Cloudera Runtime 7.2.12

Configuring Apache HBase

Date published: 2020-02-29
Date modified: 2021-10-25

https://docs.cloudera.com/

Legal Notice

Contents

Using DNS with HBase

You must configure DNS to resolve RegionServer and Master hostnames in your cluster. HBase uses the local hostname to report its IP address. Both forward and reverse DNS resolving works.

Procedure

1. In Cloudera Manager navigate to HBase Configuration .
2. Find the RegionServer Advanced Configuration Snippet (Safety-Valve) for hbase-site.xml property.
3. Click the plus icon to add the property that is applicable for your use case:

   If your server has multiple interfaces, HBase uses the interface that the primary hostname resolves to, and you must ensure that the cluster configuration is consistent and every host has the same network interface configuration. If this is your use case add the following configuration:

   - Name: hbase.regionserver.dns.interface
   - Value: primary interface

   If you want to use a different DNS name server than the system-wise default, add the following configuration:

   - Name: hbase.regionserver.dns.nameserver
   - Value: a different DNS name server
4. Click Save Changes.

Use the Network Time Protocol (NTP) with HBase

Ensure that that the clocks on all the cluster members are sychronized for your cluster to function correctly. You must configure NTP to synchronize the clock.

About this task

The clocks on cluster members must be synchronized for your cluster to function correctly. Some skew is tolerable, but excessive skew could generate odd behaviors.

Using NTP and DNS ensures that you won't run into odd behaviors when one node A thinks that the time is tomorrow and node B thinks it's yesterday. You will also prevent situations where the master node tells node C to serve a region but node C doesn't know its own name and doesn't answer.

Procedure

1. Run NTP or another clock synchronization mechanism on your cluster.
2. Verify that the system time is sychronized across your cluster nodes.

What to do next

For more information about NTP, see the NTP website

Configure the graceful shutdown timeout property

You must configure this property to allow enough time for a graceful shutdown of a RegionServer. A graceful shutdown of an HBase RegionServer allows the regions hosted by that RegionServer to be moved to other RegionServers before stopping the RegionServer.

About this task

This timeout only affects a graceful shutdown of the entire HBase service, not individual RegionServers. Therefore, if you have a large cluster with many RegionServers, you should strongly consider increasing the timeout from its default of 180 seconds.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope > HBASE-1 (Service Wide).
4. Use the Search box to search for the Graceful Shutdown Timeout property and edit the value.
5. Click Save Changes to save this setting.

Related Information

Graceful HBase Shutdown

Setting user limits for HBase

You must set user limits to avoid opening many files at the same time. Overloading many files at the same time leads to failure and causes error messages.

Because HBase is a database, it opens many files at the same time. The default setting of 1024 for the maximum number of open files on most Unix-like systems is insufficient. Any significant amount of loading will result in failures and cause error message such as java.io.IOException...(Too many open files) to be logged in the HBase or HDFS log files. For more information about this issue, see the Apache HBase Book. You may also notice errors such as:

```
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Exception inc
reateBlockOutputStream java.io.EOFException
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Abandoning bl
ock blk_-6935524980745310745_1391901
```

Another setting you should configure is the number of processes a user is permitted to start. The default number of processes is typically 1024. Consider raising this value if you experience OutOfMemoryException errors.

Configure ulimit for HBase using Cloudera Manager

You can use Cloudera Manager to configure ulimit.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope Master or Scope RegionServer .
4. Locate the Maximum Process File Descriptors property or search for it by typing its name in the Search box.
5. Edit the property value.
6. Restart the role.
7. Restart the service.

Configuring ulimit for HBase

You can configure the ulimit for HBase using the Maximum Process File Descriptors property in Cloudera Manager.

About this task

Cloudera recommends increasing the maximum number of file handles to more than 10,000. Increasing the file handles for the user running the HBase process is an operating system configuration, not an HBase configuration. A common mistake is to increase the number of file handles for a particular user when HBase is running as a different user. HBase prints the ulimit it is using on the first line in the logs. Make sure that it is correct.

Procedure

1. In Cloudera Manager navigate to HBase Configuration .
2. Find the Maximum Process File Descriptors property.
3. Set the property as applicable for your use case.
4. Click Save Changes.

Important: After changing the maximum process file descriptor limit specified in the above steps, the number of file descriptors still limits service roles. Raising the maximum process file descriptors above the Linux kernel file descriptor limit has no effect. Check the Linux kernel file descriptor limit on every host in the cluster and raise that if necessary.

You can find the Linux kernel file descriptor limit by running the following command on the Linux command line:

```
sudo cat /proc/sys/fs/nr_open
```

Configure ulimit using Pluggable Authentication Modules using the Command Line

Follow these command-line instructions to configure ulimit using Pluggable Authentication Modules on systems that do not use Cloudera Manager.

About this task

If you are using ulimit, you must make the following configuration changes:

Procedure

1. In the /etc/security/limits.conf file, add the following lines, adjusting the values as appropriate. This assumes that your HDFS user is called hdfs and your HBase user is called hbase.

```
hdfs  -       nofile  32768
hdfs  -       nproc   2048
hbase -       nofile  32768
hbase -       nproc   2048
```

Note:

- Only the root user can edit this file.
- If this change does not take effect, check other configuration files in the /etc/security/limits.d/ directory for lines containing the hdfs or hbase user and the nofile value. Such entries may be overriding the entries in /etc/security/limits.conf.

2. To apply the changes in /etc/security/limits.conf on Ubuntu and Debian systems, add the following line in the /etc /pam.d/common-session file:

```
session required  pam_limits.so
```

For more information on the ulimit command or per-user operating system limits, refer to the documentation for your operating system.

Using dfs.datanode.max.transfer.threads with HBase

You must configure the dfs.datanode.max.transfer.threads with HBase to specify the maximum number of files that a DataNode can serve at any one time.

About this task

A Hadoop HDFS DataNode has an upper bound on the number of files that it can serve at any one time. The upper bound is controlled by the dfs.datanode.max.transfer.threads property. Before loading, make sure you have configured this property to at least 4096.

Procedure

1. In Cloudera Manager navigate to HDFS Configuration .
2. Find the DataNode Advanced Configuration Snippet (Safety-Valve) for hdfs-site.xml property.
3. Click the plus icon to add a new property:
   - Name: dfs.datanode.max.transfer.threads
   - Value: at least 4096
4. Clik Save Changes.
5. Restart the HDFS service.

If the value is not set to an appropriate value, strange failures can occur and an error message about exceeding the number of transfer threads will be added to the DataNode logs. Other error messages about missing blocks are also logged, such as the following:

```
06/12/14 20:10:31 INFO hdfs.DFSClient: Could not obtain block blk_XXXXXXXXXX
XXXXXXXXXXXX_YYYYYYYY from any node: java.io.IOException: No live nodes cont
ain current block. Will get new block locations from namenode and retry…
```

Configure encryption in HBase

You must encrypt the HBase root directory to ensure that you have an additional layer of protection in case the HDFS filesystem is compromised. You can encrypt the HBase root directory within HDFS, using HDFS Transparent Encryption.

About this task

HBase stores all of its data under its root directory in HDFS configured in the hbase.rootdir. If you use this feature in combination with bulk-loading of HFiles, you must configure hbase.bulkload.staging.dir to point to a location within the same encryption zone as the HBase root directory. Otherwise, you may encounter errors such as:

```
org.apache.hadoop.ipc.RemoteException(java.io.IOException): /tmp/output/f/5
can't be moved into an encryption zone.
```

Procedure

- Enable HDFS encryption using the HDFS encryption wizard.
- Follow the instructions for setting up HDFS Transparent Encryption.
- Validate and verify that HDFS encryption is enabled and working.

  For more information see, HDFS Transparent Encryption.

Using hedged reads

You can enable hedged reads if you want to increase the performance of a read operation from an HDFS block that occasionally takes a long time. This feature helps in situations where a read occasionally takes a long time rather than when there is a systemic problem.

If a read from an HDFS block is slow, the HDFS client starts up another parallel, 'hedged' read against a different block replica. The result of whichever read returns first is used, and the outstanding read is cancelled. Hedged reads can be enabled for HBase when the HFiles are stored in HDFS. This feature is disabled by default.

Enable hedged reads for HBase

You need to enable hedged read if a read operation from a HDFS block is slow.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope > HBASE-1 (Service-Wide).
4. Select Category > Performance.
5. Configure the HDFS Hedged Read Threadpool Size and HDFS Hedged Read Delay Threshold properties. The descriptions for each of these properties on the configuration pages provide more information.
6. Enter a Reason for change, and then click Save Changes to commit the changes.

Monitor the performance of hedged reads

You can monitor the performance of hedged reads using the following metrics provided by Hadoop when hedged reads are enabled.

You can monitor the following properties:

- hedgedReadOps - the number of hedged reads that have occurred
- hedgeReadOpsWin - the number of times the hedged read returned faster than the original read

Understanding HBase garbage collection

You can configure garbage collection to free up the memory that is no longer referenced by Java objects.

Warning: Configuring the JVM garbage collection for HBase is an advanced operation. Incorrect configuration can have major performance implications for your cluster. Test any configuration changes carefully.

Garbage collection (memory cleanup) by the JVM can cause HBase clients to experience excessive latency.

To tune the garbage collection settings, you pass the relevant parameters to the JVM.

Example configuration values are not recommendations and should not be considered as such. This is not the complete list of configuration options related to garbage collection. See the documentation for your JVM for details on these settings.

- -XX:+UseG1GC: Use the 'G1' garbage collection algorithm. You can tune G1 garbage collection to provide a consistent pause time, which benefits long-term running Java processes such as HBase, NameNode, Solr, and ZooKeeper. For more information about tuning G1, see the Oracle documentation on tuning garbage collection.

- -XX:MaxGCPauseMillis=value: The garbage collection pause time. Set this to the maximum amount of latency your cluster can tolerate while allowing as much garbage collection as possible. XX:+ParallelRefProcEnabled Enable or disable parallel reference processing by using a + or - symbol before the parameter name.
- -XX:-ResizePLAB: Enable or disable resizing of Promotion Local Allocation Buffers (PLABs) by using a + or - symbol before the parameter name.
- -XX:ParallelGCThreads=value: The number of parallel garbage collection threads to run concurrently.
- -XX:G1NewSizePercent=value: The percent of the heap to be used for garbage collection. If the value is too low, garbage collection is ineffective. If the value is too high, not enough heap is available for other uses by HBase.

Related Information

Tuning Java Garbage Collection for HBase

Configure HBase garbage collection

You must configure garbage collection using Cloudera Manager.

About this task

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope > RegionServer.
4. Select Category > Advanced.
5. Locate the Java Configuration Options for HBase RegionServer property or search for it by typing its name in the Search box.
6. Add or modify JVM configuration options.
7. Enter a Reason for change, and then click Save Changes to commit the changes.
8. Restart the role.

Disable the BoundedByteBufferPool

HBase uses a BoundedByteBufferPool to avoid fragmenting the heap. You can disable BoundedByteBufferPool using Cloudera Manager.

About this task

The G1 garbage collector reduces the need to avoid fragmenting the heap in some cases. If you use the G1 garbage collector, you can disable the BoundedByteBufferPool in HBase. This can reduce the number of "old generation" items that need to be collected. This configuration is experimental.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope > RegionServer.
4. Select Category > Advanced.
5. Locate the HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property, or search for it by typing its name in the Search box.
6. Add the following XML:

```
<property>
```

```
  <name>hbase.ipc.server.reservoir.enabled</name>
  <value>false</value>
</property>
```

7. Enter a Reason for change, and then click Save Changes to commit the changes.
8. Restart the service.

Configure the HBase canary

The HBase canary is an optional service that you can configure to check periodically if a RegionServer is alive. The HBase canary is disabled by default.

About this task

This canary is different from the Cloudera Service Monitoring canary and is provided by the HBase service. After enabling the canary, you can configure several different thresholds and intervals relating to it, as well as exclude certain tables from the canary checks. The canary works on Kerberos-enabled clusters if you have the HBase client configured to use Kerberos.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope > HBase or HBase Service-Wide.
4. Select Category > Monitoring.
5. Locate the HBase Canary property or search for it by typing its name in the Search box. Several properties have Canary in the property name.
6. Select the checkbox.
7. Review other HBase Canary properties to configure the specific behavior of the canary. To apply this configuration property to other role groups as needed, edit the value for the appropriate role group.See Modifying Configuration Properties Using Cloudera Manager.
8. Enter a Reason for change, and then click Save Changes to commit the changes.
9. Restart the role.
10. Restart the service.

Configuring auto split policy in an HBase table

Know how to configure automatic split policy in an HBase table.

About this task

When a region in an HBase table reaches a certain limit, HBase automatically divides it into two regions. This is the HBase default split policy. However, you can override the default split policy and configure a custom split policy for the regions in an HBase table.

Using the Cloudera Manager, you can configure the default split policy at the system level or the table level using the HBase shell.

In this topic, let us consider an example of configuring the split policy for regions in an HBase table by setting the hbase.regionserver.region.split.policy parameter.

Before you begin

Ensure that you have administrator privileges in the CDP environment.

Procedure

1. Log in to the Cloudera Manager as an administrator.
2. Select the HBase service.
3. Go to Configuration Category Advanced HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml .
4. Configure the split policy that you need.

   For example, to set the split policy for the regions in an HBase table set the value of hbase.regionserver.region.split.policy as org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy.

   For more information on custom policies, see *HBase custom split policies*.
5. Click Save Changes.
6. Restart the HBase service.

   Alternatively, you can use the HBase shell to set the split policy for the regions at the table level.

   a. Log in to the HBase shell.
   b. Use the ALTER command to set the value to org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplit Policy.

      The following is a sample command.

```
$ hbase shell
$ hbase > alter '<TABLE_NAME>', {METADATA => {'SPLIT_POLICY' => 'org.a
pache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy'}}
```

Related Information

Apache HBase Region Splitting and Merging

HBase custom split policies

Using HBase blocksize

You must configure the HBase blocksize to set the smallest unit of data HBase can read from the column family's HFiles.

HBase data is stored in one (after a major compaction) or more (possibly before a major compaction) HFiles per column family per region. The blocksize determines:

- The blocksize for a given column family determines the smallest unit of data HBase can read from the column family's HFiles.
- The basic unit of measure cached by a RegionServer in the BlockCache.

The default blocksize is 64 KB. The appropriate blocksize is dependent upon your data and usage patterns. Use the following guidelines to tune the blocksize size, in combination with testing and benchmarking as appropriate.

Warning: The default blocksize is appropriate for a wide range of data usage patterns, and tuning the blocksize is an advanced operation. The wrong configuration can negatively impact performance.

- Consider the average key/value size for the column family when tuning the blocksize. You can find the average key/value size using the HFile utility:

```
$ hbase org.apache.hadoop.hbase.io.hfile.HFile -f /path/to/HFILE -m -v
...
Block index size as per heapsize: 296
```

```
reader=hdfs://srv1.example.com:9000/path/to/HFILE, \
compression=none, inMemory=false, \
firstKey=US6683275_20040127/mimetype:/1251853756871/Put, \
lastKey=US6684814_20040203/mimetype:/1251864683374/Put, \
avgKeyLen=37, avgValueLen=8, \
entries=1554, length=84447
...
```

- Consider the pattern of reads to the table or column family. For instance, if it is common to scan for 500 rows on various parts of the table, performance might be increased if the blocksize is large enough to encompass 500-1000 rows, so that often, only one read operation on the HFile is required. If your typical scan size is only 3 rows, returning 500-1000 rows would be overkill.

  It is difficult to predict the size of a row before it is written, because the data will be compressed when it is written to the HFile. Perform testing to determine the correct blocksize for your data.

Configure the blocksize for a column family

You can configure the blocksize of a column family at table creation or by disabling and altering an existing table.

About this task

These instructions are valid whether or not you use Cloudera Manager to manage your cluster.

To configure the blocksize for a column family:

Procedure

1. In the HBase shell, type:

   ```
   hbase> create ‘test_table#,{NAME => ‘test_cf#, BLOCKSIZE => '262144'}
   hbase> disable 'test_table'
   hbase> alter 'test_table', {NAME => 'test_cf', BLOCKSIZE => '524288'}
   hbase> enable 'test_table'
   ```

   After changing the blocksize, the HFiles will be rewritten during the next major compaction.
2. To trigger a major compaction, issue the following command in HBase Shell:

   ```
   hbase> major_compact 'test_table'
   ```

   Depending on the size of the table, the major compaction can take some time and have a performance impact while it is running.
3. To view the blocksize metrics, see the block_cache* entries in the RegionServer metrics section in the HBase web user interface.

Configuring HBase BlockCache

You can configure BlockCache in two different ways in HBase: the default on-heap LruBlockCache and the BucketCache, which is usually off-heap.

If you have less than 20 GB of RAM available for use by HBase, consider tailoring the default on-heap BlockCache implementation (LruBlockCache) for your cluster.

If you have more than 20 GB of RAM available, consider adding off-heap BlockCache (BucketCache).

In the default configuration, HBase uses a single on-heap cache. If you configure the off-heap BucketCache, the on-heap cache is used for Bloom filters and indexes, and the off-heap BucketCache is used to cache data blocks. This is called the Combined Blockcache configuration. The Combined BlockCache allows you to use a larger in-memory

cache while reducing the negative impact of garbage collection in the heap, because HBase manages the BucketCache instead of relying on the garbage collector.

Contents of the BlockCache

In HBase, a block is a single unit of I/O. The block cache keeps data blocks resident in the memory after they are read.

To size the BlockCache correctly, you need to understand what HBase places into it.

- Your data: Each time a Get or Scan operation occurs, the result is added to the BlockCache if it was not already cached there. If you use the BucketCache, data blocks are always cached in the BucketCache.
- Row keys: When a value is loaded into the cache, its row key is also cached. This is one reason to make your row keys as small as possible. A larger row key takes up more space in the cache.
- hbase:meta: The hbase:meta catalog table keeps track of which RegionServer is serving which regions. It can consume several megabytes of cache if you have a large number of regions, and has in-memory access priority, which means HBase attempts to keep it in the cache as long as possible.
- Indexes of HFiles: HBase stores its data in HDFS in a format called HFile. These HFiles contain indexes which allow HBase to seek for data within them without needing to open the entire HFile. The size of an index is a factor of the block size, the size of your row keys, and the amount of data you are storing. For big data sets, the size can exceed 1 GB per RegionServer, although the entire index is unlikely to be in the cache at the same time. If you use the BucketCache, indexes are always cached on-heap.
- Bloom filters: If you use Bloom filters, they are stored in the BlockCache. If you use the BucketCache, Bloom filters are always cached on-heap.

The sum of the sizes of these objects is highly dependent on your usage patterns and the characteristics of your data. For this reason, the HBase Web UI and Cloudera Manager each expose several metrics to help you size and tune the BlockCache.

Size the BlockCache

When you use the LruBlockCache, the blocks needed to satisfy each read are cached, old blocks are evicted to make room for new blocks using a Least-Recently-Used algorithm . Set the size of the BlockCache to satisfy your read requirements.

The size cached objects for a given read may be significantly larger than the actual result of the read. For instance, if HBase needs to scan through 20 HFile blocks to return a 100 byte result, and the HFile blocksize is 100 KB, the read will add 20 * 100 KB to the LruBlockCache.

Because the LruBlockCache resides entirely within the Java heap, the amount of which is available to HBase and what percentage of the heap is available to the LruBlockCache strongly impact performance. By default, the amount of HBase heap reserved for LruBlockCache (hfile.block.cache.size) is .40, or 40%. To determine the amount of heap available for the LruBlockCache, use the following formula. The 0.99 factor allows 1% of heap to be available as a "working area" for evicting items from the cache. If you use the BucketCache, the on-heap LruBlockCache only stores indexes and Bloom filters, and data blocks are cached in the off-heap BucketCache.

```
number of RegionServers * heap size * hfile.block.cache.size * 0.99
```

To tune the size of the LruBlockCache, you can add RegionServers or increase the total Java heap on a given RegionServer to increase it, or you can tune hfile.block.cache.size to reduce it. Reducing it will cause cache evictions to happen more often, but will reduce the time it takes to perform a cycle of garbage collection. Increasing the heap will cause garbage collection to take longer but happen less frequently.

Decide to use the BucketCache

The BucketCache manages areas of memory called *buckets* for holding the cached blocks. You can use BucketCache if any of the conditions listed in here are are true.

- If the result of a Get or Scan typically fits completely in the heap, the default configuration, which uses the on-heap LruBlockCache, is the best choice, as the L2 cache will not provide much benefit. If the eviction rate is low, garbage collection can be 50% less than that of the BucketCache, and throughput can be at least 20% higher.
- Otherwise, if your cache is experiencing a consistently high eviction rate, use the BucketCache, which causes 30-50% of the garbage collection of LruBlockCache when the eviction rate is high.
- BucketCache using file mode on solid-state disks has a better garbage-collection profile but lower throughput than BucketCache using off-heap memory.

About the Off-heap BucketCache

If the BucketCache is enabled, it stores data blocks, leaving the on-heap cache free for storing indexes and Bloom filters.

The physical location of the BucketCache storage can be either in memory (off-heap) or in a file stored in a fast disk.

- Off-heap: This is the default configuration.
- File-based: You can use the file-based storage mode to store the BucketCache on an SSD or FusionIO device,

You can configure a column family to keep its data blocks in the L1 cache instead of the BucketCache, using the HColumnDescriptor.cacheDataInL1(true) method or by using the following syntax in HBase Shell:

```
hbase> alter 'myTable', CONFIGURATION => {CACHE_DATA_IN_L1 => 'true'}}
```

Off-heap BucketCache

If the BucketCache is enabled, it stores data blocks, leaving the on-heap cache free for storing indexes and Bloom filters. The physical location of the BucketCache storage can be either in memory (off-heap) or in a file stored in a fast disk.

- Off-heap: This is the default configuration.
- File-based: You can use the file-based storage mode to store the BucketCache on an SSD or FusionIO device,

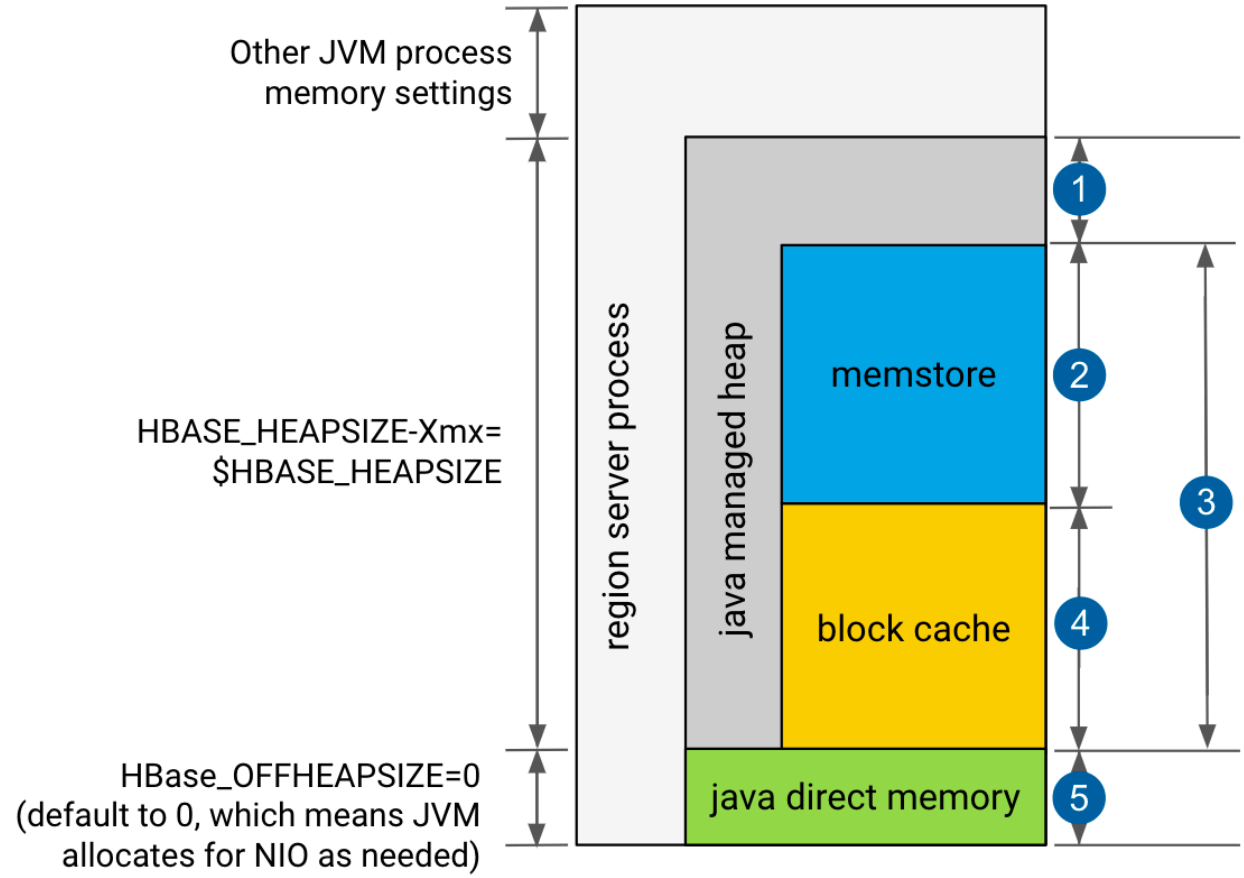
This table summaries the important configuration properties for the BucketCache. To configure the BucketCache. The table is followed by three diagrams that show the impacts of different blockcache settings.

Table 1: BucketCache Configuration Properties

Property	Default	Description
hbase.bucketcache.combinedcache.enabled	true	When BucketCache is enabled, use it as a L2 cache for LruBlockCache. If set to true, indexes and Bloom filters are kept in the LruB lockCache and the data blocks are kept in the BucketCache.
hbase.bucketcache.ioengine	none (BucketCache is disabled by default)	Where to store the contents of the BucketCa che. Its value can be offheap, file:*PATH*, mmap:*PATH* or pmem:*PATH* where PATH is the path to the file that host the file-based cache.
hfile.block.cache.size	0.4	A float between 0.0 and 1.0. This factor multiplied by the Java heap size is the size of the L1 cache. In other words, the percentage of the Java heap to use for the L1 cache.

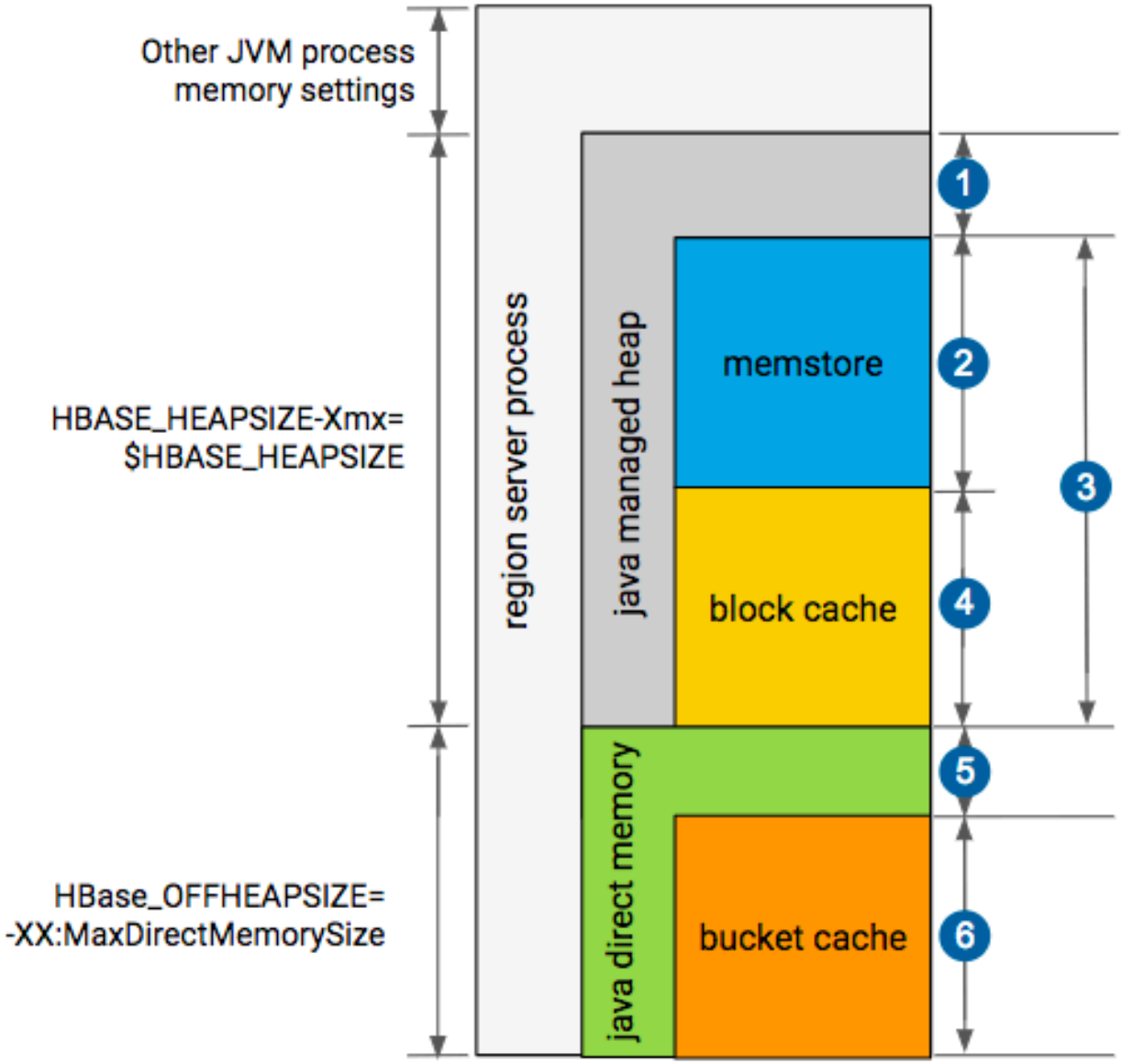
Property	Default	Description
hbase.bucketcache.size	not set	When using BucketCache, this is a float that represents one of two different values, depending on whether it is a floating-point decimal less than 1.0 or an integer greater than 1.0. • If less than 1.0, it represents a percentage of total heap memory size to give to the cache. • If greater than 1.0, it represents the capacity of the cache in megabytes
hbase.bucketcache.bucket.sizes	4, 8, 16, 32, 40, 48, 56, 64, 96, 128, 192, 256, 384, 512 KB	A comma-separated list of sizes for buckets for the BucketCache if you prefer to use multiple sizes. The sizes should be multiples of the default blocksize, ordered from smallest to largest. The sizes you use will depend on your data patterns. This parameter is experimental.
-XX:MaxDirectMemorySize	MaxDirectMemorySize = BucketCache + 1	A JVM option to configure the maximum amount of direct memory available for the JVM. It is automatically calculated and configured based on the following formula: MaxDirectMemorySize = BucketCache size + 1 GB for other features using direct memory, such as DFSClient. For example, if the BucketCache size is 8 GB, it will be -XX: MaxDirectMemorySize=9G.

Figure 1: Default LRUCache, L1 only block cache hbase.bucketcache.ioengine=NULL



1. 20% minimum reserved for operations and rpc call queues
2. hbase.regionserver.global.memstore.size: default is 0.4, which means 40%
3. hbase.regionserver.global.memstore.size + hfile.block.cache.size #0.80, which means 80%
4. hfile.block.cache.size: default is 0.4, which means 40%
5. slack reserved for HDFS SCR/NIO: number of open HFiles * hbase.dfs.client.read.shortcircuit.buffer.size, where hbase.dfs.client.read.shortcircuit.buffer.size is set to 128k.

Figure 2: Default LRUCache, L1 only block cache hbase.bucketcache.ioengine=offheap



1. 20% minimum reserved for operations and rpc call queues
2. hbase.regionserver.global.memstore.size: default is 0.4, which means 40%
3. hbase.regionserver.global.memstore.size + hfile.block.cache.size #0.80, which means 80%
4. hfile.block.cache.size: default is 0.4 which means 40%
5. slack reserved for HDFS SCR/NIO: number of open HFiles * hbase.dfs.client.read.shortcircuit.buffer.size, where hbase.dfs.client.read.shortcircuit.buffer.size is set to 128k.

6. hbase.bucketcache.size: default is 0.0

   If hbase.bucketcache.size is float <1, it represents the percentage of total heap size.

   If hbase.bucketcache.size is #1, it represents the absolute value in MB. It must be < HBASE_OFFHEAPSIZE

BucketCache IO engine

Use the hbase.bucketcache.ioengine parameter to define where to store the content of the BucketCache. Its value can be offheap, file:*PATH*, mmap:*PATH*, pmem:*PATH* , or it can be empty. By default it is empty which means that BucketCache is disabled.

You can set the following values in the hbase.bucketcache.ioengine parameter to define where to store the BucketCache:

- offheap: When hbase.bucketcache.ioengine is set to offheap the content of the BucketCache is stored off-heap.
- file:*PATH*: When hbase.bucketcache.ioengine is set to file:*PATH*, the BucketCache uses file caching.
- mmap:*PATH*: When hbase.bucketcache.ioengine is set to mmap:*PATH*, the content of the BucketCache is stored and accessed through memory mapping to a file under the specified path.
- pmem:*PATH*: When hbase.bucketcache.ioengine is set to pmem:*PATH*, BucketCache uses direct memory access to and from a file on the specified path. The specified path must be under a volume that is mounted on a persistent memory device that supports direct access to its own address space.

  The advantage of the pmem engine over the mmap engine is that it supports large cache size. That is because pmem allows for reads straight from the device address, which means in this mode no copy is created on DRAM. Therefore, swapping due to DRAM free memory exhaustion is not an issue when large cache size is specified. With devices currently available, the bucket cache size can be set to the order of hundreds of GBs or even a few TBs.

  When bucket cache size is set to larger than 256GB, the OS limit must be increased, which can be configured by the max_map_count property. Make sure you have an extra 10% for other processes on the host that require the use of memory mapping. This additional overhead depends on the load of processes running on the RS hosts. To calculate the OS limit divide the block cache size in GB by 4 MB and then multiply it by 1.1: (block cache size in GB / 4 MB) * 1.1.

Configure BucketCache IO engine

You must configure the BucketCache IO engine using Cloudera Manager.

Set the value offheap and file:*PATH*

1. In Cloudera Manager select the HBase service and go to Configuration.
2. Search for BucketCache IOEngine and set it to the required value.

Set the value mmap:*PATH* and pmem:*PATH*

Important: These values can only be set using safety valves.

1. In Cloudera Manager select the HBase service and go to Configuration.
2. Search for RegionServer Advanced Configuration Snippet (Safety Valve) for hbase-site.xml.
3. Click the plus icon.
4. Set the required value:
   - Name: Add hbase.bucketcache.ioengine.
   - Value: Add either mmap:*PATH*: or pmem:*PATH*.

Configure the off-heap BucketCache using Cloudera Manager

You can configure the off-heap BucketCache engine using Cloudera Manager.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select the RegionServer scope and do the following:
   a) Set BucketCache IOEngine to offheap.
   b) Update the value of BucketCache Size according to the required BucketCache size.

      When you set the BucketCache Size, Cloudera Manager automatically sets the -XX:MaxDirectMemorySize JVM option for the RegionServers.

      Cloudera Manager automatically adds the JVM option -XX:MaxDirectMemorySize=<size> replacing <*SIZE*> with a value not smaller than the off-heap BucketCache, expressed as a number of gigabytes + around 1GB used for HDFS short circuit read. For example, if the off-heap BucketCache is 16GB, the total value of MaxDirectMemorySize could be 17G: -XX:MaxDirectMamorySize=17G.
4. Optionally, when combined BucketCache is in use, you can decrease the heap size ratio allocated to the L1 BlockCache, and increase the Memstore size. The on-heap BlockCache only stores indexes and Bloom filters, the actual data resides in the off-heap BucketCache. A larger Memstore is able to accommodate more write request before flushing them to disks.
   - Decrease HFile Block Cache Size to 0.3 or 0.2.
   - Increase Maximum Size of All Memstores in RegionServer to 0.5 or 0.6 respectively.
5. Enter a Reason for change, and then click Save Changes to commit the changes.
6. Restart or rolling restart your RegionServers for the changes to take effect.

Configure the off-heap BucketCache using the command line

You can configure the off-heap BucketCache engine from the command-line interface.

About this task

Procedure

1. Configure the MaxDirectMemorySize option for the RegionServers JVMS. Add the JVM option $HBASE_REGIONSERVER_OPTS -XX:MaxDirectMemorySize=<size>G, replacing <size> with a value not smaller than the aggregated heap size expressed as a number of gigabytes + the off-heap BucketCache, expressed as a number of gigabytes + around 1GB used for HDFS short circuit read. For example, if the off-heap BucketCache is 16GB and the heap size is 15GB, the total value of MaxDirectMemorySize could be 32: -XX:MaxDirectMamorySize=32G. This can be done adding the following line in hbase-env.sh:

   ```
   HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS -XX:MaxDirectMemorySize=<size>G"
   ```
2. In the hbase-site.xml files on the RegionServers, configure the properties in BucketCache configuration properties as appropriate, using the example below as a model.

   Important: If you are using Cloudera Manager, it can re-generate and therefore overwrite the hbase-site.xml configuration file. If you are using Cloudera Manager, Cloudera recommends to use the RegionServer Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property in Cloudera Manager to edit the hbase-site.xml configuration file on the RegionServer.

   ```
   <property>
     <name>hbase.bucketcache.combinedcache.enabled</name>
     <value>true</value>
   </property>
   <property>
     <name>hbase.bucketcache.ioengine</name>
     <value>offheap</value>
   </property>
   ```

```
<property>
  <name>hbase.bucketcache.size</name>
  <value>8388608</value>
</property>
<property>
  <name>hfile.block.cache.size</name>
  <value>0.2</value>
</property>
<property>
  <name>hbase.regionserver.global.memstore.size</name>
  <value>0.6</value>
</property>
```

Optionally, when combined BucketCache is in use, you can decrease the heap size ratio allocated to the L1 BlockCache, and increase the Memstore size as it is done in the above example. The on-heap BlockCache only stores indexes and Bloom filters, the actual data resides in the off-heap BucketCache. A larger Memstore is able to accommodate more write request before flushing them to disks.

- Decrease hfile.block.cache.size to 0.3 or 0.2.
- Increase hbase.regionserver.global.memstore.size to 0.5 or 0.6 respectively.

3. Restart each RegionServer for the changes to take effect.

Cache eviction priorities

You must decide on the cache eviction priorities to allow for scan-resistance and in-memory column families.

Both the on-heap cache and the off-heap BucketCache use the same cache priority mechanism to decide which cache objects to evict to make room for new objects. Three levels of block priority allow for scan-resistance and in-memory column families. Objects evicted from the cache are subject to garbage collection.

- Single access priority: The first time a block is loaded from HDFS, that block is given single access priority, which means that it will be part of the first group to be considered during evictions. Scanned blocks are more likely to be evicted than blocks that are used more frequently.
- Multi access priority: If a block in the single access priority group is accessed again, that block is assigned multi access priority, which moves it to the second group considered during evictions, and is therefore less likely to be evicted.
- In-memory access priority: If the block belongs to a column family which is configured with the in-memory configuration option, its priority is changed to in memory access priority, regardless of its access pattern. This group is the last group considered during evictions, but is not guaranteed not to be evicted. Catalog tables are configured with in-memory access priority.

  To configure a column family for in-memory access, use the following syntax in HBase Shell:

  ```
  hbase> alter 'myTable', 'myCF', CONFIGURATION => {IN_MEMORY => 'true'}
  ```

  To use the Java API to configure a column family for in-memory access, use the HColumnDescriptor.setInMemory(true) method.

Bypass the BlockCache

You can bypass the BlockCache if the data needed for a specific but atypical operation does not all fit in memory.

For an atypical operation does not all fit in memory, using the BlockCache can be counter-productive because data that you are still using may be evicted, or even if other data is not evicted, excess garbage collection can adversely effect performance. For this type of operation, you may decide to bypass the BlockCache. To bypass the BlockCache for a given Scan or Get, use the setCacheBlocks(false) method.

In addition, you can prevent a specific column family's contents from being cached, by setting its BLOCKCACHE configuration to false. Use the following syntax in HBase Shell:

```
hbase> alter 'myTable', CONFIGURATION => {NAME => 'myCF', BLOCKCACHE => 'fal
se'}
```

Monitor the BlockCache

Cloudera Manager provides metrics to monitor the performance of the BlockCache, to assist you in tuning your configuration.

You can view further detail and graphs using the RegionServer UI. To access the RegionServer UI in Cloudera Manager, go to the Cloudera Manager page for the host, click the RegionServer process, and click HBase RegionServer Web UI.

If you do not use Cloudera Manager, access the BlockCache reports at http://regionServer_host:22102/rs-status#m emoryStats, replacing regionServer_host with the hostname or IP address of your RegionServer.

Using quota management

Two types of HBase quotas are well established: throttle quota and number-of tables-quota. These two quotas can regulate users and tables.

In a multitenant HBase environment, ensuring that each tenant can use only its allotted portion of the system is key in meeting SLAs.

Table 2: Quota Support Matrix

Quota Type	Resource Type	Purpose	Namespace applicable?	Table applicable?	User applicable?
Throttle	Network	Limit overall network throughput and number of RPC requests	Yes	Yes	Yes
New space	Storage	Limit amount of storage used for table or namespaces	Yes	Yes	No
Number of tables	Metadata	Limit number of tables for each namespace or user	Yes	No	Yes
Number of regions	Metadata	Limit number of regions for each namespace	Yes	No	No

Configuring quotas

HBase quotas are disabled by default. To enable quotas, the relevant hbase-site.xml property must be set to true and the limit of each quota specified on the command line.

Before you begin

hbase superuser privileges

Procedure

1. Set the hbase.quota.enabled property in the hbase-site.xml file to true.
2. Enter the command to set the limit of the quota, type of quota, and to which entity to apply the quota. The command and its syntax are:

```
$hbase_shell> set_quota TYPE =>

    quota_type,
    arguments
```

General Quota Syntax

The general quota syntax are `THROTTLE_TYPE`, `Request sizes and space limit`, `Number of requests`, `Time limits` and `Number of tables or regions`.

THROTTLE_TYPE

Can be expressed as READ-only, WRITE-only, or the default type (both READ and WRITE permissions)

Timeframes

Can be expressed in the following units of time:

- sec (second)

  min (minute)

  hour

  day

Request sizes and space limit

Can be expressed in the following units:

- B: bytes

  K: kilobytes

  M: megabytes

  G: gigabytes

  P: petabytes

When no size units is included, the default value is bytes.

Number of requests

Expressed as integer followed by the string request

Time limits

Expressed as requests per unit-of-time or size per unit-of-time

Examples: 10req/day or 100P/hour

Number of tables or regions

Expressed as integers

Throttle quotas

The throttle quota, also known as RPC limit quota, is commonly used to manage length of RPC queue as well as network bandwidth utilization.

It is best used to prioritize time-sensitive applications to ensure latency SLAs are met.

Throttle quota examples

Following examples details the usage of adding throttle quotas commands, listing throttle quotas commands, and updating and deleting throttle quotas commands.

Examples of Adding Throttle Quotas Commands

Limit user u1 to 10 requests per second globally:

```
hbase> set_quota => TYPE => THROTTLE, USER => 'u1', LIMIT => '10req/sec'
```

Note: When you exceed the throttle quota limit defined under the LIMIT option, the command fails with an error.

Limit user u1 to up to 10MB of traffic per second globally:

```
hbase> set_quota => TYPE => THROTTLE, USER => 'u1', LIMIT => '10M/sec'
```

Limit user u1 to 10 requests/second globally for read operations. User u1 can still issue unlimited writes:

```
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => READ, USER => 'u1', LIMI
T => '10req/sec'
```

Limit user u1 to 10 requests/second globally for read operations. User u1 can still issue unlimited reads:

```
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => WRITE, USER => 'u1', LIM
IT => '10M/sec'
```

Limit user u1 to 5 KB/second for all operations on table t2. User u1 can still issue unlimited requests for other tables, regardless of type of operation:

```
hbase> set_quota TYPE => THROTTLE, USER => 'u1', TABLE => 't2', LIMIT => '5K
/min'
```

Limit request to namespaces:

```
hbase> set_quota TYPE => THROTTLE, NAMESPACE => 'ns1', LIMIT => '10req/sec'
```

Limit request to tables:

```
hbase> set_quota TYPE => THROTTLE, TABLE => 't1', LIMIT => '10M/sec'
```

Limit requests based on type, regardless of users, namespaces, or tables:

```
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => WRITE, TABLE => 't1', LI
MIT => '10M/sec'
```

Examples of Listing Throttle Quotas Commands

Show all quotas:

```
hbase> list_quotas
```

Show all quotas applied to user bob:

```
hbase> list_quotas USER => 'bob.*'
```

Show all quotas applied to user bob and filter by table or namespace:

```
hbase> list_quotas USER => 'bob.*', TABLE => 't1'
hbase> list_quotas USER => 'bob.*', NAMESPACE => 'ns.*'
```

Show all quotas and filter by table or namespace:

```
hbase> list_quotas TABLE => 'myTable'
hbase> list_quotas NAMESPACE => 'ns.*'
```

Examples of Updating and Deleting Throttle Quotas Commands

To update a quota, simply issue a new `set_quota` command. To remove a quota, you can set LIMIT to NONE. The actual quota entry will not be removed, but the policy will be disabled.

```
hbase> set_quota TYPE => THROTTLE, USER => 'u1', LIMIT => NONE
```

```
hbase> set_quota TYPE => THROTTLE, USER => 'u1', NAMESPACE => 'ns2', LIMIT =
> NONE
```

```
hbase> set_quota TYPE => THROTTLE, THROTTLE_TYPE => WRITE, USER => 'u1', LIM
IT => NONE
```

```
hbase> set_quota USER => 'u1', GLOBAL_BYPASS => true
```

Space quotas

Space quotas, also known as filesystem space quotas, limit the amount of stored data. It can be applied at a table or namespace level where table-level quotas take priority over namespace-level quotas.

Space quotas are special in that they can trigger different policies when storage goes above thresholds. The following list describes the policies, and they are listed in order of least strict to most strict:

NO_INSERTS

Prohibits new data from being ingested (for example, data from `Put`, `Increment`, and `Append` operations are not ingested).

NO_WRITES

Performs the same function as NO_INSERTS but `Delete` operations are also prohibited.

NO_WRITES_COMPACTIONS

Performs the same function as NO_INSERTS but compactions are also prohibited.

DISABLE

Diables tables.

Examples of Adding Space Quotas

Add quota with the condition that Insert operations are rejected when table t1 reaches 1 GB of data:

```
hbase> set_quota TYPE => SPACE, TABLE => 't1', LIMIT => '1G', POLICY => NO_I
NSERTS
```

Add quota with the condition that table t2 is disabled when 50 GB of data is exceeded:

```
hbase> set_quota TYPE => SPACE, TABLE => 't2', LIMIT => '50G', POLICY => DIS
ABLE
```

Add quota with the condition that Insert and Delete operations cannot be applied to namespace ns1 when it reaches 50 terabytes of data:

```
hbase> set_quota TYPE => SPACE, NAMESPACE => 'ns1', LIMIT => '50T', POLICY =
> NO_WRITES
```

Listing Space Quotas

See "Examples of Listing Throttle Quotas Commands" above for the supported syntax.

Examples of Updating and Deleting Space Quotas

A quota can be removed by setting LIMIT to NONE.

```
hbase> set_quota TYPE => SPACE, TABLE => 't1', LIMIT => NONE
```

```
hbase> set_quota TYPE => SPACE, NAMESPACE => 'ns1', LIMIT => NONE
```

Quota enforcement

When a quota limit is exceeded, the Master server instructs RegionServers to enable an enforcement policy for the namespace or table that violated the quota.

It is important to note the storage quota is not reported in real-time. There is a window when threshold is reached on RegionServers but the threshold accounted for on the Master server is not updated.

Note:

Set a storage limit lower than the amount of available disk space to provide extra buffer.

Quota violation policies

If quotas are set for the amount of space each HBase tenant can fill on HDFS, then a quota violation policy should be planned and implemented on the system.

When a quota violation policy is enabled, the table owner should not be allowed to remove the policy. The expectation is that the Master automatically removes the policy. However, the HBase superuser should still have permission.

Automatic removal of the quota violation policy after the violation is resolved can be accomplished via the same mechanisms that it was originally enforced. But the system should not immediately disable the violation policy when the violation is resolved.

The following describes quota violation policies that you might consider.

Disabling Tables

This is the “brute-force” policy, disabling any tables that violated the quota. This policy removes the risk that tables over quota affect your system. For most users, this is likely not a good choice as most sites want READ operations to still succeed.

One hypothetical situation when a disabling tables policy might be advisable is when there are multiple active clusters hosting the same data and, because of a quota violation, it is discovered that one copy of the data does not have all of the data it should have. By disabling tables, you can prevent further discrepancies until the administrator can correct the problem.

Rejecting All WRITE Operations, Bulk Imports, and Compactions

This policy rejects all WRITEs and bulk imports to the region which the quota applies. Compactions for this region are also disabled to prevent the system from using more space because of the temporary space demand of a compaction. The only resolution in this case is administrator intervention to increase the quota that is being exceeded.

Rejecting All WRITE Operations and Bulk Imports

This is the same as the previous policy, except that compactions are still allowed. This allows users to set or alter a TTL on table and then perform a compaction to reduce the total used space. Inherently, using this violation policy means that you let used space to slightly rise before it is ultimately reduced.

Allowing DELETE Operations But Rejecting WRITE Operations and Bulk Imports

This is another variation of the two previously listed policies. This policy allows users to run processes to delete data in the system. Like the previous policy, using this violation policy means that you let used space slightly rises before it is ultimately reduced. In this case, the deletions are propagated to disk and a compaction actually removes data previously stored on disk. TTL configuration and compactions can also be used to remove data.

Impact of quota violation policy

Quota violation policies can impact live write access, bulk write access, and read access. You must understand what the quota violation policies mean for your deployment before you plan and implement it on your system.

Live write access

You must understand how a quota violation policy configuration affects your ability to write data to HBase.

Every violation policy disables the ability to write new data into the system. This means that any operation on the data other than DELETE operation could be rejected by HBase.

Bulk Write Access

Bulk loading HFiles can be an extremely effective way to increase the overall throughput of ingest into HBase. Quota management when bulk loading is important because large HFiles have the potential to quickly violate a quota.

Clients group HFiles by region boundaries and send the file for each column family to the RegionServer presently hosting that region. The RegionServer ultimately inspects each file, ensuring that it should be loaded into this region, and then, sequentially, load each file into the correct column family.

As a part of the precondition-check of the file’s boundaries before loading it, the quota state should be inspected to determine if loading the next file will violate the quota. If the RegionServer determines that it will violate the quota, it should not load the file and inform the client that the file was not loaded because it would violate the quota.

Read access

In most cases, quota violation policies can affect the ability to read the data stored in HBase. You must understand how a quota violation policy affects your ability to read the data stored in HBase.

In most cases, quota violation policies can affect the ability to read the data stored in HBase. A goal of applying these HBase quotas is to ensure that the storage remains healthy and sustains a higher level of availability to HBase users. Guaranteeing that there is always free space in your HDFS can yield a higher level of health of the physical machines and the DataNodes. This leaves the HDFS-reserved space percentage as a fail-safe mechanism.

Metrics and Insight

You can view the quotas and metrics about the quotas in the HBase Master user interface. The list of defined quotas are displayed along with those quotas whose violation policy is being enforced.

The list of tables/namespaces with enforced violation policies is also available in the the JMX metrics exposed by the Master.

Examples of overlapping quota policies

With the ability to define a quota policy on namespaces and tables, you have to define how the policies are applied. A table quota should take precedence over a namespace quota.

Scenario 1

For example, consider Scenario 1, which is outlined in the following table. Namespace n has the following collection of tables: n1.t1, n1.t2, and n1.t3. The namespace quota is 100 GB. Because the total storage required for all tables is less than 100 GB, each table can accept new WRITEs.

Table 3: Scenario 1: Overlapping Quota Policies

Object	Quota	Storage Utilization
Namespace n1	100 GB	80 GB
Table n1.t1	10 GB	5 GB
Table n1.t2	(not set)	50 GB
Table n1.t3	(not set)	25 GB

Scenario 2

In Scenario 2, as shown in the following table, WRITEs to table n1.t1 are denied because the table quota is violated, but WRITEs to tablen1.t2 and table n1.t3 are still allowed because they are within the namespace quota. The violation policy for the table quota on table n1.t1 is enacted.

Table 4: Scenario 2: Overlapping Quota Policies

Object	Quota	Storage Utilization
Namespace n1	100 GB	60 GB
Table n1.t1	10 GB	15 GB
Table n1.t2	(not set)	30 GB
Table n1.t3	(not set)	15 GB

Scenario 3

In the Scenario 3 table below, WRITEs to all tables are not allowed because the storage utilization of all tables exceeds the namespace quota limit. The namespace quota violation policy is applied to all tables in the namespace.

Table 5: Scenario 3: Overlapping Quota Policies

Object	Quota	Storage Utilization
Namespace n1	100 GB	108 GB
Table n1.t1	10 GB	8 GB
Table n1.t2	(not set)	50 GB
Table n1.t3	(not set)	50 GB

Scenario 4

In the Scenario 4 table below, table n1.t1 violates the quota set at the table level. The table quota violation policy is enforced. In addition, the disk utilization of table n1t1 plus the sum of disk utilization for table n1t2 and table n1t3 exceeds the 100 GB namespace quota. Therefore, the namespace quota violation policy is also applied.

Table 6: Scenario 4: Overlapping Quota Policies

Object	Quota	Storage Utilization
Namespace n1	100 GB	115 GB
Table n1.t1	10 GB	15 GB
Table n1.t2	(not set)	50 GB
Table n1.t3	(not set)	50 GB

Number-of-Tables Quotas

The number-of-tables quota is set as part of the namespace metadata and does not involve the `set_quota` command.

Examples of Commands Relevant to Setting and Administering Number-of-Tables Quotas

Create namespace ns1 with a maximum of 5 tables

```
hbase> create_namespace 'ns1', {'hbase.namespace.quota.maxtables'=>'5'}
```

Alter an existing namespace ns1 to set a maximum of 8 tables

```
hbase> alter_namespace 'ns1', {METHOD => 'set', 'hbase.namespace.quota.maxta
bles'=>'8'}
```

Show quota information for namespace ns1

```
hbase> describe_namespace 'ns1'
```

Alter existing namespace ns1 to remove a quota

```
hbase> alter_namespace 'ns1', {METHOD => 'unset', NAME=>'hbase.namespace.quo
ta.maxtables'}
```

Number-of-Regions Quotas

The number-of-regions quota is similar to the number-of-tables quota. The number-of-regions quota is set as part of the namespace metadata and does not involve the `set_quota` command.

Examples of Commands Relevant to Setting and Administering Number-of-Regions Quotas

Create namespace ns1 with a maximum of 5 tables

```
hbase> create_namespace 'ns1', {'hbase.namespace.quota.maxregions'=>'5'}
```

Alter an existing namespace ns1 to set a maximum of 8 regions

```
hbase> alter_namespace 'ns1', {METHOD => 'set', 'hbase.namespace.quota.maxre
gions'=>'8'}
```

Show quota information for namespace ns1

```
hbase> describe_namespace 'ns1'
```

Alter existing namespace ns1 to remove a quota

```
hbase> alter_namespace 'ns1', {METHOD => 'unset', NAME=>'hbase.namespace.quo
ta.maxregions'}
```

Using HBase scanner heartbeat

A scanner heartbeat check enforces a time limit on the execution of scan RPC requests. This helps prevent scans from taking too long and causing a timeout at the client.

When the server receives a scan RPC request, a time limit is calculated to be half of the smaller of two values: hbase.client.scanner.timeout.period and hbase.rpc.timeout (which both default to 60000 milliseconds, or one minute). When the time limit is reached, the server returns the results it has accumulated up to that point. This result set may be empty. If your usage pattern includes that scans will take longer than a minute, you can increase these values.

To make sure the timeout period is not too short, you can configure hbase.cells.scanned.per.heartbeat.check to a minimum number of cells that must be scanned before a timeout check occurs. The default value is 10000. A smaller value causes timeout checks to occur more often.

Configure the scanner heartbeat using Cloudera Manager

You can configure the HBase scanner heartbeat using Cloudera Manager.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select HBase or HBase Service-Wide.
4. Locate the RPC Timeout property by typing its name in the Search box, and edit the property.
5. Locate the HBase RegionServer Lease Period property by typing its name in the Search box, and edit the property.
6. Enter a Reason for change, and then click Save Changes to commit the changes.
7. Restart the role.
8. Restart the service.

Storing medium objects (MOBs)

Medium Object Storage (MOB) is a feature in Apache HBase that helps you store medium-size objects in the size of 100 KB to 10 MB. You can use this to feature to store documents, images, and other moderately-sized objects.

Data comes in many sizes. You can save different kinds of data in HBase, including binary objects such as images and documents. HBase can technically handle binary objects with cells that are up to 10 MB. However, HBase normal read and write paths are optimized for values smaller than 100 KB. When HBase handles a large number of objects up to 10 MB, the performance is degraded because of write amplification caused by splits and compactions. MOB operates by storing a reference of the object data within the main table. The reference in the table points to external HFiles that contain the actual data, which can be on any storage.

MOB support must be enabled on individual column families within a table. You can do this either through the HBase shell or using the Java API. MOB settings can be configured at table creation time or can be modified on an existing table’s column family.

Cloudera OpDB has a new feature called distributed MOB compaction. This feature overcomes a drawback of the older implementations of MOB compaction by moving maintenance of MOB data files from a centralized process handled by the HBase Master to a parallel process that is distributed across the RegionServers.

If you are currently using MOB compaction in an older version of CDP Runtime, CDH, or HDP, you must be aware of the following changes when you use distributed MOB compaction in CDP:

- You can no longer set MOB compaction policies
- The storage of MOB values is no longer grouped by the date of the original cell's timestamp according to said compaction policies, daily, or otherwise. Instead, they are grouped by the region that performed the most recent maintenance write of the backing MOB data file.
- The MOB system no longer tracks the deletion of individual cells through the use of special files in the MOB storage area with the suffix_del. After upgrading, you must manually move these files.
- Under the default configuration, the MOB system attempts to maximize the throughput for the compaction of MOB stored values. This means that it will take much less time to perform a given compaction of MOB stored values. However, this change places a much larger load on the underlying filesystem when compared to the HBase Master handled MOB compaction.
- When the MOB system detects that a table has HFiles with references to the MOB data but the reference HFiles do not yet have the needed file-level metadata then it does not archive any MOB HFiles from that table. The reference files will be updated as a part of normal HBase maintenance operations over time.

Prerequisites

You must be aware of the following prerequisites before you configure HBase to store MOBs.

Before you configure HBase to store MOBs, you need:

- HBase superuser privileges
- HFile version 3

Configure columns to store MOBs

You can configure a column to store MOBs using the HBase Shell or the Java API.

About this task

Use the following options in the HBase Shell or Java API to configure a column to store MOBs:

- IS_MOB specifies whether or not the column can store MOBs. This is a Boolean option where you can set it to true or false.
- MOB_THRESHOLD configures the number of bytes at which an object is considered to be a MOB. If you do not specify a value for MOB_THRESHOLD, the default is 100 KB. If you write a value larger than this threshold, it is treated as a MOB.

Procedure

- Using the HBase Shell:

```
hbase> create 't1', {NAME => 'f1', IS_MOB => true, MOB_THRESHOLD => 1024
00}
hbase> alter 't1', {NAME => 'f1', IS_MOB => true, MOB_THRESHOLD => 102400}
```

- Usin the Java API:

```
HColumnDescriptor hcd = new HColumnDescriptor(“f”);
hcd.setMobEnabled(true);
hcd.setMobThreshold(102400L);
```

Configure the MOB cache using Cloudera Manager

To configure the MOB cache within Cloudera Manager, edit the HBase Service advanced configuration snippet for the cluster. Cloudera recommends testing your configuration with the default settings first.

Procedure

1. Go to the HBase service.
2. Click Configuration.
3. Search for the property HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml.
4. Paste your configuration into the Value field and save your changes. The following example sets the hbase.mob.cache.evict.period property to 5000 seconds. See *MOB Cache Properties* for a list of configurable properties.

```
<property>
  <name>hbase.mob.cache.evict.period</name>
  <value>5000</value>
</property>
```

5. Restart your cluster for the changes to take effect.

Related Information

MOB cache properties

Test MOB storage and retrieval performance

You can test the MOB storage and retrieval performance by using the Apache HBase Load Test Tool (ltt), and by configuring it to generate MOB values.

About this task

Procedure

- Run this HBase load test tool command in your command-line interface:

```
$ hbase ltt -mob_threshold 102400 -generator \
org.apache.hadoop.hbase.util.LoadTestDataGeneratorWithMOB:medium_column:1
02400:104857 \
    -tn example_table -families small_column,medium_column
    -num_keys 10000 -write 3:1024
```

Important: This command must be run by a user who has the permission to create the passed table, or the table must be created earlier and the user should be given the required permissions.

  - -mob_threshold is the size (in bytes) at which an object is considered to be a MOB. You must set this threshold for the tool to configure column families as MOB-enabled.
  - -generator is the class that generates load for the tool. You must set this parameter to tell the tool how the values must be created. For example, when you set this as LoadTestDataGeneratorWithMOB:medium_column:102400:104857, the parameter accepts the name of a column family, and a range of data sizes for that column. This option pseudo-randomly chooses to make cells in that column family that are of this size range (102400:104857, rather than the default cell size.
  - -tn is the name of the table to create, and use
  - -families is a list of column families to create on the table you provided in using the -tn parameter.
  - -num_keys is the number of rows to write.

- -write is the upper bound on the number of cells to write per column family and the average size of those cells.

MOB cache properties

Opening a MOB file places the corresponding HFile-formatted data in active memory. Too many open MOB files can cause a RegionServer to exceed the memory capacity and cause performance degradation. To minimize the possibility of this issue arising on a RegionServer, you must tune the MOB file reader cache to a size that HBase can scale.

The MOB file reader cache is a least recently used (LRU) cache that keeps only the most recently used MOB files open. Refer to the MOB Cache Properties table for variables that can be tuned in the cache. MOB file reader cache configuration is specific to each RegionServer, so assess and change, if needed, each RegionServer individually. You must manually add any of the following properties you may require in the HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property.

The following properties are available for tuning the HBase MOB cache.

Property	Default	Description
hbase.mob.file.cache.size	1000	The opened file handlers to cache. A larger value benefits read operations by providing more file handlers per MOB file cache and reduces frequent file opening and closing. However, if the value is too high, errors such as "Too many opened file handlers" might be logged.
hbase.mob.cache.evict.period	3600	The amount of time in seconds after a file is opened before the MOB cache evicts cached files.
hbase.mob.cache.evict.remain.ratio	0.5f	The ratio expressed as a float between 0.0 and 1.0, that controls how many files remain cached after an eviction is triggered due to the number of cached files exceeding the value assigned to the hbase.mob.file.cache.size property.

Related Information

Configure the MOB cache using Cloudera Manager

Limiting the speed of compactions

You can limit the speed at which HBase compactions run, by configuring hbase.regionserver.throughput.controller and its related settings.

The default controller is org.apache.hadoop.hbase.regionserver.throttle.PressureAwareCompactionThroughputController, which uses the following algorithm:

- If compaction pressure is greater than 1.0, there is no speed limitation.
- In off-peak hours, use a fixed throughput limitation, configured using hbase.hstore.compaction.throughput.offpeak, hbase.offpeak.start.hour, and hbase.offpeak.end.hour.
- In normal hours, the max throughput is tuned between hbase.hstore.compaction.throughput.higher.bound and hbase.hstore.compaction.throughput.lower.bound (which default to 20 MB/sec and 10 MB/sec respectively), using the following formula, where compactionPressure is between 0.0 and 1.0. The compactionPressure refers to the number of store files that require compaction.

```
lower + (higher - lower) * compactionPressure
```

To disable compaction speed limits, set hbase.regionserver.throughput.controller to org.apache.hadoop.hbase.regionserver.throttle.NoLimitThroughputController.

Configure the compaction speed using Cloudera Manager

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select HBase or HBase Service-Wide.
4. Search for HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml. Paste the relevant properties from the following example into the field and modify the values as needed:

```
<property>
  <name>hbase.hstore.compaction.throughput.higher.bound</name>
  <value>52428800</value>
  <description>The default is 50 MiB/sec</description>
</property>
<property>
  <name>hbase.hstore.compaction.throughput.lower.bound</name>
  <value>104857600</value>
  <description>The default is 100 MiB/sec</description>
</property>
<property>
  <name>hbase.hstore.compaction.throughput.offpeak</name>
  <value>104857600</value>
  <description>The default is 100 MiB/sec</description>
</property>
<property>
  <name>hbase.offpeak.start.hour</name>
  <value>20</value>
  <description>The start of off-peak hours, expressed as an integer betwe
en 0 and 23, inclusive.
      Set to -1 to disable off-peak.</description>
</property>
<property>
  <name>hbase.offpeak.end.hour</name>
  <value>6</value>
  <description>The end of off-peak hours, expressed as an integer between
 0 and 23, inclusive. Set
      to -1 to disable off-peak.</description>
</property>
<property>
  <name>hbase.hstore.compaction.throughput.tune.period</name>
  <value>60000</value>
</property>
```

5. Enter a Reason for change, and then click Save Changes to commit the changes.
6. Restart the service.

Enable HBase indexing

You can enable HBase indexing using Cloudera Manager.

Before you begin

HBase indexing is dependent on the Key-Value Store Indexer service.

Procedure

1. Go to the HBase service.
2. Select ScopeHBASE-1 (Service Wide)
3. Select CategoryBackup.
4. Select the Enable Replication and Enable Indexing properties.
5. Click Save Changes.

Using HBase coprocessors

You can configure HBase coprocessors to run your own custom code. The HBase coprocessor framework provides a way to extend HBase with custom functionality. Coprocessors provide a way to run server-level code against locally-stored data.

Coprocessors are not designed to be used by end users of HBase, but by HBase developers who need to add specialized functionality to HBase. One example of the use of coprocessors is pluggable compaction and scan policies.

Related Information

Apache HBase blog: Coprocessor Introduction

Add a custom coprocessor

You can add a custom coprocessor to to extend HBase with custom functionality using Cloudera Manager.

About this task

To configure these properties in Cloudera Manager:

Procedure

1. Select the HBase service.
2. Click the Configuration tab.
3. Select Scope All .
4. Select Category All .
5. Type HBase Coprocessor in the Search box.
6. You can configure the values of the following properties:
   - HBase Coprocessor Abort on Error (Service-Wide)
   - HBase Coprocessor Master Classes (Master Default Group)
   - HBase Coprocessor Region Classes (RegionServer Default Group)
7. Enter a Reason for change, and then click Save Changes to commit the changes.

Disable loading of coprocessors

You can disable loading of coprocessors using Cloudera Manager.

About this task

Cloudera recommends against disabling loading of system coprocessors, because HBase security functionality is implemented using system coprocessors. However, disabling loading of user coprocessors may be appropriate.

Procedure

1. Select the HBase service.
2. Click the Configuration tab.
3. Search for HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml.
4. To disable loading of all coprocessors, add a new property with the name hbase.coprocessor.enabled and set its value to false. Cloudera does not recommend this setting.
5. To disable loading of user coprocessors, add a new property with the name hbase.coprocessor.user.enabled and set its value to false.
6. Enter a Reason for change, and then click Save Changes to commit the changes.

Configuring HBase MultiWAL

You can configure multiple write-ahead logs (MultiWAL) for HBase. If you do not configure MultiWAL, each region on a RegionServer writes to the same WAL.

A busy RegionServer might host several regions, and each write to the WAL is serial because HDFS only supports sequentially written files. This causes the WAL to negatively impact performance.

MultiWAL allows a RegionServer to write multiple WAL streams in parallel by using multiple pipelines in the underlying HDFS instance, which increases total throughput during writes.

Note: In the current implementation of MultiWAL, incoming edits are partitioned by Region. Therefore, throughput to a single Region is not increased.

To configure MultiWAL for a RegionServer, set the value of the property hbase.wal.provider to multiwal and restart the RegionServer. To disable MultiWAL for a RegionServer, unset the property and restart the RegionServer.

RegionServers using the original WAL implementation and those using the MultiWAL implementation can each handle recovery of either set of WALs, so a zero-downtime configuration update is possible through a rolling restart.

Configuring MultiWAL support using Cloudera Manager

You can configure MultiWAL using Cloudera Manager.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope RegionServer .
4. Select Category Main .
5. Set WAL Provider to MultiWAL.
6. Set the Per-RegionServer Number of WAL Pipelines to a value greater than 1.
7. Enter a Reason for change, and then click Save Changes to commit the changes.
8. Restart the RegionServer roles.

Configuring the storage policy for the Write-Ahead Log (WAL)

You can configure the preferred HDFS storage policy for HBase's write-ahead log (WAL) replicas. This feature allows you to tune HBase's use of SSDs to your available resources and the demands of your workload.

These instructions assume that you have followed the instructions to configure storage directories for DataNodes, and that your cluster has SSD storage available to HBase. If HDFS is not configured to use SSDs, these configuration changes will have no effect on HBase. The following policies are available:

- NONE: no preference about where the replicas are written.
- ONE_SSD: place one replica on SSD storage and the remaining replicas in default storage. This allows you to derive some benefit from SSD storage even if it is a scarce resource in your cluster.

  Warning: ONE_SSD mode has not been thoroughly tested with HBase and is not recommended.
- ALL_SSD: place all replicas on SSD storage.

Configure the storage policy for WALs using Cloudera Manager

You can configure the preferred HDFS storage policy for HBase's write-ahead log (WAL) replicas using Cloudera Manager.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Search for the property WAL HSM Storage Policy.
4. Select your desired storage policy.
5. Save your changes. Restart all HBase roles.

Results

Changes will take effect after the next major compaction.

Configure the storage policy for WALs using the Command Line

You can configure the preferred HDFS storage policy for HBase's write-ahead log (WAL) replicas using the command line.

About this task

Important: Follow these command-line instructions on systems that do not use Cloudera Manager.

Procedure

- Paste the following XML into hbase-site.xml. Uncomment the <value> line that corresponds to your desired storage policy.

```
<property>
  <name>hbase.wal.storage.policy</name>
  <value>NONE</value>
```

```
  <!--<value>ONE_SSD</value>-->
  <!--<value>ALL_SSD</value>-->
</property>
```

Warning: ONE_SSD mode has not been thoroughly tested with HBase and is not recommended.

- Restart HBase. Changes will take effect for a given region during its next major compaction.

Using RegionServer grouping

You can use RegionServer Grouping (rsgroup) to impose strict isolation between RegionServers by partitioning RegionServers into distinct groups. You can use HBase Shell commands to define and manage RegionServer Grouping.

You must first create an rsgroup before you can add RegionServers to it. Once you have created an rsgroup, you can move your HBase tables into this rsgroup so that only the RegionServers in the same rsgroup can host the regions of the table.

Note: RegionServers and tables can only belong to one rsgroup at a time. By default, all the tables and RegionServers belong to the default rsgroup.

A custom balancer implementation tracks assignments per rsgroup and moves regions to the relevant RegionServers in that rsgroup. The rsgroup information is stored in a regular HBase table, and a ZooKeeper-based read-only cache is used at cluster bootstrap time.

Enable RegionServer grouping using Cloudera Manager

You must use Cloudera Manager to enable RegionServer Grouping before you can define and manage rsgroups.

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope Master .
4. Locate the HBase Coprocessor Master Classes property or search for it by typing its name in the Search box.
5. Add the following property value: org.apache.hadoop.hbase.rsgroup.RSGroupAdminEndpoint.
6. Locate the Master Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property or search for it by typing its name in the Search box.
7. Click View as XML and add the following property:

```
<property>
  <name>hbase.master.loadbalancer.class</name>
  <value>org.apache.hadoop.hbase.rsgroup.RSGroupBasedLoadBalancer</value>
</property>
```

8. Enter a Reason for change, and then click Save Changes to commit the changes.
9. Restart the role.
10. Restart the service.

Configure RegionServer grouping

When you add a new rsgroup, you are creating an rsgroup other than the default group.

About this task

To configure a rsgroup, in the HBase shell:

Procedure

1. Add an rsgroup: $hbase> add_rsgroup '*MYGROUP*'.
2. Add RegionServers and tables to this rsgroup: $hbase> move_servers_tables_rsgroup '*MYGROUP*', ['*SERVER1:PORT*','*SERVER2:PORT*'],['*TABLE1*','*TABLE2*'].
3. Run the balance_rsgroup command if the tables are slow to migrate to the group's dedicated server.

Note: The term *rsgroup* refers to servers in a cluster with only the hostname and port. It does not make use of the HBase ServerName type identifying RegionServers (hostname + port + start time) to distinguish RegionServer instances.

Monitor RegionServer grouping

You can monitor the status of the commands using the Tables tab on the HBase Master UI home page.

You can monitor the status of the commands using the Tables tab on the HBase Master UI home page. If you click on a table name, you can see the RegionServers that are deployed.

You must manually align the RegionServers referenced in rsgroups with the actual state of nodes in the cluster that is active and running.

Remove a RegionServer from RegionServer grouping

You can remove a RegionServer by moving it to the default rsgroup. Edits made using shell commands to all rsgroups, except the default rsgroup, are persisted to the system hbase:rsgroup table. If an rsgroup references a decommissioned RegionServer, then the rsgroup should be updated to undo the reference.

Procedure

1. Move the RegionServer to the default rsgroup using the command: $hbase> move_servers_rsgroup 'default', ['*SERVER1:PORT*'].
2. Check the list of RegionServers in your rsgourp to ensure that that the RegionServer is successfully removed using the command: $hbase> get_rsgroup '*MYGROUP*'

   The default rsgroup's RegionServer list mirrors the current state of the cluster. If you shut down a RegionServer that was part of the default rsgroup, and then run the get_rsgroup 'default' command to list its content in the shell, the server is no longer listed. If you move the offline server from the non-default rsgroup to default, it will not show in the default list; the server will just be removed from the list.

Enabling ACL for RegionServer grouping

You can enable ACL for RegionServer grouping using Cloudera Manager. If authorization is enabled only Global Admins can manage rsgroups.

Procedure

1. In Cloudera Manager navigate to HBase Configuration .
2. Find the Master Advanced Configuration Snippet (Safety-Valve) for hbase-site.xml property.
3. Click the plus icon to add a new property:
   - Name: hbase.security.authorization
   - Value: true

4. Click Save Changes.
5. Restart your HBase Master server.

Best practices when using RegionServer grouping

You must keep in mind the following best practices when using rsgroups.

- Isolate system tables: You can either have a system rsgroup where all the system tables are present or just leave the system tables in default rsgroup and have all user-space tables in non-default rsgroups.
- Handle dead nodes: You can have a special rsgroup of dead or questionable nodes to help you keep them without running until the nodes are repaired. Be careful when replacing dead nodes in an rsgroup, and ensure there are enough live nodes before you start moving out the dead nodes. You can move the good live nodes first before moving out the dead nodes.

  If you have configured a table to be in a rsgroup, but all the RegionServers in that rsgroup die, the tables become unavailable and you can no longer access those tables.

Disable RegionServer grouping

When you no longer require rsgroups, you can disable it for your cluster.

About this task

Removing RegionServer Grouping for a cluster on which it was enabled involves more steps in addition to removing the relevant properties from hbase-site.xml. You must ensure that you clean the RegionServer grouping-related metadata so that if the feature is re-enabled in the future, the old metadata will not affect the functioning of the cluster.

To disable RegionServer Grouping:

Procedure

1. Move all the tables in non-default rsgroups to default RegionServer group.

```
#Reassigning table t1 from the non-default group - hbase shell
hbase> move_tables_rsgroup 'default',['T1']
```

2. Move all RegionServers in non-default rsgroups to default regionserver group.

```
#Reassigning all the servers in the non-default rsgroup to default - hbase
 shell
hbase> move_servers_rsgroup 'default',
['REGIONSERVER1:PORT','REGIONSERVER2:PORT','REGIONSERVER3:PORT']
```

3. Remove all non-default rsgroups. default rsgroup created implicitly does not have to be removed.

```
#removing non-default rsgroup - hbase shell
hbase> remove_rsgroup 'MYGROUP'
```

4. Remove the changes made in hbase-site.xml and restart the cluster.
5. Drop the table hbase:rsgroup from HBase.

```
#Through hbase shell drop table hbase:rsgroup
hbase> disable 'hbase:rsgroup'
0 row(s) in 2.6270 seconds
hbase> drop 'hbase:rsgroup'
0 row(s) in 1.2730 seconds
```

6. Remove the znode rsgroup from the cluster ZooKeeper using zkCli.sh.

```
#From ZK remove the node /hbase/rsgroup through zkCli.sh
rmr /hbase/rsgroup
```

Optimizing HBase I/O

You can optimize HBase I/O using several ways. Two HBase key concepts that helps you in the process are BlockCache and MemStore tuning.

The information in this section is oriented toward basic BlockCache and MemStore tuning. As such, it describes only a subset of cache configuration options. HDP supports additional BlockCache and MemStore properties, as well as other configurable performance optimizations such as remote procedure calls (RPCs), HFile block size settings, and HFile compaction. For a complete list of configurable properties, see the hbase-default.xml source file in GitHub.

HBase I/O components

The concepts related to HBase file operations and memory (RAM) caching are HFile, Block, BlockCache, MemStore and Write Ahead Log (WAL).

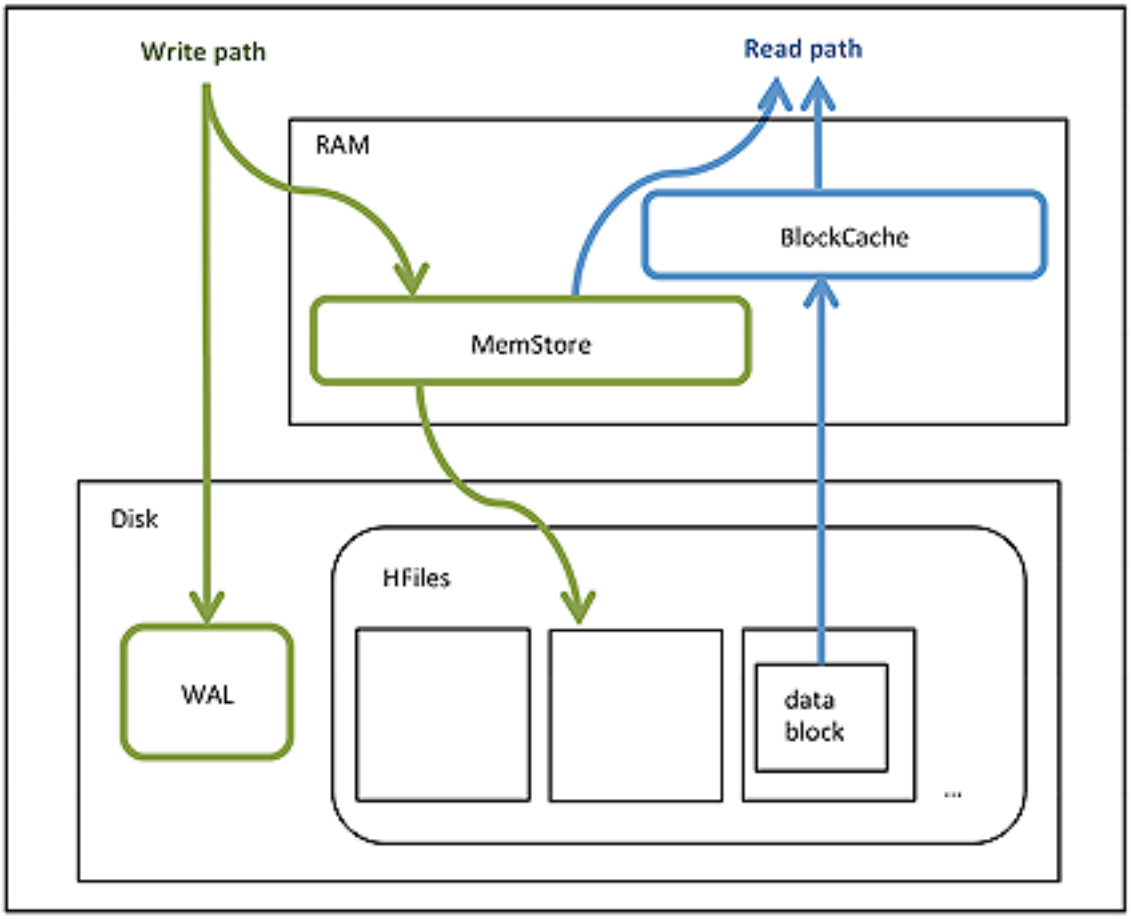
HBase Component	Description
HFile	An HFile contains table data, indexes over that data, and metadata about the data.
Block	An HBase block is the smallest unit of data that can be read from an HFile. Each HFile consists of a series of blocks. (Note: an HBase block is different from an HDFS block or other underlying file system blocks.)
BlockCache	BlockCache is the main HBase mechanism for low-latency random read operations. BlockCache is one of two memory cache structures maintained by HBase. When a block is read from HDFS, it is cached in BlockCache. Frequent access to rows in a block cause the block to be kept in cache, improving read performance.
MemStore	MemStore ("memory store") is in-memory storage for a RegionServer. MemStore is the second of two cache structures maintained by HBase. MemStore improves write performance. It accumulates data until it is full, and then writes ("flushes") the data to a new HFile on disk. MemStore serves two purposes: it increases the total amount of data written to disk in a single operation, and it retains recently written data in memory for subsequent low-latency reads.
Write Ahead Log (WAL)	The WAL is a log file that records all changes to data until the data is successfully written to disk (MemStore is flushed). This protects against data loss in the event of a failure before MemStore contents are written to disk.

HBase Read/Write Operations

BlockCache and MemStore reside in random-access memory (RAM). HFiles and the Write Ahead Log are persisted to HDFS.

The following figure shows these simplified write and read paths:

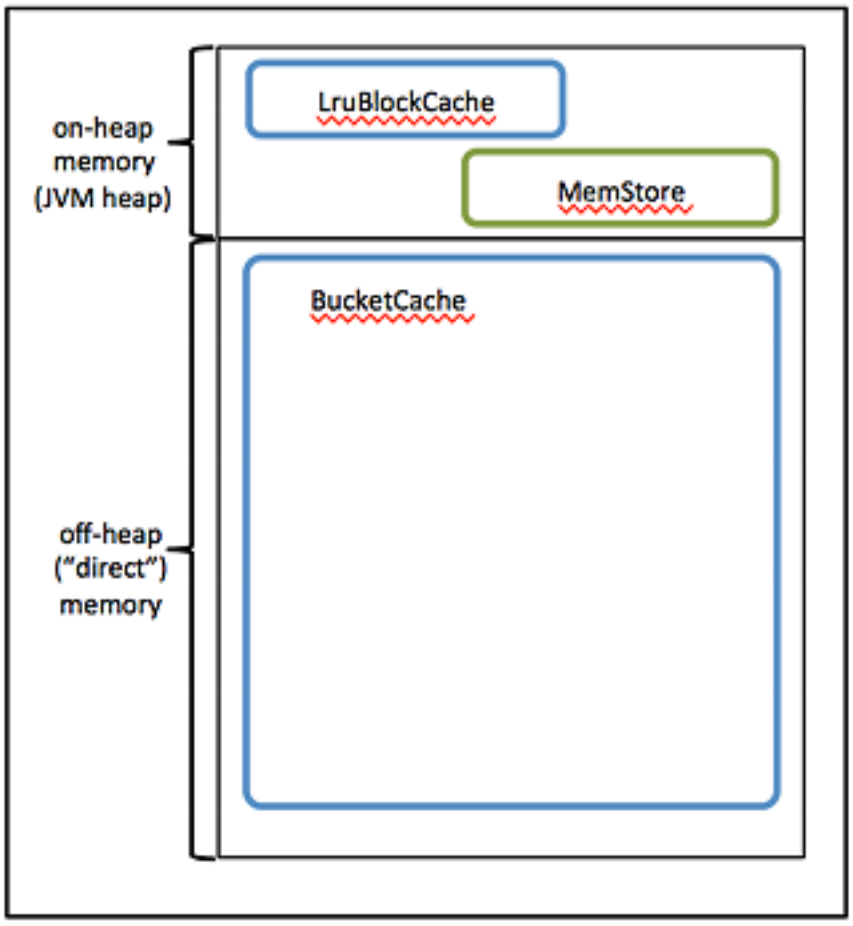
- During write operations, HBase writes to WAL and MemStore. Data is flushed from MemStore to disk according to size limits and flush interval.
- During read operations, HBase reads the block from BlockCache or MemStore if it is available in those caches. Otherwise, it reads from disk and stores a copy in BlockCache.



By default, BlockCache resides in an area of RAM that is managed by the Java Virtual Machine (JVM) garbage collector; this area of memory is known as on-heap memory or the JVM heap. The BlockCache implementation that manages the on-heap cache is called LruBlockCache.

If you have stringent read latency requirements and you have more than 20 GB of RAM available on your servers for use by HBase RegionServers, consider configuring BlockCache to use both on-heap and off-heap memory. BucketCache is the off-heap memory equivalent to LruBlockCache in on-heap memory. Read latencies for BucketCache tend to be less erratic than LruBlockCache for large cache loads because BucketCache (not JVM garbage collection) manages block cache allocation. The MemStore always resides in the on-heap memory.

Figure 3: Relationship among Different BlockCache Implementations and MemStore



- Additional notes:
- BlockCache is enabled by default for all HBase tables.
- BlockCache is beneficial for both random and sequential read operations although it is of primary consideration for random reads.
- All regions hosted by a RegionServer share the same BlockCache.
- You can turn BlockCache caching on or off per column family.

Advanced configuration for write-heavy workloads

HBase includes several advanced configuration parameters for adjusting the number of threads available to service flushes and compactions in the presence of write-heavy workloads. Tuning these parameters incorrectly can severely degrade performance and is not necessary for most HBase clusters. If you use Cloudera Manager, configure these options using the HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml

hbase.hstore.flusher.count

The number of threads available to flush writes from memory to disk. Never increase hbase.hstore .flusher.count to more of 50% of the number of disks available to HBase. For example, if you have 8 solid-state drives (SSDs), hbase.hstore.flusher.count should never exceed 4. This allows scanners and compactions to proceed even in the presence of very high writes.

hbase.regionserver.thread.compaction.large and hbase.regionserver.thread.compaction.small

The number of threads available to handle small and large compactions, respectively. Never increase either of these options to more than 50% of the number of disks available to HBase.

Ideally, hbase.regionserver.thread.compaction.small should be greater than or equal to hbase.region server.thread.compaction.large, since the large compaction threads do more intense work and will be in use longer for a given operation.

In addition to the above, if you use compression on some column families, more CPU will be used when flushing these column families to disk during flushes or compaction. The impact on CPU usage depends on the size of the flush or the amount of data to be decompressed and compressed during compactions.