be specified in bytes.

For example, if the value of dfs.datanode.max.locked.memory is set to 128000 bytes:

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>128000</value>
</property>
```

Set the memlock (max locked-in-memory address space) to a slightly higher value. For example, to set memlock to 130 KB (130,000 bytes) for the hdfs user, you would add the following line to /etc/security/limits.conf.

```
hdfs - memlock 130
```

Commands for using cache pools and directives

You can use the Command-Line Interface (CLI) to create, modify, and list cache pools and cache directives using the hdfs cacheadmin subcommand.

Cache Directives are identified by a unique, non-repeating, 64-bit integer ID. IDs will not be reused even if a Cache Directive is removed.

Cache Pools are identified by a unique string name.

You must first create a Cache Pool, and then add Cache Directives to the Cache Pool.

Cache Pool Commands

- addPool -- Adds a new Cache Pool.

  Usage:

  ```
  hdfs cacheadmin -addPool <name> [-owner <owner>] [-group <group>]
  [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]
  ```

  Options:

Table 5: Cache Pool Add Options

Option	Description
<name>	The name of the pool.
<owner>	The user name of the owner of the pool. Defaults to the current user.
<group>	The group that the pool is assigned to. Defaults to the primary group name of the current user.
<mode>	The UNIX-style permissions assigned to the pool. Permissions are specified in octal (e.g. 0755). Pool permissions are set to 0755 by default.

Option	Description
<limit>	The maximum number of bytes that can be cached by directives in the pool, in aggregate. By default, no limit is set.
<maxTtl>	The maximum allowed time-to-live for directives being added to the pool. This can be specified in seconds, minutes, hours, and days (e.g. 120s, 30m, 4h, 2d). Valid units are [smhd]. By default, no maximum is set. A value of "never" specifies that there is no limit.

- modifyPool -- Modifies the metadata of an existing Cache Pool.

  Usage:

  ```
  hdfs cacheadmin -modifyPool <name> [-owner <owner>] [-group <group>]
  [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]
  ```

  Options:

  Table 6: Cache Pool Modify Options

Option	Description
<name>	The name of the pool to modify.
<owner>	The user name of the owner of the pool.
<group>	The group that the pool is assigned to.
<mode>	The UNIX-style permissions assigned to the pool. Permissions are specified in octal (e.g. 0755).
<limit>	The maximum number of bytes that can be cached by directives in the pool, in aggregate.
<maxTtl>	The maximum allowed time-to-live for directives being added to the pool. This can be specified in seconds, minutes, hours, and days (e.g. 120s, 30m, 4h, 2d). Valid units are [smdh]. By default, no maximum is set. A value of "never" specifies that there is no limit.

- removePool -- Removes a Cache Pool. This command also "un-caches" paths that are associated with the pool.

  Usage:

  ```
  hdfs cacheadmin -removePool <name>
  ```

  Options:

  Table 7: Cache Pool Remove Options

Option	Description
<name>	The name of the Cache Pool to remove.

- listPools -- Displays information about one or more Cache Pools, such as name, owner, group, permissions, and so on.

  Usage:

  ```
  hdfs cacheadmin -listPools [-stats] [<name>]
  ```

  Options:

Table 8: Cache Pools List Options

Option	Description
-stats	Displays additional Cache Pool statistics.
<name>	If specified, lists only the named Cache Pool.

- help -- Displays detailed information about a command.

  Usage:

  ```
  hdfs cacheadmin -help <command-name>
  ```

  Options:

Table 9: Cache Pool Help Options

Option	Description
<command-name	Displays detailed information for the specified command name. If no command name is specified, detailed help is displayed for all commands.

Cache Directive Commands

- addDirective -- Adds a new Cache Directive.

  Usage:

  ```
  hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force]
  [-replication <replication>] [-ttl <time-to-live>]
  ```

  Options:

Table 10: Cache Pool Add Directive Options

Option	Description
<path>	The path to the cache directory or file.
<pool-name>	The Cache Pool to which the Cache Directive will be added. You must have Write permission for the Cache Pool in order to add new directives.
-force	Skips checking of the Cache Pool resource limits.
<replication>	The cache replication factor to use. Default setting is 1.
<time-to-live>	How long the directive is valid. This can be specified in minutes, hours and days (e.g. 30m, 4h, 2d). Valid units are [smdh]. A value of "never" indicates a directive that never expires. If unspecified, the directive never expires.

- removeDirective -- Removes a Cache Directive.

  Usage:

  ```
  hdfs cacheadmin -removeDirective <id>
  ```

  Options:

Table 11: Cache Pool Remove Directive Options

Option	Description
<id>	The ID of the Cache Directive to remove. You must have Write permission for the pool that the directive belongs to in order to remove it. You can use the -listDirectives command to display a list of Cache Directive IDs.

- removeDirectives -- Removes all of the Cache Directives in a specified path.

  Usage:

  ```
  hdfs cacheadmin -removeDirectives <path>
  ```

  Options:

  Table 12: Cache Pool Remove Directives Options

Option	Description
<path>	The path of the Cache Directives to remove. You must have Write permission for the pool that the directives belong to in order to remove them. You can use the -listDirectives command to display a list of Cache Directives.

- listDirectives -- Returns a list of Cache Directives.

  Usage:

  ```
  hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]
  ```

  Options:

  Table 13: Cache Pools List Directives Options

Option	Description
<path>	Lists only the Cache Directives in the specified path. If there is a Cache Directive in the <path> that belongs to a Cache Pool for which you do not have Read access, it will not be listed.
<pool>	Lists on the Cache Directives in the specified Cache Pool.
-stats	Lists path-based Cache Directive statistics.

Configuring HDFS rack awareness

The NameNode in an HDFS cluster maintains rack IDs of all the DataNodes. The NameNode uses this information about the distribution of DataNodes among various racks in the cluster to select the closer DataNodes for effective block placement during read/write operations. This concept of selecting the closer DataNodes based on their location in the cluster is termed as rack awareness. Rack awareness helps in maintaining fault tolerance in the event of a failure.

About this task

Configuring rack awareness on an HDP cluster involves creating a rack topology script, adding the script to core-site.xml, restarting HDFS, and verifying the rack awareness.

Create a rack topology script

HDFS uses topology scripts to determine the rack location of nodes and uses this information to replicate block data to redundant racks.

Procedure

1. Create an executable topology script and a topology data file.

   Consider the following examples:

   The following is an example topology script named rack-topology.sh.

```
#!/bin/bash
# Adjust/Add the property "net.topology.script.file.name"
# to core-site.xml with the "absolute" path the this
# file. ENSURE the file is "executable".

# Supply appropriate rack prefix
RACK_PREFIX=default

# To test, supply a hostname as script input:
if [ $# -gt 0 ]; then

CTL_FILE=${CTL_FILE:-"rack_topology.data"}

HADOOP_CONF=${HADOOP_CONF:-"/etc/hadoop/conf"}

if [ ! -f ${HADOOP_CONF}/${CTL_FILE} ]; then
 echo -n "/$RACK_PREFIX/rack "
 exit 0
fi

while [ $# -gt 0 ] ; do
 nodeArg=$1
 exec< ${HADOOP_CONF}/${CTL_FILE}
 result=""
 while read line ; do
 ar=( $line )
 if [ "${ar[0]}" = "$nodeArg" ] ; then
 result="${ar[1]}"
 fi
 done
 shift
 if [ -z "$result" ] ; then
 echo -n "/$RACK_PREFIX/rack "
 else
 echo -n "/$RACK_PREFIX/rack_$result "
 fi
done

else
 echo -n "/$RACK_PREFIX/rack "
fi
```

   The following is an example topology data file named rack_topology.data.

```
# This file should be:
# - Placed in the /etc/hadoop/conf directory
# - On the Namenode (and backups IE: HA, Failover, etc)
# - On the Job Tracker OR Resource Manager (and any Failover JT's/RM's)
# This file should be placed in the /etc/hadoop/conf directory.
# Add Hostnames to this file. Format <host ip> <rack_location>
192.168.2.10 01
192.168.2.11 02
192.168.2.12 03
```

2. Copy the topology script and the data file to the /etc/hadoop/conf directory on all cluster nodes.
3. Run the topology script to ensure that it returns the correct rack information for each host.

Add the topology script property to core-site.xml

Assign the name of the topology script to the net.topology.script.file.name property in core-site.xml.

Procedure

1. Stop the HDFS cluster.
2. Add the topology script file to core-site.xml.

   In the following example, the value of the property net.topology.script.file.name is the name of the topology script file.

   ```
   <property>
     <name>net.topology.script.file.name</name>
     <value>/etc/hadoop/conf/rack-topology.sh</value>
   </property>
   ```

   By default, the topology script processes up to 100 requests per invocation. You can specify a number other than the default value with the net.topology.script.number.args property, as shown in the following example:

   ```
   <property>
     <name>net.topology.script.number.args</name>
     <value>75</value>
   </property>
   ```

Restart HDFS and MapReduce services

After adding the topology script property to core-site.xml, you must restart the HDFS and MapReduce services.

Verify rack awareness

You must perform a series of checks to verify if rack awareness is activated on the cluster.

Procedure

1. Check the NameNode logs located in /var/log/hadoop/hdfs/ for the addition of nodes.

   ```
   014-01-13 15:58:08,495 INFO org.apache.hadoop.net.NetworkTopology: Adding
                               a new node: /rack01/<ipaddress>
   ```

2. Run the hdfs fsck command to ensure that there are no inconsistencies.
   For a cluster with two racks, the fsck command returns a status similar to the following:

```
Status: HEALTHY  Total size: 123456789 B  Total dirs: 0  Total files: 1
 Total blocks (validated): 1 (avg. block size 123456789 B)
 Minimally replicated blocks: 1 (100.0 %)  Over-replicated blocks: 0 (0.0 %)
  Under-replicated blocks: 0 (0.0 %)  Mis-replicated blocks: 0 (0.0 %)
 Default replication factor: 3  Average block replication: 3.0  Corrupt
 blocks: 0  Missing replicas: 0 (0.0 %)  Number of data-nodes: 40  Number of
 racks: 2  FSCK ended at Mon Jan 13 17:10:51 UTC 2014 in 1 milliseconds
```

3. Run the dfsadmin -report command for a report that includes the rack name next to each DataNode.
   The dfsadmin -report command returns a report similar to the following excerpted example:

   ```
   Configured Capacity: 19010409390080 (17.29 TB) Present Capacity:
    18228294160384 (16.58 TB) DFS Remaining: 5514620928000 (5.02 TB) DFS
    Used: 12713673232384 (11.56 TB) DFS Used%: 69.75% Under replicated
    blocks: 181 Blocks with corrupt replicas: 0  Missing blocks: 0
    -------------------------------------------------  Datanodes available:
    5 (5 total, 0 dead)  Name: 192.168.90.231:50010 (h2d1.hdp.local)
    Hostname: h2d1.hdp.local Rack: /default/rack_02 Decommission Status :
    Normal Configured Capacity: 15696052224 (14.62 GB) DFS Used: 314380288
   (299.82 MB) Non DFS Used: 3238612992 (3.02 GB) DFS Remaining: 12143058944
    (11.31 GB) DFS Used%: 2.00% DFS Remaining%: 77.36%
    Configured Cache Capacity: 0 (0 B) Cache Used: 0 (0 B) Cache Remaining: 0
    (0 B) Cache Used%: 100.00% Cache Remaining%: 0.00% Last contact: Thu Jun
    12 11:39:51 EDT 2014
   ```

Customizing HDFS

You can use the dfs.user.home.base.dir property to customize the HDFS home directory. In addition, you can configure properties to control the size of the directory that holds the NameNode edits directory.

Customize the HDFS home directory

By default, the HDFS home directory is set to /user/<user_name>. Use the dfs.user.home.base.dir property to customize the HDFS home directory.

Procedure

In hdfs-site.xml file, set the value of the dfs.user.home.base.dir property.

```
<property>
  <name>dfs.user.home.base.dir</name>
  <value>/user</value>
  <description>Base directory of user home.</description>
</property>
```

In the example, <value> is the path to the new home directory.

Properties to set the size of the NameNode edits directory

You can configure the dfs.namenode.num.checkpoints.retained and dfs.namenode.num.extra.edits.retained properties to control the size of the directory that holds the NameNode edits directory.

- dfs.namenode.num.checkpoints.retained: The number of image checkpoint files that are retained in storage directories. All edit logs necessary to recover an up-to-date namespace from the oldest retained checkpoint are also retained.
- dfs.namenode.num.extra.edits.retained: The number of extra transactions that should be retained beyond what is minimally necessary for a NameNode restart. This can be useful for audit purposes, or for an HA setup where a remote Standby Node may have been offline for some time and require a longer backlog of retained edits in order to start again.

Optimizing NameNode disk space with Hadoop archives

Hadoop Archives (HAR) are special format archives that efficiently pack small files into HDFS blocks.

The Hadoop Distributed File System (HDFS) is designed to store and process large data sets, but HDFS can be less efficient when storing a large number of small files. When there are many small files stored in HDFS, these small files occupy a large portion of the namespace. As a result, disk space is under-utilized because of the namespace limitation.

Hadoop Archives (HAR) can be used to address the namespace limitations associated with storing many small files. A Hadoop Archive packs small files into HDFS blocks more efficiently, thereby reducing NameNode memory usage while still allowing transparent access to files. Hadoop Archives are also compatible with MapReduce, allowing transparent access to the original files by MapReduce jobs.

Overview of Hadoop archives

Storing a large number of small files in HDFS leads to inefficient utilization of space – the namespace is overutilized while the disk space might be underutilized. Hadoop Archives (HAR) address this limitation by efficiently packing small files into large files without impacting the file access.

The Hadoop Distributed File System (HDFS) is designed to store and process large (terabytes) data sets. For example, a large production cluster may have 14 PB of disk space and store 60 million files.

However, storing a large number of small files in HDFS is inefficient. A file is generally considered to be "small" when its size is substantially less than the HDFS block size, which is 256 MB by default in HDP. Files and blocks are name objects in HDFS, meaning that they occupy namespace (space on the NameNode). The namespace capacity of the system is therefore limited by the physical memory of the NameNode.

When there are many small files stored in the system, these small files occupy a large portion of the namespace. As a consequence, the disk space is underutilized because of the namespace limitation. In one real-world example, a production cluster had 57 million files less than 256 MB in size, with each of these files taking up one block on the NameNode. These small files used up 95% of the namespace but occupied only 30% of the cluster disk space.

Hadoop Archives (HAR) can be used to address the namespace limitations associated with storing many small files. HAR packs a number of small files into large files so that the original files can be accessed transparently (without expanding the files).

HAR increases the scalability of the system by reducing the namespace usage and decreasing the operation load in the NameNode. This improvement is orthogonal to memory optimization in the NameNode and distributing namespace management across multiple NameNodes.

Hadoop Archive is also compatible with MapReduce — it allows parallel access to the original files by MapReduce jobs.

Hadoop archive components

You can use the Hadoop archiving tool to create Hadoop Archives (HAR). The Hadoop Archive is integrated with the Hadoop file system interface. Files in a HAR are exposed transparently to users. File data in a HAR is stored in multipart files, which are indexed to retain the original separation of data.

Hadoop archiving tool

Hadoop Archives can be created using the Hadoop archiving tool. The archiving tool uses MapReduce to efficiently create Hadoop Archives in parallel. The tool can be invoked using the command:

```
hadoop archive -archiveName name -p <parent> <src>* <dest>
```

A list of files is generated by traversing the source directories recursively, and then the list is split into map task inputs. Each map task creates a part file (about 2 GB, configurable) from a subset of the source files and outputs the metadata. Finally, a reduce task collects metadata and generates the index files.

HAR file system

Most archival systems, such as tar, are tools for archiving and de-archiving. Generally, they do not fit into the actual file system layer and hence are not transparent to the application writer in that the archives must be expanded before use.

The Hadoop Archive is integrated with the Hadoop file system interface. The HarFileSystem implements the FileSystem interface and provides access via the har:// scheme. This exposes the archived files and directory tree structures transparently to users. Files in a HAR can be accessed directly without expanding them.

For example, if we have the following command to copy an HDFS file to a local directory:

```
hdfs dfs –get hdfs://namenode/foo/file-1 localdir
```

Suppose a Hadoop Archive bar.har is created from the foo directory. With the HAR, the command to copy the original file becomes:

```
hdfs dfs –get har://namenode/bar.har/foo/file-1 localdir
```

Users only need to change the URI paths. Alternatively, users may choose to create a symbolic link (from hdfs://namenode/foo to har://namenode/bar.har/foo in the example above), and then even the URIs do not need to be changed. In either case, HarFileSystem will be invoked automatically to provide access to the files in the HAR. Because of this transparent layer, HAR is compatible with the Hadoop APIs, MapReduce, the FS shell command-line interface, and higher-level applications such as Pig, Zebra, Streaming, Pipes, and DistCp.

HAR format data model

The Hadoop Archive data format has the following layout:

```
foo.har/_masterindex //stores hashes and offsets
foo.har/_index //stores file statuses
foo.har/part-[1..n] //stores actual file data
```

The file data is stored in multipart files, which are indexed in order to retain the original separation of data. Moreover, the file parts can be accessed in parallel by MapReduce programs. The index files also record the original directory tree structures and file status.

Create a Hadoop archive

Use the hadoop archive command to invoke the Hadoop archiving tool.

Procedure

Run the hadoop archive command by specifying the archive name to create, the parent directory relative to the archive location, the source files to archive, and the destination archive location.

```
hadoop archive -archiveName name -p <parent> <src>* <dest>
```

The archive name must have a .har extension

Note:

- Archiving does not delete the source files. If you want to delete the input files after creating an archive to reduce namespace, you must manually delete the source files.
- Although the hadoop archive command can be run from the host file system, the archive file is created in the HDFS file system from directories that exist in HDFS. If you reference a directory on the host file system and not HDFS, the system displays the following error:

```
The resolved paths set is empty. Please check whether the srcPaths
 exist, where srcPaths
 = [</directory/path>]
```

Example

Consider the following example of archiving two files:

```
hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo
```

This example creates an archive using /user/hadoop as the relative archive directory. The directories /user/hadoop/dir1 and /user/hadoop/dir2 will be archived in the /user/zoo/foo.har archive.

List files in Hadoop archives

Use the hdfs dfs -ls command to list files in Hadoop archives.

Procedure

Run the hdfs dfs -ls command by specifying the archive directory location.
To specify the directories in an archive directory foo.har located in /usr/zoo, run the following command:

```
hdfs dfs -ls har:///user/zoo/foo.har/
```

Assuming the archive directory foo.har contains two directories dir1 and dir2, the command returns the following

```

har:///user/zoo/foo.har/dir1
har:///user/zoo/foo.har/dir2
```

Note:

Consider an archive created using the following command:

```
hadoop archive -archiveName foo.har -p /user/ hadoop/dir1 hadoop/dir2 /
user/zoo
```

If you list the files of the archive created in the preceding command, the command returns the following:

```
har:///user/zoo/foo.har/hadoop
har:///user/zoo/foo.har/hadoop/dir1
har:///user/zoo/foo.har/hadoop/dir2
```

Note that the modified parent argument causes the files to be archived relative to /user/.

Format for using Hadoop archives with MapReduce

To use Hadoop Archives with MapReduce, you must reference files differently than you would with the default file system. If you have a Hadoop Archive stored in HDFS in /user/zoo/foo.har, you must specify the input directory as har:///user/zoo/foo.har to use it as a MapReduce input.

Because Hadoop Archives are exposed as a file system, MapReduce can use all of the logical input files in Hadoop Archives as input.

Detecting slow DataNodes

Slow DataNodes in an HDFS cluster can negatively impact the cluster performance. Therefore, HDFS provides a mechanism to detect and report slow DataNodes that have a negative impact on the performance of the cluster.

HDFS is designed to detect and recover from complete failure of DataNodes:

- There is no single point of failure.
- Automatic NameNode failover takes only a few seconds.

- Because data replication can be massively parallelized in large clusters, recovery from DataNode loss occurs within minutes.
- Most jobs are not affected by DataNode failures.

However, partial failures can negatively affect the performance of running DataNodes:

- Slow network connection due to a failing or misconfigured adapter.
- Bad OS or JVM settings that affect service performance.
- Slow hard disk.
- Bad disk controller.

Slow DataNodes can have a significant impact on cluster performance. A slow DataNode may continue sending heartbeats successfully, and the NameNode will keep redirecting clients to slow DataNodes. HDFS DataNode monitoring provides detection and reporting of slow DataNodes that negatively affect cluster performance.

Enable disk IO statistics

Disk IO statistics are disabled by default. To enable disk IO statistics, you must set the file IO sampling percentage to a non-zero value in the hdfs-site.xml file.

Procedure

1. Set the dfs.datanode.fileio.profiling.sampling.percentage property to a non-zero value in hdfs-site.xml.

```
<property>
  <name>dfs.datanode.fileio.profiling.sampling.fraction</name>
  <value>25</value>
</property>
```

Note: Sampling disk IO might have a minor impact on cluster performance.

2. Access the disk IO statistics from the NameNode JMX page at http://<namenode_host>:50070/jmx.
In the following JMX output example, the time unit is milliseconds, and the disk is healthy because the IO latencies are low:

```
    "name" : "Hadoop:service=DataNode,name=DataNodeVolume-/data/disk2/dfs/
data/",
    "modelerType" : "DataNodeVolume-/data/disk2/dfs/data/",
    "tag.Context" : "dfs",
    "tag.Hostname" : "n001.hdfs.example.com",
    "TotalMetadataOperations" : 67,
    "MetadataOperationRateAvgTime" : 0.08955223880597014,
  ...
    "WriteIoRateNumOps" : 7321,
    "WriteIoRateAvgTime" : 0.050812730501297636
```

Enable detection of slow DataNodes

When slow DataNode detection is enabled, DataNodes collect latency statistics on their peers during write pipelines, and use periodic outlier detection to determine slow peers. The NameNode aggregates reports from all DataNodes and flags potentially slow nodes. Slow DataNode detection is disabled by default.

Procedure

1. To enable slow DataNode detection, set the value of the dfs.datanode.peer.stats.enabled property to true in hdfs-site.xml.

```
<property>
  <name>dfs.datanode.peer.stats.enabled</name>
  <value>true</value>
</property>
```

2. Access the slow DataNode statistics either from the NameNode JMX page at http://<namenode_host>:50070/jmx or from the DataNode JMX page at http://<datanode_host>:50075/jmx.
   In the following JMX output example, the time unit is milliseconds, and the peer DataNodes are healthy because the latencies are in milliseconds:

```
"name" : "Hadoop:service=DataNode,name=DataNodeInfo",
"modelerType" : "org.apache.hadoop.hdfs.server.datanode.DataNode",
 "SendPacketDownstreamAvgInfo" : "{
        \"[192.168.7.202:50075]RollingAvgTime\" : 1.4476967370441458,
        \"[192.168.7.201:50075]RollingAvgTime\" : 1.5569170444798432
}"
```

Allocating DataNode memory as storage (Technical Preview)

HDFS supports efficient writes of large data sets to durable storage, and also provides reliable access to the data. This works well for batch jobs that write large amounts of persistent data. Emerging classes of applications are driving use cases for writing smaller amounts of temporary data. Using DataNode memory as storage addresses the use case of applications that want to write relatively small amounts of intermediate data sets with low latency.

Note: This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at https://support.hortonworks.com.

Writing block data to memory reduces durability, as data can be lost due to process restart before it is saved to disk. HDFS attempts to save replica data to disk in a timely manner to reduce the window of possible data loss.

DataNode memory is referenced using the RAM_DISK storage type and the LAZY_PERSIST storage policy.

HDFS storage types

HDFS storage types can be used to assign data to different types of physical storage media.

The following storage types are available:

- DISK: Disk drive storage (default storage type)
- ARCHIVE: Archival storage (high storage density, low processing resources)
- SSD: Solid State Drive
- RAM_DISK: DataNode Memory

If no storage type is assigned, DISK is used as the default storage type.

LAZY_PERSIST memory storage policy

Use the LAZY_PERSIST storage policy to store data blocks on the configured DataNode memory.

For LAZY_PERSIST, the first replica is stored on RAM_DISK (DataNode memory), and the remaining replicas are stored on DISK. The fallback storage for both creation and replication is DISK.

The following table summarizes these replication policies:

Policy ID	Policy Name	Block Placement (for n replicas)	Fallback storage for creation	Fallback storage for replication
15	LAZY_PERSIST	RAM_DISK: 1, DISK:n-1	DISK	DISK

Configure DataNode memory as storage

Configuring memory on a DataNode as storage requires you to shut down the particular DataNode, set RAM_DISK as the storage type, set the LAZY_PERSIST storage policy to store data, and then start the DataNode.

Procedure

1. Shut down the DataNode.
2. Use required mount commands to allocate a certain portion of the DataNode memory as storage.
   The following example shows how you can allocate 2GB memory for use by HDFS.

   ```
   sudo mkdir -p /mnt/hdfsramdisk
   sudo mount -t tmpfs -o size=2048m tmpfs /mnt/hdfsramdisk
   sudo mkdir -p /usr/lib/hadoop-hdfs
   ```
3. Assign the RAM_DISK storage type to ensure that HDFS can assign data to the DataNode memory configured as storage.

   To specify the DataNode as RAM_DISK storage, insert [RAM_DISK] at the beginning of the local file system mount path and add it to the dfs.name.dir property in hdfs-default.xml.

   The following example shows the updated mount path values for dfs.datanode.data.dir

   ```
   <property>
     <name>dfs.datanode.data.dir</name>
     <value>file:///grid/3/aa/hdfs/data/,[RAM_DISK]file:///mnt/hdfsramdisk/</
   value>
   </property>
   ```
4. Set the LAZY_PERSIST storage policy to store data on the configured DataNode memory.
   The following example shows how you can use the hdfs dfsadmin -getStoragepolicy command to configure the LAZY_PERSIST storage policy:

   ```
   hdfs dfsadmin -getStoragePolicy /memory1 LAZY_PERSIST
   ```

   Note: When you update a storage policy setting on a file or directory, the new policy is *not* automatically enforced. You must use the HDFS mover data migration tool to actually move blocks as specified by the new storage policy.
5. Start the DataNode.
6. Use the HDFS mover tool to move data blocks according to the specified storage policy.

   The HDFS mover data migration tool scans the specified files in HDFS and verifies if the block placement satisfies the storage policy. For the blocks that violate the storage policy, the tool moves the replicas to a different storage type in order to fulfill the storage policy requirements.

Improving performance with short-circuit local reads

In HDFS, reads normally go through the DataNode. Thus, when a client asks the DataNode to read a file, the DataNode reads that file off of the disk and sends the data to the client over a TCP socket. "Short-circuit" reads bypass the DataNode, allowing the client to read the file directly. This is only possible in cases where the client is co-located with the data. Short-circuit reads provide a substantial performance boost to many applications.

Prerequisites for configuring short-ciruit local reads

To configure short-circuit local reads, you must enable libhadoop.so.

See the Native Libraries Guide for details on enabling this library.

Related Information

Native Libraries Guide

Properties for configuring short-circuit local reads on HDFS

To configure short-circuit local reads, you must add various properties to the hdfs-site.xml file. Short-circuit local reads must be configured on both the DataNode and the client.

Property Name	Property Value	Description
dfs.client.read.shortcircuit	true	Set this to true to enable short-circuit local reads.
dfs.domain.socket.path	/var/lib/hadoop-hdfs/ dn_socket	The path to the domain socket. Short-circuit reads make use of a UNIX domain socket. This is a special path in the file system that allows the client and the DataNodes to communicate. You will need to set a path to this socket. The DataNode needs to be able to create this path. On the other hand, it should not be possible for any user except the hdfs user or root to create this path. For this reason, paths under /var/run or /var/lib are often used. In the file system that allows the client and the DataNodes to communicate. You will need to set a path to this socket. The DataNode needs to be able to create this path. On the other hand, it should not be possible for any user except the hdfs user or root to create this path. For this reason, paths under /var/run or /var/lib are often used.
dfs.client.domain.socket.data.traffic	false	This property controls whether or not normal data traffic will be passed through the UNIX domain socket. This feature has not been certified with HDP releases, so it is recommended that you set the value of this property to false. Abnormal data traffic will be passed through the UNIX domain socket.
dfs.client.use.legacy.blockreader.local	false	Setting this value to false specifies that the new version (based on HDFS-347) of the short-circuit reader is used. This new new short-circuit reader implementation is supported and recommended for use with HDP. Setting this value to true would mean that the legacy short-circuit reader would be used.
dfs.datanode.hdfs-blocks-metadata.enabled	true	Boolean which enables back-end DataNode-side support for the experimental DistributedFileSystem#getFile VBlockStorageLocationsAPI.
dfs.client.file-block-storage-locations.timeout	60	Timeout (in seconds) for the parallel RPCs made in DistributedFileSystem #getFileBlockStorageLocations(). This property is deprecated but is still supported for backward compatibility

Property Name	Property Value	Description
dfs.client.file-block-storage-locations.timeout.millis	60000	Timeout (in milliseconds) for the parallel RPCs made in DistributedFileSystem #getFileBlockStorageLocations(). This property replaces dfs.client.file-block-storage-locations.timeout, and offers a finer level of granularity.
dfs.client.read.shortcircuit.skip.checksum	false	If this configuration parameter is set, short-circuit local reads will skip checksums. This is normally not recommended, but it may be useful for special setups. You might consider using this if you are doing your own checksumming outside of HDFS.
dfs.client.read.shortcircuit.streams.cache.size	256	The DFSClient maintains a cache of recently opened file descriptors. This parameter controls the size of that cache. Setting this higher will use more file descriptors, but potentially provide better performance on workloads involving many seeks.
dfs.client.read.shortcircuit.streams.cache.expiry.ms	300000	This controls the minimum amount of time (in milliseconds) file descriptors need to sit in the client cache context before they can be closed for being inactive for too long.

The XML for these entries:

```
<configuration>

<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
</property>

<property>
  <name>dfs.domain.socket.path</name>
  <value>/var/lib/hadoop-hdfs/dn_socket</value>
</property>

<property>
  <name>dfs.client.domain.socket.data.traffic</name>
  <value>false</value>
</property>

<property>
  <name>dfs.client.use.legacy.blockreader.local</name>
  <value>false</value>
</property>

<property>
  <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
  <value>true</value>
</property>

<property>
  <name>dfs.client.file-block-storage-locations.timeout.millis</name>
  <value>60000</value>
</property>
```

```
<property>
  <name>dfs.client.read.shortcircuit.skip.checksum</name>
  <value>false</value>
</property>

<property>
  <name>dfs.client.read.shortcircuit.streams.cache.size</name>
  <value>256</value>
</property>

<property>
  <name>dfs.client.read.shortcircuit.streams.cache.expiry.ms</name>
  <value>300000</value>
</property>

</configuration>
```

Using the NFS Gateway for accessing HDFS

The NFS Gateway for HDFS allows clients to mount HDFS and interact with it through NFS, as if it were part of their local file system. The Gateway supports NFSv3.

After mounting HDFS, a client user can perform the following tasks:

- Browse the HDFS file system through their local file system on NFSv3 client-compatible operating systems.
- Upload and download files between the HDFS file system and their local file system.
- Stream data directly to HDFS through the mount point. File append is supported, but random write is not supported.

Prerequisites for using NFS Gateway

- The NFS Gateway machine must be running all components that are necessary for running an HDFS client, such as a Hadoop core JAR file and a HADOOP_CONF directory.
- The NFS Gateway can be installed on any DataNode, NameNode, or HDP client machine. Start the NFS server on that machine.

Configure the NFS Gateway

You must ensure that the proxy user for the NFS Gateway can proxy all the users accessing the NFS mounts. In addition, you must configure settings specific to the Gateway.

Procedure

1. Ensure that the proxy user for the NFS Gateway can proxy all the users accessing the NFS mounts.

   In non-secure mode, the user running the Gateway is the proxy user, while in secure mode the user in Kerberos keytab is the proxy user.

   If a user nfsserver is running the Gateway and there are users belonging to groups nfs-users1 and nfs-users2, then set the following values in core-site.xml on the NameNode.

   Note:

   Replace nfsserver with the user account that starts the Gateway in your cluster.

   ```
   <property>
     <name>hadoop.proxyuser.nfsserver.groups</name>
     <value>nfs-users1,nfs-users2</value>
   ```

```
  <description>
    The 'nfsserver' user is allowed to proxy all members of the
    'nfs-users1' and 'nfs-users2' groups. Set this to '*' to allow
    nfsserver user to proxy any group.
  </description>
</property>

<property>
  <name>hadoop.proxyuser.nfsserver.hosts</name>
  <value>nfs-client-host1.com</value>
  <description>
    This is the host where the nfs gateway is running. Set this to
    '*' to allow requests from any hosts to be proxied.
  </description>
</property>
```

For a Kerberized cluster, set the following properties in hdfs-site.xml:

```
<property>
  <name>dfs.nfsgateway.keytab.file</name>
  <value>/etc/hadoop/conf/nfsserver.keytab</value> <!-- path to the
      nfs gateway keytab -->
</property>

<property>
  <name>dfs.nfsgateway.kerberos.principal</name>
  <value>nfsserver/_HOST@YOUR-REALM.COM</value>
</property>
```

2. Configure settings for the NFS Gateway.

The NFS Gateway uses the same settings that are used by the NameNode and DataNode. Configure various properties based on your application's requirements:

a) Edit the hdfs-site.xml file on your NFS Gateway machine.

```
<property>
  <name>dfs.namenode.accesstime.precision</name>
  <value>3600000</value>
  <description>
    The access time for HDFS file is precise up to this value.
    The default value is 1 hour. Setting a value of 0 disables
    access times for HDFS.
  </description>
</property>
```

Note: If the export is mounted with access time update allowed, ensure that this property is not disabled in the configuration file. Only the NameNode needs to restart after this property is changed. If you have disabled access time update by mounting with noatime, you do NOT have to change this property nor restart your NameNode.

b) Add the value of the dfs.nfs3.dump.dir property in hdfs-site.xml.

```

<property>
    <name>dfs.nfs3.dump.dir</name>
    <value>/tmp/.hdfs-nfs</value>
</property>
```

Note: The NFS client often reorders writes. Sequential writes can arrive at the NFS Gateway in a random order. This directory is used to temporarily save out-of-order writes before writing to HDFS. Ensure that the directory has enough space. For example, if the application uploads 10 files with each having 100MB, it is recommended for this directory to have 1GB space in case a write reorder happens to every file.

c) Update the value of the dfs.nfs.exports.allowed.hosts property in hdfs-site.xml as specified.

```
<property>
    <name>dfs.nfs.exports.allowed.hosts</name>
    <value>* rw</value>
</property>
```

Note: By default, the export can be mounted by any client. You must update this property to control access. The value string contains the machine name and access privilege, separated by whitespace characters. The machine name can be in single host, wildcard, or IPv4 network format. The access privilege uses rw or ro to specify readwrite or readonly access to exports. If you do not specify an access privilege, the default machine access to exports is readonly. Separate machine entries by ;. For example, 192.168.0.0/22 rw ; host*.example.com ; host1.test.org ro;.

d) Restart the NFS Gateway.

e) Optional: Customize log settings by modifying the log4j.property file.

To change the trace level, add the following:

log4j.logger.org.apache.hadoop.hdfs.nfs=DEBUG

To view more information about ONCRPC requests, add the following:

log4j.logger.org.apache.hadoop.oncrpc=DEBUG

3. Specify JVM heap space (HADOOP_NFS3_OPTS) for the NFS Gateway.

   You can increase the JVM heap allocation for the NFS Gateway using this option. To set this option, specify the following in hadoop-env.sh:

   export HADOOP_NFS3_OPTS=<memory-setting(s)>

   The following example specifies a 2GB process heap (2GB starting size and 2GB maximum):

   ```
   export HADOOP_NFS3_OPTS="-Xms2048m -Xmx2048m"
   ```

4. To improve the performance of large file transfers, you can increase the values of the dfs.nfs.rtmax and dfs.nfs.wtmax properties.

   These properties are configuration settings on the NFS Gateway server that change the maximum read and write request size supported by the Gateway. The default value for both settings is 1MB.

Start and stop the NFS Gateway services

You must start the following daemons to run the NFS services on the Gateway: rpcbind (or portmap), mountd, and nfsd. The NFS Gateway process includes both nfsd and mountd. Although NFS Gateway works with portmap included with most Linux distributions, you must use the portmap included in the NFS Gateway package on some Linux systems such as SLES 11 and RHEL 6.2.

Procedure

1. Stop the nfs/rpcbind/portmap services provided by the platform.

   service nfs stop

   service rpcbind stop

2. Start the included portmap package using one of the following commands: hadoop portmap or hadoop-daemon.sh start portmap.

   Note: You must have root privileges to run these commands.

3. Start mountd and nfsd using one of the following commands: hdfs nfs3 or hadoop-daemon.sh start nfs3.

   Note: hdfs nfs3 starts the services as a foreground process while hadoop-daemon.sh start nfs3 starts the services as a background process.

   No root privileges are required for this command. However, verify that the user starting the Hadoop cluster and the user starting the NFS Gateway are the same.

   Note: If the hadoop-daemon.sh script starts the NFS Gateway, its log file can be found in the hadoop log folder (/var/log/hadoop).

   For example, if you launched the NFS Gateway services as the root user, the log file would be found in a path similar to the following:

   /var/log/hadoop/root/hadoop-root-nfs3-63ambarihdp21.log
4. Stop the NFS Gateway services.

   hadoop-daemon.sh stop nfs3

   hadoop-daemon.sh stop portmap

What to do next

Verify the validity of the NFS services

Verify validity of the NFS services

After starting the NFS services on the Gateway, you must verify whether all the services are running. Additionally, you must ensure that the HDFS namespace is exported and can be mounted.

Procedure

1. Run the rpcinfo command to verify whether all the NFS services are running.

   rpcinfo -p $nfs_server_ip

   The command returns an output similar to the following:

```
    program vers proto   port
    100005    1   tcp   4242  mountd
    100005    2   udp   4242  mountd
    100005    2   tcp   4242  mountd
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100005    3   udp   4242  mountd
    100005    1   udp   4242  mountd
    100003    3   tcp   2049  nfs
    100005    3   tcp   4242  mountd
```
2. Verify that the HDFS namespace is exported and can be mounted.

   showmount -e $nfs_server_ip

   The command returns an exports list similar to the following:

   Exports list on $nfs_server_ip : / (everyone)

Access HDFS from the NFS Gateway

To access HDFS, you must first mount the namespace and then set up the NFS client hosts to interact with HDFS through the Gateway.

Procedure

1. Mount the HDFS namespace.

   mount -t nfs -o vers=3,proto=tcp,nolock,sync,rsize=1048576,wsize=1048576 $server:/ $mount_point

   Access HDFS as part of the local file system, except that hard link or symbolic link and random write are not supported.

   Note: Because NLM is not supported, the mount option nolock is required.

   You can use the sync option for improving the performance and reliability when writing large files. If the sync option is specified, the NFS client flushes write operations to the NFS Gateway before returning control to the client application. Additionally, the client does not issue reordered writes. This reduces buffering requirements on the NFS gateway.
2. Set up the NFS client users to interact with HDFS through the NFS Gateway.

Related Concepts

How NFS Gateway authenticates and maps users

How NFS Gateway authenticates and maps users

The manner in which NFS Gateway authenticates and maps users determines how NFS client users access HDFS through the NFS Gateway.

NFS Gateway uses AUTH_UNIX-style authentication, which means that the user logged on the client is the same that NFS passes to HDFS.

For example, suppose that the NFS client has current user as admin. When the user accesses the mounted HDFS directory, NFS gateway accesses HDFS as the user admin. To access HDFS as hdfs user, you must first switch the current user to hdfs on the client system before accessing the mounted directory.

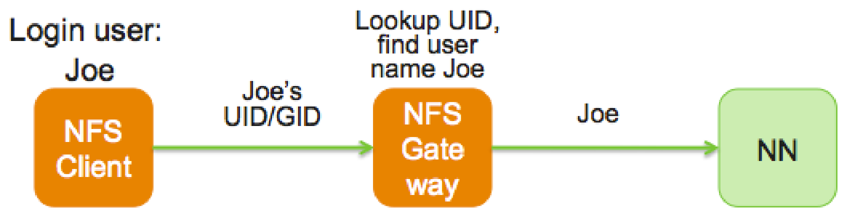
For the NFS client users accessing HDFS, the Gateway converts the User Identifier (UID) to a username, which is used by HDFS to check permissions. Therefore, you must ensure that the user on the NFS client host has the same name and UID as that on the NFS Gateway host.

Note: If you use the same user management system such as LDAP/NIS to create and deploy users to the HDP nodes and to the client host, then the user names and UIDs are usually identical on both the client and the Gateway.

If the user is created manually on different hosts, you might need to modify the UID, as specified in the following example, on either the client or the NFS Gateway to make them the same:

usermod -u 123 $myusername

The following diagram illustrates how the UID and name are communicated between the NFS client, NFS gateway, and NameNode.



Using the NFS Gateway with ViewFs

You can use the NFS Gateway to export federated namespaces that are available as ViewFs mounts. You can access multiple ViewFs mounts simultaneously using a single NFS Gateway and perform operations on them.

Considerations for using NFS Gateway with ViewFs

- You can use NFS Gateway to export only those ViewFs mounts that have HDFS as the underlying file system.
- The ViewFs root cannot be mounted.
- Files and directories cannot be renamed across mounts.

Export ViewFs mounts using the NFS Gateway

You can export ViewFs mounts using the NFS Gateway. For every ViewFs mount entry to export, you must specify an NFS export point and mount it to a corresponding directory path on the NFS Gateway.

Before you begin

The file with the ViewFs mount table entries must already be created.

About this task

You can export only those ViewFs mounts that have HDFS as the underlying file system.

Procedure

1. Configure the proxy user for the NFS Gateway and the various Gateway settings.

   Configure the NFS Gateway
2. Start the following daemons to run the NFS services on the Gateway: rpcbind (or portmap), mountd, and nfsd.

   Start and stop the NFS Gateway services
3. In the NFS Gateway node, specify an export point corresponding to each ViewFs mount that you want to export.

   Consider the following example of a ViewFs mount table entry:

   ```
   <property>
     <name>fs.viewfs.mounttable.ClusterX.link./home</name>
     <value>hdfs://nn1-clusterx.example.com:8020/home</value>
   </property>
   ```

   You can specify a corresponding NFS export point as follows:

   ```
   <property>
     <name>nfs.export.point</name>
     <value>/home</value>
   </property>
   ```
4. Verify if the exported namespace can be mounted.

   ```
   showmount -e $nfs_server_ip
   ```

   Note: The IP address of the NFS Gateway need not be the same as the IP address of the NameNode.
5. Mount the exported namespace to a corresponding path on the NFS Gateway.

   For the export point example in step 3, you can create a mount as follows:

   ```
   mount -t nfs -o vers=3,proto=tcp,nolock,noacl,sync nfs_server_ip:/home /
   mount_dir
   ```

Related Concepts

Example of ViewFs mount table entries

Related Information

HDFS NFS Gateway - Mount the export

Data storage metrics

Use Java Management Extensions (JMX) APIs to collect the metrics exposed by the various HDFS daemons. For the garbage collection metrics from the NameNode, you can use the Concurrent Mark Sweep (CMS) GC or configure the Garbage First Garbage Collector (G1GC).

Using JMX for accessing HDFS metrics

You can access HDFS metrics over Java Management Extensions (JMX) through either the web interface of an HDFS daemon or by directly accessing the JMX remote agent.

Using the HDFS Daemon Web Interface

You can access JMX metrics through the web interface of an HDFS daemon. This is the recommended method.

For example, use the following command format to access the NameNode JMX:

```
curl -i http://localhost:50070/jmx
```

You can use the qry parameter to fetch only a particular key:

```
curl -i http://localhost:50070/jmx?
qry=Hadoop:service=NameNode,name=NameNodeInfo
```

Directly Accessing the JMX Remote Agent

This method requires that the JMX remote agent is enabled with a JVM option when starting HDFS services.

For example, the following JVM options in hadoop-env.sh are used to enable the JMX remote agent for the NameNode. It listens on port 8004 with SSL disabled. The user name and password are saved in the mxremote.password file.

```
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.password.file=$HADOOP_CONF_DIR/
jmxremote.password
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=8004 $HADOOP_NAMENODE_OPTS"
```

See the Oracle Java SE documentation for more information about the related settings.

You can also use the jmxquery tool to retrieve information through JMX.

Hadoop has a built-in JMX query tool, jmxget. For example:

```
hdfs jmxget -server localhost -port 8004 -service NameNode
```

Note: jmxget requires that authentication be disabled, as it does not accept a user name and password.

Using JMX can be challenging for operations personnel who are not familiar with JMX setup, especially JMX with SSL and firewall tunnelling. Therefore, we recommend that you collect JMX information through the web interface of HDFS daemons rather than directly accessing the JMX remote agent.

Related Information

JMX Query

Monitoring and Management Using JMX Query

Configure the G1GC garbage collector (Technical Preview)

The Oracle JDK 7 update 4 introduced the Garbage First Garbage Collector (G1GC). You must follow certain recommendations when switching from the currently used Concurrent Mark Sweep (CMS) GC to G1GC.

Note: This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at https://support.hortonworks.com.

Recommended settings for G1GC

The recommended settings for configuring Garbage First Garbage Collector (G1GC) include allocating more Java heap space when compared to the Concurrent Mark Sweep (CMS) GC, and setting specific values for properties such as MaxGCPauseMillis and ParallelGCThreads.

No significant improvements have been observed in the NameNode startup process when using G1GC instead of CMS.

The following NameNode settings are recommended for G1GC in a large cluster:

- Approximately 10% more Java heap space (-XX:Xms and -XX:Xmx) should be allocated to the NameNode, as compared to CMS setup.

  See Command Line Installation Guide for recommendations on setting the CMS heap size.
- For large clusters (>50M files), MaxGCPauseMillis should be set to 4000.
- You should set ParallelGCThreads to 20 (default for a 32-core machine), as opposed to 8 for CMS.
- Other G1GC parameters should be left set to their default values.

We have observed that the G1GC does not comply with the maximum heap size (-XX:Xmx) setting. For Xmx = 110 GB, we observed the following VM statistics:

- For CMS: Maximum heap (VmPeak) = 113 GB.
- For G1GC: Maximum heap (VmPeak) = 147 GB.

Related Tasks

Switching from CMS to G1GC

Switching from CMS to G1GC

To move from Concurrent Mark Sweep (CMS) GC to Garbage First (G1) GC, you must update the HADOOP_NAMENODE_OPTS settings in hadoop-env.sh.

Procedure

On the Ambari dashboard, select HDFS > Configs > Advanced > Advanced hadoop-env.

Make the following changes to the HADOOP_NAMENODE_OPTS settings:

- Replace -XX:+UseConcMarkSweepGC with -XX:+UseG1GC
- Remove -XX:+UseCMSInitiatingOccupancyOnly and -XX:CMSInitiatingOccupancyFraction=####
- Remove -XX:NewSize=#### and -XX:MaxNewSize=####
- (Optional) Add -XX:MaxGCPauseMillis=####
- (Optional) Add -XX:InitiatingHeapOccupancyPercent=####
- (Optional) Add -XX:ParallelGCThreads=####, if not present.

  The default value of this parameter is set to the number of logical processors (up to a value of 8). For more than eight logical processors, the default value is set to 5/8th the number of logical processors.

Related Concepts

Recommended settings for G1GC

APIs for accessing HDFS

Use the WebHDFS REST API to access an HDFS cluster from applications external to the cluster. WebHDFS supports all HDFS user operations including reading files, writing to files, making directories, changing permissions and renaming. In addition, WebHDFS uses Kerberos and delegation tokens for authenticating users.

Related Information

WebHDFS - HTTP REST Access to HDFS

Set up WebHDFS on a secure cluster

Setting up WebHDFS on a secure cluster requires you to update certain properties on hdfs-site.xml, create an HTTP service user principal, create a keytab for the principal, and verify the association of the principal and keytab with the correct HTTP service.

Procedure

1. Set the value of the dfs.webhdfs.enabled property in hdfs-site.xml to true.

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
```

2. Create an HTTP service user principal.

```
kadmin: addprinc -randkey HTTP/$<Fully_Qualified_Domain_Name>@
$<Realm_Name>.COM
```

   where:

   - Fully_Qualified_Domain_Name: Host where the NameNode is deployed.
   - Realm_Name: Name of your Kerberos realm.

3. Create a keytab file for the HTTP principal.

```
kadmin: xst -norandkey -k /etc/security/spnego.service.keytab HTTP/
$<Fully_Qualified_Domain_Name>
```

4. Verify that the keytab file and the principal are associated with the correct service.

```
klist –k -t /etc/security/spnego.service.keytab
```

5. Add the dfs.web.authentication.kerberos.principal and dfs.web.authentication.kerberos.keytab properties to hdfs-site.xml.

```
<property>
  <name>dfs.web.authentication.kerberos.principal</name>
  <value>HTTP/$<Fully_Qualified_Domain_Name>@$<Realm_Name>.COM</value>
</property>
<property>
  <name>dfs.web.authentication.kerberos.keytab</name>
  <value>/etc/security/spnego.service.keytab</value>
</property>
```

6. Restart the NameNode and the DataNodes.