# Using Apache HBase to store and access data

**Date of Publish:** 2018-08-30

# Contents

# Medium Object (MOB) storage support in Apache HBase...............................35

# Methods to enable MOB storage support..............................................................35

# Method 1:Enable MOB Storage support using configure options in the command line.......................................................................................................... 35

# Method 2: Invoke MOB support parameters in a Java API............................36

# Test the MOB storage support configuration.....................................................36

# MOB storage cache properties...........................................................................37

# HBase quota management.....................................................................................38

# Understanding Apache HBase Hive integration.................................................45

# HBase Best Practices..........................................................................................49

# What's New in Apache HBase

HBase in Hortonworks Data Platform (HDP) 3.0 includes the following new features:

- Procedure V2

  You can use Procedure V2 or procv2, which is an updated framework for executing multi-step, HBase administrative operations when there is a failure. The introduction of this capability is to implement all master operations using procv2 to remove the need for tools like hbck in the future. Use procv2 for creating, modifying and deleting tables. Other systems like new AssignmentManager is implemented using proc-v2.

- Fully off-heap read/write path

  When you write data into HBase through Put operation, the cell objects do not enter JVM heap until the data is flushed to disk in an HFile. This helps to reduce total heap usage of a RegionServer and it copies less data making it more efficient.

- Use of Netty for RPC layer and Async API

  This replaces the old Java NIO RPC server with a Netty RPC server. Netty provides you the ability to easily provide an Asynchronous Java client API.

- In-memory compactions

  Periodic reorganization of the data in the Memstore can result in a reduction of overall I/O, that is data written and accessed from HDFS. The net performance increases when we keep more data in memory for a longer period of time.

- Better dependency management

  HBase now internally shades commonly-incompatible dependencies to prevent issues for downstream users. You can use shaded client jars that will reduce the burden on the existing applications.

- Coprocessor and Observer API rewrite

  There are minor changes made to the API to remove ambiguous, misleading, and dangerous calls.

- Backup/restore

  You can use the built-in tooling in HBase to create full and incremental backups of the HBase data.

# Overview of Apache HBase

Hortonworks Data Platform (HDP) includes the Apache HBase database, which provides random, persistent access to data in Hadoop. This "NoSQL" database is ideal for scenarios that require real-time analysis and tabular data for end-user applications. Apache HBase can host big data tables because it scales linearly to handle very large (petabyte scale), column-oriented data sets. The data store is predicated on a key-value model that supports low latency reads, writes, and updates in a distributed environment.

As a natively nonrelational database, Apache HBase can combine data sources that use a wide variety of structures and schemas. It is natively integrated with the Apache Hadoop Distributed File System (HDFS) for resilient data storage and is designed for hosting very large tables with sparse data.

# Apache HBase installation

When you install Apache HBase as part of HDP distribution, two components that must coexist are Apache Hadoop Distbuted File System (HDFS) as a filesystem and and Apache ZooKeeper for maintaining the stability of the application.

Apache HBase (often simply referred to as HBase) operates with many other big data components of the Apache Hadoop environment. Some of these components might or might not be suitable for use with the HBase deployment in your environment. However, two components that must coexist on your HBase cluster are Apache Hadoop Distributed File System (HDFS) and Apache ZooKeeper. These components are bundled with all HDP distributions.

Apache Hadoop Distributed File System (HDFS) is the persistent data store that holds data in a state that allows users and applications to quickly retrieve and write to HBase tables. While technically it is possible to run HBase on a different distributed filesystem, the vast majority of HBase clusters run with HDFS. HDP uses HDFS as its filesystem.

Apache ZooKeeper (or simply ZooKeeper) is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services in Hadoop ecosystems. ZooKeeper is essential for maintaining stability for HBase applications in the event of node failures, as well as to to store and mediate updates to important configuration information across the Hadoop cluster.

If you want to use a SQL-like interface to work with the semistructured data of an HBase cluster, a good complement to the other Hadoop components is Apache Phoenix (or simply Phoenix). Phoenix is a SQL abstraction layer for interacting with HBase. Phoenix enables you to create and interact with tables in the form of typical DDL and DML statements through its standard JDBC API. HDP supports integration of Phoenix with HBase. See Orchestrating SQL and APIs with Apache Phoenix.

The following table defines some main HBase concepts:

| HBase Concept | Description |
|---|---|
| region | A group of contiguous HBase table rows |
| | Tables start with one region, with regions dynamically added as the table grows. Regions can be spread across multiple hosts to provide load balancing and quick recovery from failure. There are two types of regions: primary and secondary. A secondary region is a replicated primary region located on a different RegionServer. |
| RegionServer | Serves data requests for one or more regions |
| | A single region is serviced by only one RegionServer, but a RegionServer may serve multiple regions. |
| column family | A group of semantically related columns stored together |
| MemStore | In-memory storage for a RegionServer |
| | RegionServers write files to HDFS after the MemStore reaches a configurable maximum value specified with the hbase.hregion.memstore.flush.size property in the hbase-site.xml configuration file. |
| Write Ahead Log (WAL) | In-memory log in which operations are recorded before they are stored in the MemStore |
| compaction storm | A short period when the operations stored in the MemStore are flushed to disk and HBase consolidates and merges many smaller files into fewer large files |
| | This consolidation is called compaction, and it is usually very fast. However, if many RegionServers reach the data limit specified by the MemStore at the same time, HBase performance might degrade from the large number of simultaneous major compactions. You can avoid this by manually splitting tables over time. |

**Related Information**
Orchestrating SQL and APIs with Apache Phoenix

## Installing HBase through Ambari

You can use Ambari installation wizard to install and configure Apache HBase for your HDP cluster.

- Ambari installation wizard

  This wizard is part of the Apache Ambari web-based platform that guides HDP installation, including deploying various Hadoop components, such as HBase, depending on the needs of your cluster. For more information, see the Ambari Install Guide.

  > **Note:**

  Your HBase installation must be the same version as the one that is packaged with the distribution of the HDP stack version that is deployed across your cluster.

  **Note:** Starting with HDP 3.0, the default value of the "yarn.scheduler.capacity.root.<queue>.acl_submit_applications" property is changed to yarn, indicating that only the yarn user can submit applications by default. Any other user (or group) will be able to submit applications to the queue only if the value of the property is explicitly set to that user (or group).

**Related Information**
Ambari Install Guide


## HBase cluster capacity planning

There are several aspects to consider when planning the capacity of an HBase cluster and the size of its RegionServers.

### Configuring HBase cluster for the first time

If you are an administrator, use any of the recommended methods to plan the the capacity of an HBase cluster and the size of its RegionServers.

plan

- Increasing the request handler thread count
- Configuring the size and number of WAL files
- Configuring compactions
- Splitting tables
- Tuning JVM garbage collection in RegionServers

#### Increase the request handler thread count
If you are an administator and your cluster needs to handle high volume of request patterns, increase the number of listeners generated by the RegionServers.

#### Procedure

- In the hbase-site.xmlconfiguration file, increase the hbase.regionserver.handler.count property higher than the default value. The default value is 30.

#### Example

```
<property>
<name>hbase.regionserver.handler.count</name>
<value>30</value>
</property>
```

### Configure the size and number of WAL files

HBase uses the Write Ahead Log, or WAL, to recover MemStore data not yet flushed to disk if a RegionServer crashes. Administrators should configure these WAL files to be slightly smaller than the HDFS block size.

### About this task

By default, an HDFS block is 64 MB and a WAL is approximately 60 MB. You should ensure that enough WAL files are allocated to contain the total capacity of the MemStores. Use the following formula to determine the number of WAL files needed:

(regionserver_heap_size * memstore fraction) / (default_WAL_size)

For example, assume that your environment has the following HBase cluster configuration:

1. 16 GB RegionServer heap
2. 0.4 MemStore fraction
3. 60 MB default WAL size

The formula for this configuration is as follows:

(16384 MB * 0.4) / 60 MB = approximately 109 WAL files

Use the following properties in the hbase-site.xml configuration file to configure the size and number of WAL files:

| Configuration Property | Description | Default |
|---|---|---|
| hbase.regionserver.maxlogs | Sets the maximum number of WAL files | 32 |
| hbase.regionserver.logroll.multiplier | Multiplier of HDFS block size | 0.95 |
| hbase.regionserver.hlog.blocksize | Optional override of HDFS block size | Value assigned to actual HDFS block size |

**Note:**

If recovery from failure takes longer than expected, try reducing the number of WAL files to improve performance.

### Configure compactions

If you are an administrator and expect the HBase clusters to host large amounts of data, consider the effect that compactions have on write throughput. For write-intensive data request patterns, you should consider less frequent compactions and more StoreFiles per region.

### Procedure

- In the hbase-site.xmlconfiguration file, increase the minimum number of files required in hbase.hstore.compaction.minproperty to trigger a compaction.
- If you opt to increase this value, you should also increase the value assigned to the hbase.hstore.blockingStoreFiles property because more files will accumulate.

### Considerations for splitting tables

You can split tables during table creation based on the target number of regions per RegionServer to avoid costly dynamic splitting as the table starts to fill.

Splitting table ensures that the regions in the pre-split table are distributed across many host machines. Pre-splitting a table avoids the cost of compactions required to rewrite the data into separate physical files during automatic splitting.

If a table is expected to grow very large, you should create at least one region per RegionServer. However, you should not immediately split the table into the total number of desired regions. Rather, choose a low to intermediate value. For multiple tables, you should not create more than one region per RegionServer, especially if you are uncertain how large the table will grow. Creating too many regions for a table that will never exceed 100 MB is not useful; a single region can adequately service a table of this size.

**Tune JVM garbage collection in RegionServers**

You can tune garbage collection in HBase RegionServers for stability, because a RegionServer cannot utilize a very large heap due to the cost of garbage collection. Administrators should specify no more than 24 GB for one RegionServer.

**Procedure**

1.  Specify the following configurations in the HBASE_REGIONSERVER_OPTS configuration option in the /conf/hbase-env.sh.

    ```
    -XX:+UseConcMarkSweepGC
    -Xmn2500m (depends on MAX HEAP SIZE, but should not be less than 1g and
     more than 4g)
    -XX:PermSize=128m
    -XX:MaxPermSize=128m
    -XX:SurvivorRatio=4
    -XX:CMSInitiatingOccupancyFraction=50
    -XX:+UseCMSInitiatingOccupancyOnly
    -XX:ErrorFile=/var/log/hbase/hs_err_pid%p.log
    -XX:+PrintGCDetails
    -XX:+PrintGCDateStamps
    ```

2.  Ensure that the block cache size and the MemStore size combined do not significantly exceed 0.5*MAX_HEAP, which is defined in the HBASE_HEAP_SIZE configuration option of the /conf/hbase-env.sh file.

## Node count and JVM configuration

The number of nodes in an HBase cluster is typically driven by physical size of the data set and read/write throughput.

**Physical size of the data**

The physical size of data on disk is affected by factors such as HBase overhead, compression, replication and RegionServer Write Ahead Log (WAL).

| Factor Affecting Size of Physical Data | Description |
| --- | --- |
| HBase Overhead | The default amount of disk space required for a single HBase table cell. Smaller table cells require less overhead. The minimum cell size is 24 bytes and the default maximum is 10485760 bytes. |
|  | You can customize the maximum cell size by using the hbase.client.keyvalue.maxsize property in the hbase-site.xml configuration file. HBase table cells are aggregated into blocks; you can configure the block size for each column family by using the hbase.mapreduce.hfileoutputformat.blocksize property. The default value is 65536 bytes. You can reduce this value for tables with highly random data access patterns if you want to improve query latency. |
| Compression | You should choose the data compression tool that is most appropriate to reducing the physical size of your data on disk. Although HBase is not shipped with LZO due to licensing issues, you can install LZO after installing HBase. GZIP provides better compression than LZO but is slower. HBase also supports Snappy. |
| HDFS Replication | HBase uses HDFS for storage, so replicating HBase data stored in HDFS affects the total physical size of data. A typical replication factor of 3 for all HBase tables in a cluster triples the physical size of the stored data. |
| RegionServer Write Ahead Log (WAL) | The size of the Write Ahead Log, or WAL, for each RegionServer has minimal impact on the physical size of data: typically fixed at less than half of the memory for the RegionServer. The data size of WAL is usually not configured. |

**Read-Write Throughput**

The number of nodes in an HBase cluster might also be driven by required throughput for disk reads and writes. The throughput per node greatly depends on table cell size and data request patterns, as well as node and cluster configuration.

You can use YCSB tools to test the throughput of a single node or a cluster to determine if read/write throughput should drive the number of nodes in your HBase cluster. A typical throughput for write operations for one RegionServer is 5 through 15 MB/s. Unfortunately, there is no good estimate for read throughput, which varies greatly depending on physical data size, request patterns, and hit rate for the block cache.

**Related Information**
YCSB

## Options to increase HBase Region count and size

If you are an administrator, you cannot directly configure the number of regions for a RegionServer, however, you can indirectly increase the number of regions by increasing the size of the MemStore for a RegionServer and the size of the region.

In general, an HBase cluster runs more smoothly with fewer regions. You can also increase the number of regions for a RegionServer by splitting large regions to spread data and the request load across the cluster. HBase enables you to configure each HBase table individually, which is useful when tables have different workloads and use cases. Most region settings can be set on a per-table basis by using HTableDescriptor class, as well as by using the HBase CLI. These methods override the properties in the hbase-site.xml configuration file. For further information, see configure compactions.

> **Note:**
>
> The HDFS replication factor defined in the previous table affects only disk usage and should not be considered when planning the size of regions.

**Related Tasks**
Configure compactions
**Related Information**
HTableDescriptor class

**Increasing MemStore size for RegionServer**

If you are an administrator, you can adjust the size of the MemStore in hbase-site.xml configuration file depending on your need.

Use of the RegionServer MemStore largely determines the maximum number of regions for the RegionServer. Each region has one MemStore for each column family, which grows to a configurable size, usually between 128 and 256 MB. You can specify this size by using the hbase.hregion.memstore.flush.size property in the hbase-site.xml configuration file. The RegionServer dedicates some fraction of total memory to region MemStores based on the value of the hbase.regionserver.global.memstore.size configuration property. If usage exceeds this configurable size, HBase might become unresponsive or compaction storms might occur.

You can use the following formula to estimate the number of regions for a RegionServer:

(regionserver_memory_size) * (memstore_fraction) / ((memstore_size) * (num_column_families))

For example, assume that your environment uses the following configuration:

- RegionServer with 16 GB RAM (or 16384 MB)
- MemStore fraction of .4
- MemStore with 128 MB RAM
- One column family in table

The formula for this configuration is as follows:

(16384 MB * .4) / ((128 MB * 1) = approximately 51 regions

You can adjust the memory consumption of the regions for this example RegionServer by increasing the RAM size of the memstore to 256 MB. The reconfigured RegionServer then has MemStore space for approximately 25 regions, and the HBase cluster should run more smoothly given a uniform distribution of load.

The formula can be used for multiple tables with the same configuration by using the total number of column families in all the tables.

> **Note:**
>
> The formula is based on the assumption that all regions are filled at approximately the same rate. If a fraction of the cluster's regions are written to, divide the result by this fraction.

If the data request pattern is dominated by write operations rather than read operations, you should increase the MemStore fraction. However, this increase negatively impacts the block cache.

### Increasing the size of Region

To indirectly increase the number of regions for a RegionServer, increase the size of the region by using the hbase.hregion.max.filesize property in the hbase-site.xml configuration file. You can increase the number of regions for a RegionServer by increasing the specified size at which new regions are dynamically allocated.

Maximum region size is primarily limited by compactions. Very large compactions can degrade cluster performance. The recommended maximum region size is 10 through 20 GB. For HBase clusters running version 0.90.x, the maximum recommended region size is 4 GB and the default is 256 MB. If you are unable to estimate the size of your tables, you should retain the default value. You should increase the region size only if your table cells tend to be 100 KB or larger.

> **Note:**
>
> HBase 0.98 introduced stripe compactions as an experimental feature that also enables administrators to increase the size of regions. For more information, see Experimental: Stripe Compactions on the Apache HBase website.

### Related Information
Experimental: Stripe Compactions

# Enable multitenancy with namespaces

A namespace is a logical grouping of tables analogous to a database or a schema in a relational database system. With namespaces, a group of users can share access to a set of tables but the users can be assigned different privileges. Similarly, one application can run using the tables in a namespace simultaneously with other applications. Each group of users and each application with access to the instance of the tables defined as a namespace is a tenant.

A namespace can support varying ACL-based security modules that can exist among different tenants. Read/write permissions based on groups and users with access to one instance of the namespace function independently from the permissions in another instance.

Unlike relational databases, HBase table names can contain a dot (.) Therefore, HBase uses different syntax, a colon (:), as the separator between the namespace name and table name. For example, a table with the name store1 in a namespace that is called orders has store1:orders as a fully qualified table name. If you do not assign a table to a namespace, then the table belongs to the special default namespace.

The namespace file, which contains the objects and data for the tables assigned to a namespace, is stored in a subdirectory of the HBase root directory ($hbase.rootdir) on the HDFS layer of your cluster. If $hbase.rootdir is at the default location, the path to the namespace file and table is /apps/hbase/data/data/namespace/table_name.

Example of Namespace Usage

A software company develops applications with HBase. Developers and quality-assurance (QA) engineers who are testing the code must have access to the same HBase tables that contain sample data for testing. The HBase tables with sample data are a subset of all HBase tables on the system. Developers and QA engineers have different goals in their interaction with the tables and need to separate their data read/write privileges accordingly.

By assigning the sample-data tables to a namespace, access privileges can be provisioned appropriately so that QA engineers do not overwrite developers' work and vice versa. As tenants of the sample-data table namespace, when developers and QA engineers are logged in as users of this namespace domain they do not access other HBase tables in different domains. This helps ensure that not every user can view all tables on the HBase cluster for the sake of security and ease-of-use.

## Default HBase namespace actions

**Note:**

If you do not require multitenancy or formalized schemas for HBase data, then do not concern yourself with namespace definitions and assignments. HBase automatically assigns a default namespace when you create a table and do not associate it with a namespace.

The default namespaces are the following:

| | |
|---|---|
| **hbase** | A namespace that is used to contain HBase internal system tables |
| **default** | A namespace that contains all other tables when you do not assign a specific user-defined namespace |

## Define and drop namespaces

You can create and drop namespaces in the HBase shell.

### About this task

**Note:**

You can assign a table to only one namespace, and you should ensure that the table correctly belongs to the namespace before you make the association in HBase. You cannot change the namespace that is assigned to the table later.

The HBase shell has a set of straightforward commands for creating and dropping namespaces. You can assign a table to a namespace when you create the table.

| | |
|---|---|
| **create_namespace 'my_ns'** | Creates a namespace with the name my_ns. |
| **create 'my_ns:my_table', 'fam1'** | Creates my_table with a column family identified as fam1 in the my_ns namespace. |
| **drop_namespace 'my_ns'** | Removes the my_ns namespace from the system. The command only functions when there are no tables with data that are assigned to the namespace. |

## Security features that are available

The following security features are available:

- Cell-level access control lists (cell-level ACLs): These ACLs are supported in tables of HBase 0.98 and later versions.
- Column family encryption: This feature is supported in HBase 0.98 and later versions.

   **Note:**

Cell-level ACLs and column family encryption are considered under development. Do not use these features in your production systems. If you have questions about these features, contact Support by logging a case on the Hortonworks Support Portal.

**Related Information**
Hortonworks Support Portal

# Managing Apache HBase clusters

You will get an understanding on how to manage your HBase clusters.

## Monitoring Apache HBase clusters through Grafana-based dashboard

If you have an Ambari-managed HBase cluster, you can monitor the cluster performance with Grafana-based dashboards.

Grafana dashboards provide graphical visualizations of data distribution and other boilerplate performance metrics. You can hover over and click graphs to focus on specific metrics or data sets, as well as to redraw visualizations dynamically.

The interactive capabilities of the dashboards can help you to discover potential bottlenecks in your system. For example, you can scan the graphs to get an overview of cluster activity and scroll over a particular time interval to enlarge details about the activity in the time frame to uncover when the data load is unbalanced. Another potential use case is to help you examine if RegionServers need to be reconfigured.

For information about how to access the dashboards and for details about what cluster metrics are displayed, see using Grafana Dashboards in Ambari.

**Related Information**
using Grafana Dashboards in Ambari

## Optimizing Apache 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.

**Related Information**
hbase-default.xml source file

### 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. |

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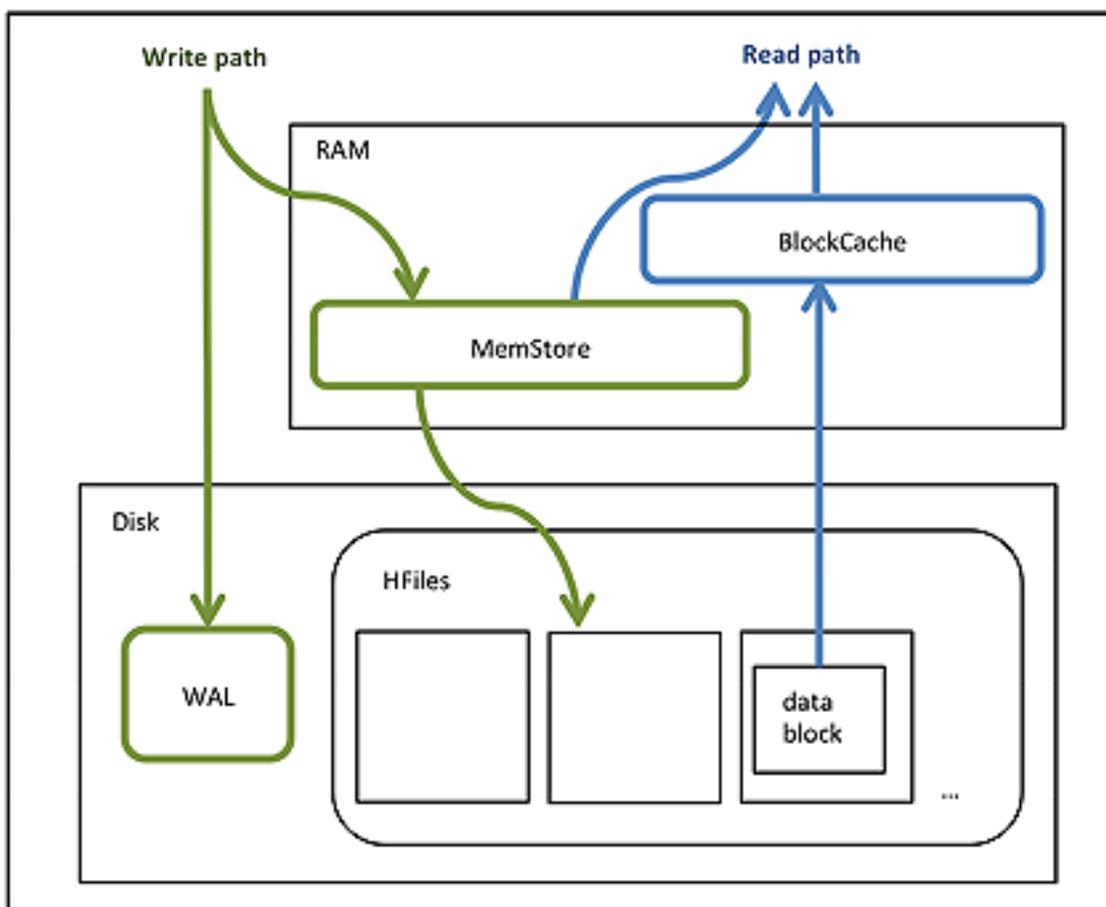
By default, BlockCache resides in an area of RAM that is managed by the Java Virtual Machine (JVM) garbage collector; this area of memory is known as on-heap memory or the JVM heap. The BlockCache implementation that manages the on-heap cache is called LruBlockCache.

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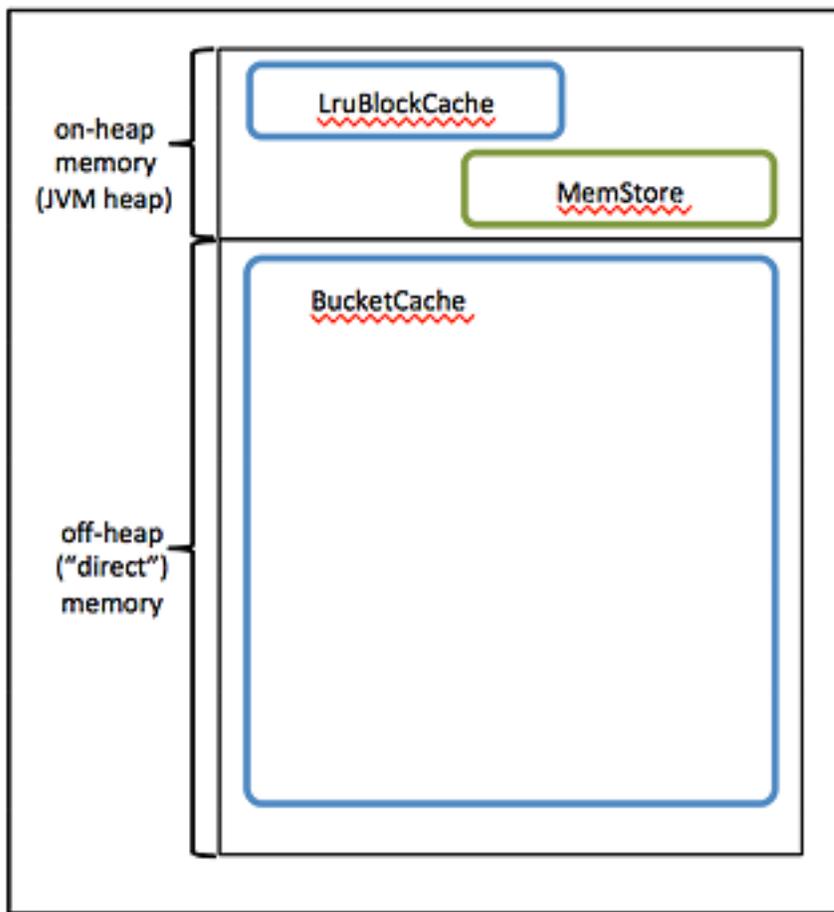
- Additional notes:
- BlockCache is enabled by default for all HBase tables.
- BlockCache is beneficial for both random and sequential read operations although it is of primary consideration for random reads.
- All regions hosted by a RegionServer share the same BlockCache.
- You can turn BlockCache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

## Configuring 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).

### Configure On-Heap BlockCache
On-Heap BlockCache is the default implementation.

### About this task

To configure either On-Heap BlockCache (LruBlockCache) or BucketCache, start by specifying the maximum amount of on-heap RAM to allocate to the HBase RegionServers on each node. The default is 1 GB, which is too small for production. You can alter the default allocation either with Ambari or in a manual installation:

**Procedure**

1.  Ambari: Set a value for the **RegionServer maximum Java heap** size.
2.  Manual Installation: Set the HBASE_HEAPSIZE environment variable in the hbase-env.sh file. Specify the value in megabytes. For example, HBASE_HEAPSIZE=20480 sets the maximum on-heap memory allocation to 20 GB in hbase-env.sh. The HBase startup script uses $HBASE_HEAPSIZE to override the default maximum JVM heap size (-Xmx).

**Results**

If you want to configure off-heap BlockCache (BucketCache) only, you are done with configuration.

**Guidelines for configuring On-Heap BlockCache (LruBlockCache)**
You need to determine the proportions of READ and WRITE operations in your workload, and use these proportions to specify on-heap memory allocation for BlockCache and MemStore.

The sum of the on-heap memory allocations for BlockCache and MemStore properties must be less than or equal to 0.8. The following table describes these two properties:

| Property | Default Value | Description |
|---|---|---|
| hfile.block.cache.size | 0.4 | Proportion of maximum JVM heap size (Java - Xmx setting) to allocate to BlockCache. A value of 0.4 allocates 40% of the maximum heap size. |
| hbase.regionserver.global.memstore.upperLimit | 0.4 | Proportion of maximum JVM heap size (Java - Xmx setting) to allocate to MemStore. A value of 0.4 allocates 40% of the maximum heap size. |

Use the following guidelines to determine the two proportions:

*   The default configuration for each property is 0.4, which configures BlockCache for a mixed workload with roughly equal proportions of random reads and writes.
*   If the amount of avaiable RAM in the off-heap cache is less than 20 GB, your workload is probably read-heavy. In this case, do not plan to configure off-heap cache, your amount of available RAM is less than 20 GB. In this case, increase the hfile.block.cache.size property and decrease the hbase.regionserver.global.memstore.upperLimit property so that the values reflect your workload proportions. These adjustments optimize read performance.
*   If your workload is write-heavy, decrease the hfile.block.cache.size property and increase the hbase.regionserver.global.memstore.upperLimit property proportionally.
*   As noted earlier, the sum of hfile.block.cache.size and hbase.regionserver.global.memstore.upperLimit must be less than or equal to 0.8 (80%) of the maximum Java heap size specified by HBASE_HEAPSIZE (-Xmx).

    If you allocate more than 0.8 across both caches, the HBase RegionServer process returns an error and does not start.
*   Do not set hfile.block.cache.size to zero.

    At a minimum, specify a proportion that allocates enough space for HFile index blocks. To review index block sizes, use the RegionServer Web GUI for each server.

Edit the corresponding values in your hbase-site.xml files.

Here are the default definitions:

```
<property>
     <name>hfile.block.cache.size</name>
     <value>0.4</value>
     <description>Percentage of maximum heap (-Xmx setting) to allocate to
block
       cache used by HFile/StoreFile. Default of 0.4 allocates 40%.
     </description>
</property>
```

```
<property>
    <name>hbase.regionserver.global.memstore.upperLimit</name>
    <value>0.4</value>
    <description>Maximum size of all memstores in a region server before
new
      updates are blocked and flushes are forced. Defaults to 40% of heap.
    </description>
</property>
```

If you have less than 20 GB of RAM for use by HBase, you are done with the configuration process. You should restart (or perform a rolling restart on) your cluster and check log files for error messages. If you have more than 20 GB of RAM for use by HBase, consider configuring the variables and properties described next.

### Prerequisites to configure Off-Heap Memory (BucketCache)

**Note:**

Before configuring off-heap memory, complete the tasks in the previous "Configuring BlockCache" section.

To prepare for BucketCache configuration, compare the figure and table below before proceeding to the "Configuring BucketCache" steps.

### Figure 2: Diagram of Configuring BucketCache
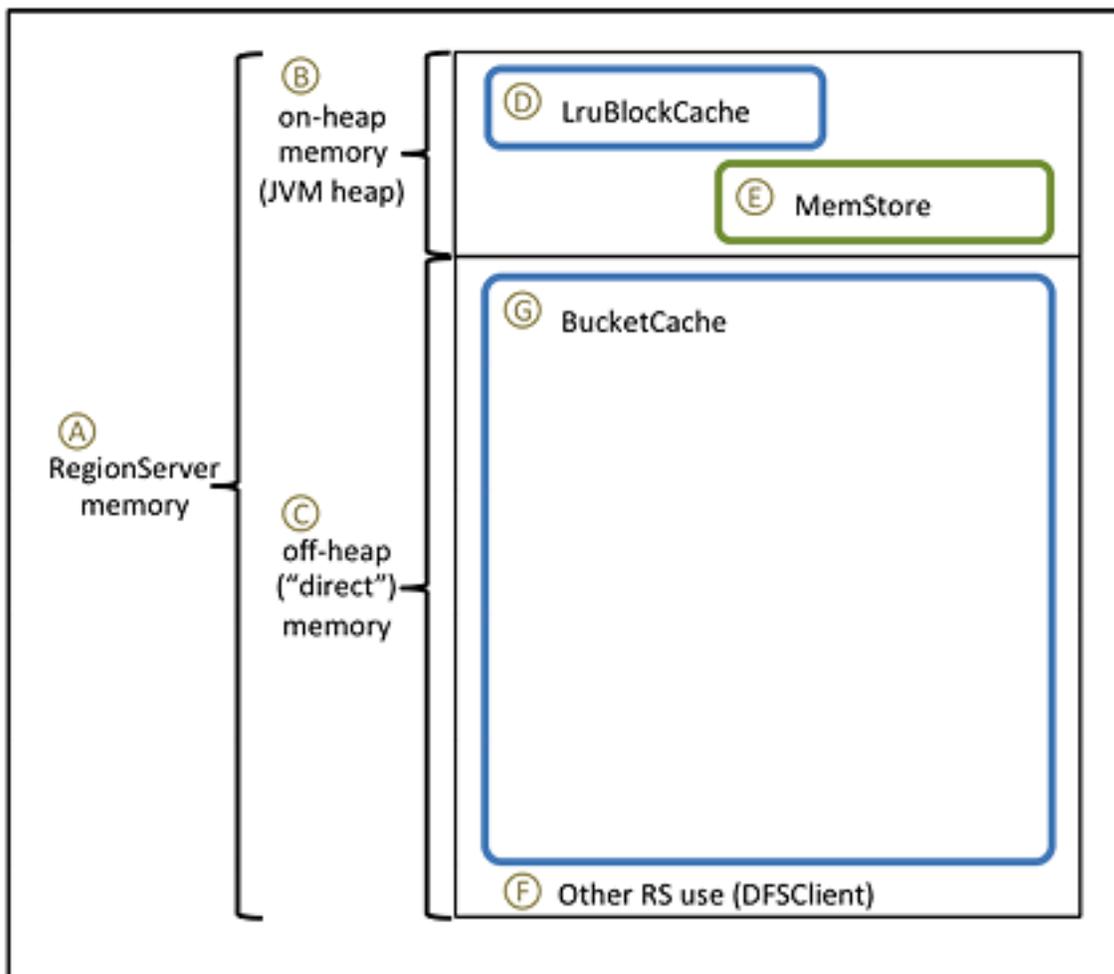


In the following table:

• The first column refers to the elements in the figure.

---

- • The second column describes each element and, if applicable, its associated variable or property name.
- • The third column contains values and formulas.
- • The fourth column computes values based on the following sample configuration parameters:

  - • 128 GB for the RegionServer process (there is additional memory available for other HDP processes)
  - • A workload of 75% reads, 25% writes
  - • HBASE_HEAPSIZE = 20 GB (20480 MB)

    **Note:**

    Most of the following values are specified in megabytes; three are proportions.

| Item | Description | Value or Formula | Example |
| --- | --- | --- | --- |
| A | Total physical memory for RegionServer operations: on-heap plus off-heap ("direct") memory (MB) | (hardware dependent) | 131072 |
| B | The HBASE_HEAPSIZE (-Xmx) property: Maximum size of JVM heap (MB)<br><br>This value was set when the BlockCache was configured. | Recommendation: 20480 | 20480 |
| C | The -XX: MaxDirectMemorySize option: Amount of off-heap ("direct") memory to allocate to HBase (MB) | A - B | 131072 - 20480 = 110592 |
| Dp | The hfile.block.cache.size property: Proportion of maximum JVM heap size (HBASE_HEAPSIZE, -Xmx) to allocate to BlockCache. The sum of this value plus the hbase.regionserver. global.memstore.size must not exceed 0.8.<br><br>This value was set when the BlockCache was configured. | (proportion of reads) * 0.8 | 0.75 * 0.8 = 0.6 |
| Dm | Maximum amount of JVM heap to allocate to BlockCache (MB) | B * Dp | 20480 * 0.6 = 12288 |
| Ep | The hbase.regionserver. global.memstore.size property: Proportion of maximum JVM heap size (HBASE_HEAPSIZE, -Xmx) to allocate to MemStore. The sum of this value plus hfile.block.cache.size must be less than or equal to 0.8. | 0.8 - Dp | 0.8 - 0.6 = 0.2 |
| F | Amount of off-heap memory to reserve for other uses (DFSClient; MB) | Recommendation: 1024 to 2048 | 2048 |
| G | Amount of off-heap memory to allocate to BucketCache (MB) | C - F | 110592 - 2048 = 108544 |
|  | The hbase.bucketcache.size property: Total amount of memory to allocate to the off-heap BucketCache (MB) | G | 108544 |

### Configure BucketCache

To configure BucketCache, you have to specify values for certain parameters in the hbase-env.sh and hbase-site.xml files.

### Procedure

1. In the hbase-env.sh file for each RegionServer, or in the hbase-env.sh file supplied to Ambari, set the -XX:MaxDirectMemorySize argument for HBASE_REGIONSERVER_OPTS to the amount of direct memory you want to allocate to HBase.

    In the sample configuration, the value would be 110592m (-XX:MaxDirectMemorySize accepts a number followed by a unit indicator; m indicates megabytes);

    HBASE_OPTS="$HBASE_OPTS -XX:MaxDirectMemorySize=110592m"

2. In the hbase-site.xml file, specify the BucketCache size.

    For the sample configuration, the values would be 120832 and 0.89830508474576, respectively. You can round up the proportion. This allocates space related to the rounding error to the (larger) off-heap memory area.

    ```
    <property>
        <name>hbase.bucketcache.size</name>
        <value>108544</value>
    </property>
    ```

3. In the hbase-site.xml file, set hbase.bucketcache.ioengine to offheap to enable BucketCache:

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

4. Restart (or perform a rolling restart on) the cluster. It can take a minute or more to allocate BucketCache, depending on how much memory you are allocating. Check logs for error messages.

### BlockCache compression

You can use BlockCache compression, when you have more data than RAM allocated to BlockCache, but your compressed data can fit into BlockCache. The savings must be worth the increased garbage collection overhead and overall CPU load.

BlockCache compression caches data and encoded data blocks in their on-disk formats, rather than decompressing and decrypting them before caching. When compression is enabled on a column family, more data can fit into the amount of memory dedicated to BlockCache. Decompression is repeated every time a block is accessed, but the increase in available cache space can have a positive impact on throughput and mean latency.

If your data can fit into block cache without compression, or if your workload is sensitive to extra CPU or garbage collection overhead, we recommend against enabling BlockCache compression.

Block cache compression is disabled by default.

> **Note:**
>
> Before you can use BlockCache compression on an HBase table, compression must be enabled for the table. For more information, see Enable Compression on a ColumnFamily on the Apache website.

### Enable BlockCache compression

You can enable BlockCache compression by setting an appropriate value for the cachecompressed property in the hbase-site configuration file.

**Procedure**

1. Set the hbase.block.data.cachecompressed to true in the hbase-site.xml file on each RegionServer.

2. Restart or perform a rolling restart of your cluster.

3. Check logs for error messages.

**Related Information**

Enable Compression on a ColumnFamily

**BlockCache-and-MemStore-Properties**

You can edit the hfile.block.cache.size and hbase.regionserver.global.memstore.upperLimit properties in the hbase-site.xml configuration file.

| Property | Default Value | Description |
| --- | --- | --- |
| hfile.block.cache.size | 0.4 | Proportion of maximum JVM heap size (Java -Xmx setting) to allocate to BlockCache. A value of 0.4 allocates 40% of the maximum heap size. |
| hbase.regionserver.global.memstore.upperLimit | 0.4 | Proportion of maximum JVM heap size (Java -Xmx setting) to allocate to MemStore. A value of 0.4 allocates 40% of the maximum heap size. |

# Import data into HBase with Bulk load

You can importa data with a bulk load operation to bypass the HBase API and writes content, properly formatted as HBase data files (HFiles), directly to the file system. It uses fewer CPU and network resources than using the HBase API for similar work.

**About this task**

The following recommended bulk load procedure uses Apache HCatalog and Apache Pig.

**Procedure**

1. Prepare the input file, as shown in the following data.tsv example input file:

```
row1 c1 c2
row2 c1 c2
row3 c1 c2
row4 c1 c2
row5 c1 c2
row6 c1 c2
row7 c1 c2
row8 c1 c2
row9 c1 c2
row10 c1 c2
```

2. Make the data available on the cluster, as shown in this continuation of the example:

```
hadoop fs -put data.tsv /tmp/
```

3. Define the HBase schema for the data, shown here as creating a script file called simple.ddl, which contains the HBase schema for data.tsv:

```
CREATE TABLE simple_hcat_load_table (id STRING, c1 STRING, c2 STRING)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ( 'hbase.columns.mapping' = 'd:c1,d:c2' )
TBLPROPERTIES ( 'hbase.table.name' = 'simple_hcat_load_table'
);
```

**4.** Create and register the HBase table in HCatalog:

```
hcat -f simple.ddl
```

**5.** Create the import file.

The following example instructs Pig to load data from data.tsv and store it in simple_hcat_load_table. For the purposes of this example, assume that you have saved the following statement in a file named simple.bulkload.pig.

```
A = LOAD 'hdfs:///tmp/data.tsv' USING PigStorage('\t') AS (id:chararray,
 c1:chararray,
c2:chararray);
-- DUMP A;
STORE A INTO 'simple_hcat_load_table' USING
 org.apache.hive.hcatalog.pig.HCatStorer();
```

> **Note:**
>
> Modify the filenames and table schema for your environment.

**6.** Execute the following command on your HBase server machine. The command directs Pig to populate the HBase table by using HCatalog bulkload.

```
pig -useHCatalog simple.bulkload.pig
```

# Using Snapshots in HBase

HBase snapshot support enables you to take a snapshot of a table without much impact on RegionServers, because snapshot, clone, and restore operations do not involve data copying. In addition, exporting a snapshot to another cluster has no impact on RegionServers.

Prior to HBase 0.94.6, the only way to back up or clone a table was to use the CopyTable or ExportTable utility, or to copy all of the HFiles in HDFS after disabling the table. The disadvantage of these methods is that using the first might degrade RegionServer performance, and using the second requires you to disable the table, which means no reads or writes can occur.

## Configure a Snapshot

You can configure Snapshot by setting the hbase.snapshot.enabled property in HBase 0.94.6 up to HBase 0.95. Snapshots are enabled by default starting with HBase 0.95 and above versions.

### Procedure

• Set the hbase.snapshot.enabled property to true

```
<property>
    <name>hbase.snapshot.enabled</name>
    <value>true</value>
</property>
```

## Take a Snapshot

You can take a Snapshot of the specified table in the HBase shell.

### Procedure

- Start the HBase shell and clone the required table.

```
$ hbase shell
hbase> snapshot 'myTable', 'myTableSnapshot-122112'
```

## List Snapshots

You can list and describe all Snapshots using list_snapshots command.

### Procedure

- In the hbase shell, enter list_snapshots command.

```
$ hbase shell
hbase> list_snapshots
```

## Delete Snapshots

You can remove Snapshots using delete_snapshot command.

### About this task

When you remove the Snapshot, the files associated with it will be removed if they are no longer needed.

### Procedure

- In the hbase shell, enter delete_snapshot command and the name of the Snapshot.

```
$ hbase shell
hbase> delete_snapshot 'myTableSnapshot-122112'
```

## Clone a table from a Snapshot

Clone operation enables you to create a new table with the same data as the original from the specified Snapshot.

### About this task
The clone operation does not involve data copies. A change to the cloned table does not impact the snapshot or the original table.

### Procedure

- In the HBase shell, enterclone_snapshot command and specify the name of the Snapshot.

```
$ hbase shell
hbase> clone_snapshot 'myTableSnapshot-122112', 'myNewTestTable'
```

## Restore a Snapshot

The restore operation brings back the table to its original state when the snapshot was taken, changing both data and schema, if required.

**About this task**

The restore operation requires the table to be disabled.

> **Note:**
>
> Because replication works at the log level and snapshots work at the file system level, after a restore, the replicas will be in a different state than the master. If you want to use restore, you need to stop replication and redo the bootstrap.
>
> In case of partial data loss due to client issues, you can clone the table from the snapshot and use a MapReduce job to copy the data that you need from the clone to the main one (instead of performing a full restore, which requires the table to be disabled).

**Procedure**

- In the HBase shell, first disable the table.
- Enterrestore_snapshot command and specify the name of the table.

```
$ hbase shell
hbase> disable 'myTable'
hbase> restore_snapshot 'myTableSnapshot-122112'
```

## Snapshot Operations and ACLs

If you are only a global administrator, you can take, clone, or restore a snapshot when using security with the AccessController coprocessor.

When you take, clone, or restore a snapshot, ACL rights are not captured. Restoring the Snapshot of a table preserves the ACL rights of the existing table, while cloning a table creates a new table that has no ACL rights until you add them.

## Export data to another cluster

The ExportSnapshot tool copies all the data related to a snapshot (HFiles, logs, and snapshot metadata) to another cluster. The tool executes a MapReduce job, similar to distcp, to copy files between the two clusters. Because it works at the file system level, the HBase cluster does not have to be online.

The HBase ExportSnapshot tool must be run as user hbase. The HBase ExportSnapshot tool uses the temp directory specified by hbase.tmp.dir (for example, /grid/0/var/log/hbase), created on HDFS with user hbase as the owner.

For example, to copy a snapshot called MySnapshot to an HBase cluster srv2 (hdfs://srv2:8020/hbase) using 16 mappers, input the following:

```
$ hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot MySnapshot
 -copy-to
hdfs://yourserver:8020/hbase_root_dir -mappers 16
```

# Backing up and restoring Apache HBase datasets

Backup-and-restore is a standard set of operations for many databases. An effective backup-and-restore strategy helps ensure that you can recover data in case of data loss or failures. The HBase backup-and-restore utility helps ensure that enterprises using HBase as a data repository can recover from these types of incidents. Another important feature of the backup-and-restore utility is the ability to restore the database to a particular point-in-time, commonly referred to as a snapshot.

The HBase backup-and-restore utility features both full backups and incremental backups. A full backup is required at least once. The full backup is the foundation on which incremental backups are applied to build iterative snapshots. Incremental backups can be run on a schedule to capture changes over time, for example by using a Cron job.

Incremental backup is more cost effective because it only captures the changes. It also enables you to restore the database to any incremental backup version. Furthermore,the utilities also enable table-level data backup-and-recovery if you do not want to restore the entire dataset of the backup.

**Related Information**
Hortonworks Support Portal

# Planning a backup-and-restore Strategy for your environment

There are a few strategies that you can use to implement backup-and-restore in your environment. They are Backup within a cluster, Backup to the dedicated HDFS archive cluster and Backup to the cloud or a storage vendor.

There are a few strategies you can use to implement backup-and-restore in your environment. The following sections show how they are implemented and identify potential tradeoffs.

> **Note:**
>
> HBase backup-and restore tools are currently not supported on Transparent Data Encryption (TDE)-enabled HDFS clusters. This is related to the Apache HBASE-16178 known issue.

**Related Information**
Apache HBASE-16178

## Backup within a Cluster

Backup-and-restore within the same cluster is only appropriate for testing.

This strategy is not suitable for production unless the underlying HDFS layer is backed up and is reliably recoverable.

**Figure 3: Intracluster Backup**
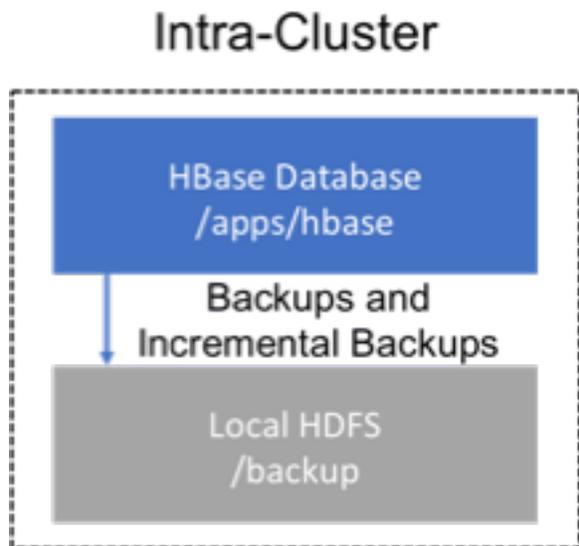


## Backup to the dedicated HDFS archive cluster

This strategy provides greater fault tolerance and provides a path towards disaster recovery.

In this setting, you will store the backup on a separate HDFS cluster by supplying the backup destination cluster's HDFS URL to the backup utility. You should consider backing up to a different physical location, such as a different data center.

Typically, a backup-dedicated HDFS cluster uses a more economical hardware profile.

**Figure 4: Backup-Dedicated HDFS Cluster**

## HDFS Archive Cluster

### HBase Online Cluster

HBase Database
/apps/hbase

Backups and
Incremental Backups
hdfs://archive:8020/backup

### Archive Cluster

HDFS
/backup
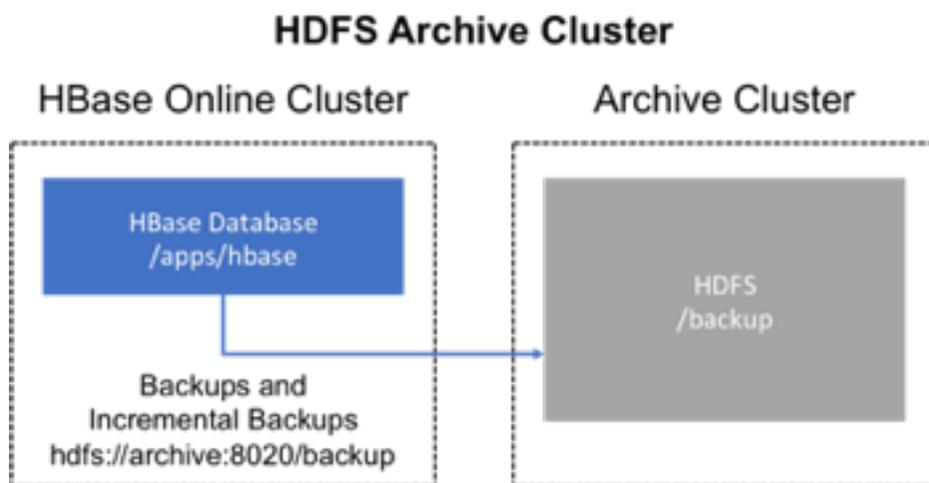
## Backup to the Cloud or a Storage vendor

This approach enables you to safeguard the HBase incremental backups by storing the data on provisioned, secure servers that belong to third-party vendors, which are located off-site.

The vendor can be a public cloud provider or a storage vendor who uses a Hadoop-compatible file system, such as S3 and other HDFS-compatible destinations.

**Figure 5: Backup to Vendor Storage Solutions**

## S3 or HDFS-Compatible

### HBase Online Cluster

HBase Database
/apps/hbase

### S3 or Appliance

Long-Term Storage
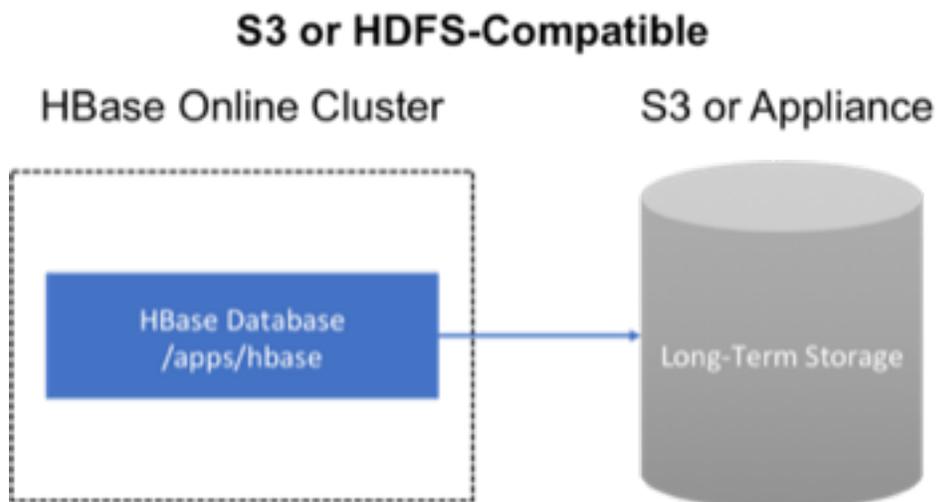
**Note:**

The HBase backup utility does not support backup to multiple destinations. A workaround is to manually create copies of the backed up files from HDFS or S3.

# Best practices for backup-and-restore

To perform a successful backup-and-restore operation, you need to formulate a restore strategy and test it, store backup data from a production cluster on a different cluster or a server, secure a full backup image first, define and use backup sets for groups of tables and finally document the backup-and-restore strategy.

**Procedure**

- Formulate a restore strategy and test it. Before you rely on a backup-and-restore strategy for your production environment, identify how backups must be performed, and more importantly, how restores must be performed. Test the plan to ensure that it is workable.
- At a minimum, store backup data from a production cluster on a different cluster or server. To further safeguard the data, use a backup location that is at a different site. If you have a unrecoverable loss of data on your primary production cluster as a result of computer system issues, you may be able to restore the data from a different cluster or server at the same site. However, a disaster that destroys the whole site renders locally stored backups useless. Consider storing the backup data and necessary resources (both computing capacity and operator expertise) to restore the data at a site sufficiently remote from the production site. In the case of a catastrophe at the whole primary site (fire, earthquake, etc.), the remote backup site can be very valuable.
- Secure a full backup image first. As a baseline, you must complete a full backup of HBase data at least once before you can rely on incremental backups. The full backup should be stored outside of the source cluster. To ensure complete dataset recovery, you must run the restore utility with the option to restore baseline full backup. The full backup is the foundation of your dataset. Incremental backup data is applied on top of the full backup during the restore operation to return you to the point in time when backup was last taken.
- Define and use backup sets for groups of tables that are logical subsets of the entire dataset. You can group tables into an object called a backup set. A backup set can save time when you have a particular group of tables that you expect to repeatedly back up or restore. When you create a backup set, you type table names to include in the group. The backup set includes not only groups of related tables, but also retains the HBase backup metadata. Afterwards, you can invoke the backup set name to indicate what tables apply to the command execution instead of entering all the table names individually.
- Document the backup-and-restore strategy, and ideally log information about each backup.  Document the whole process so that the knowledge base can transfer to new administrators after employee turnover. As an extra safety precaution, also log the calendar date, time, and other relevant details about the data of each backup. This metadata can potentially help locate a particular dataset in case of source cluster failure or primary site disaster. Maintain duplicate copies of all documentation: one copy at the production cluster site and another at the backup location or wherever it can be accessed by an administrator remotely from the production cluster.

# Running the backup-and-restore utility

To run the backup-and-restore-utility tool, you can use the commands and the arguments of the operations such as create and maintain a complete backup image, monitor backup progress and restore a backup image.

> **Note:**
>
> For Non-Ambari (Manual) Installations of HDP and HBase: You must modify the container-executor.cfg configuration file to include the allowed.system.users=hbase property setting. No spaces are allowed in entries of the container-executor.cfg file. Ambari-assisted installations of HDP automatically set the property in the configuration file.

**Example of a valid configuration file for backup-and-restore:**

```
yarn.nodemanager.log-dirs=/var/log/hadoop/mapred
yarn.nodemanager.linux-container-executor.group=yarn
banned.users=hdfs,yarn,mapred,bin
allowed.system.users=hbase
min.user.id=500
```

> **Note:**
>
> Enter hbase backup help command in your HBase command-line interface to access the online help that provides basic information about a command and its options.

## Create and maintain a complete backup image

The first step in running the backup-and-restore utilities is to perform a full backup and to store the data in a separate image from the source. At a minimum, you must do this to get a baseline before you can rely on incremental backups.

### About this task

**Note:**

For sites using Apache Phoenix: Include the SQL system catalog tables in the backup. In the event that you need to restore the HBase backup, access to the system catalog tables enable you to resume Phoenix interoperability with the restored data.

### Procedure

• Run hbase backup create command as hbase superuser to create a complete backup image.

hbase backup create full hdfs://host5:8020/data/backup -t SALES2,SALES3 -w 3

This command creates a full backup image of two tables, SALES2 and SALES3, in the HDFS instance, whose NameNode is //host5:8020/ in the path data/backup. The -w option specifies that no more than three parallel workers complete the operation.

After the command finishes running, the console prints a SUCCESS or FAILURE status message. The SUCCESS message includes a backup ID. The backup ID is the Unix time (also known as Epoch time) that the HBase master received the backup request from the client.

**Note:**

Record the backup ID that appears at the end of a successful backup. In case the source cluster fails and you need to recover the dataset with a restore operation, having the backup ID readily available can save time.

## Command for creating HBase backup image

Use hbase backup create command as hbase superuser to create a complete backup image.

Ensure that backup is enabled on the cluster. To enable backup, add the following properties to hbase-site.xml and restart the HBase cluster.

```
<property>
<name>hbase.backup.enable</name>
<value>true</value>
</property>
<property>
<name>hbase.master.logcleaner.plugins</name>
<value>YOUR_PLUGINS,org.apache.hadoop.hbase.backup.master.BackupLogCleaner
</value>
</property>
<property>
<name>hbase.procedure.master.classes</name>
<value>YOUR_CLASSES,org.apache.hadoop.hbase.backup.master.
LogRollMasterProcedureManager</value>
</property>
<property>
<name>hbase.procedure.regionserver.classes</name>
<value>YOUR_CLASSES,org.apache.hadoop.hbase.backup.regionserver.
LogRollRegionServerProcedureManager</value>
</property>
<property>
<name>hbase.coprocessor.region.classes</name>
<value>YOUR_CLASSES,org.apache.hadoop.hbase.backup.BackupObserver</value>
</property>
```

Following is the usage of the hbase backup create command with its arguments:

```
hbase backup create <type> <backup_path> [options]
```

Required command-line arguments

| | |
|---|---|
| **type** | It specifies the type of backup to execute, which can be full or incremental. |
| | Using the full argument creates a full backup image. Using the incrementalargument creates an incremental backup image. It requires a full backup to already exist. |
| **backup_path** | The backup_path argument specifies the full root path of where to store the backup image. Valid prefixes are hdfs:, webhdfs:, gpfs:, and s3fs:. |

Optional command-line arguments

| | |
|---|---|
| **-b <arg>bandwidth_per_task** | Specifies the bandwidth of each MapReduce task in MB per second. |
| **-d <arg>** | Enables DEBUG mode, which prints additional logging about the backup creation. |
| **-q <arg>** | It allows you to specify the Yarn queue name to run the backup create command on. |
| **-s <arg>** | Identify the tables to backup based on a backup set. Refer "Using Backup Sets" for the purpose and usage of backup sets. It is mutually exclusive with the -t (table list) option. |
| **-t <arg>** | A comma-separated list of tables to back up. If no tables are specified, all tables are backed up. No regular-expression or wildcard support is present; all table names must be explicitly listed. It is mutually exclusive with the -s option. One of these named options are required. |
| **-w <arg>** | Specifies the number of parallel MapReduce tasks to execute. |

## Monitor backup progress

You can monitor a running backup by running the hbase backup progress command and specifying the backup ID as an argument.

### Procedure

* Run hbase backup progress command as hbase superuser to view the progress of a backup.

  Specify the backup id that you want to monitor by viewing the progress information.

  ```
  hbase backup progress backupId_1467823988425
  ```

### Example

Command for monitoring running backup progress

Use hbase backup progresscommand as hbase superuser to view the progress of a backup.

```
hbase backup progress
backupId
```

Required command-line arguments

**backupId**                                                          Specifies the backup that you want to monitor by seeing
                                                                       the progress information. The backup ID argument is
                                                                       case-sensitive.

## Using backup sets

Backup sets can ease the administration of HBase data backups and restores by reducing the amount of repetitive
input of table names.

You can group tables into a named backup set with the hbase backup set add command. You can then use the -
set option to invoke the name of a backup set in the hbase backup create or hbase backup restore rather than list
individually every table in the group. You can have multiple backup sets.

> **Note:**
>
> Note the differentiation between the hbase backup set add command and the -set option. The hbase backup set
> add command must be run before using the -set option in a different command because backup sets must be
> named and defined before using backup sets as shortcuts.

If you run the hbase backup set add command and specify a backup set name that does not yet exist on your system,
a new set is created. If you run the command with the name of an existing backup set name, then the tables that you
specify are added to the set.

In the command, the backup set name is case-sensitive.

> **Note:**
>
> The metadata of backup sets are stored within HBase. If you do not have access to the original HBase cluster
> with the backup set metadata, then you must specify individual table names to restore the data.

### Backup set subcommands

To create a backup set, run the following command as hbase superuser.

```
hbase backup set COMMAND [name] [tables]
```

COMMAND is one of the following:

```
add
remove
list
describe
delete
```

```
backup_set_name
tables
```

Following list details subcommands of the hbase backup set command.

**Note:**

You must enter one (and no more than one) of the following subcommands after hbase backup set to complete an operation. Also, the backup set name is case-sensitive in the command-line utility.

**add**

Use this subcommand to add tables to a backup set. Specify a backup_set_name value after this argument to create a backup set.

Example of backup set addExample

**hbase backup set add Q1Data TEAM_3,TEAM_4**

Depending on the environment, this command results in one of the following actions:

- If the Q1Data backup set does not exist, a backup set containing tables TEAM_3 and TEAM_4 is created.
- If the Q1Data backup set exists already, the tables TEAM_3 and TEAM_4 are added to the Q1Data backup set.

**remove**

Use this subcommand to remove tables from a set. Specify the tables to remove in the tables argument.

**list**

Use this subcommand to list all backup sets.

**describe**

Use this subcommand to display on the screen a description of a backup set. The information includes whether the set has full or incremental backups, start and end times of the backups, and a list of the tables in the set. This subcommand must precede a valid value for the backup_set_name value.

**delete**

Use this subcommand to delete a backup set. Enter the value for the backup_set_name option directly after the hbase backup set delete command.

Optional command-line arguments

**backup_set_name**

Use this argument to assign or invoke a backup set name. The backup set name must contain only printable characters and cannot have any spaces.

**tables**

List of tables (or a single table) to include in the backup set. Enter the table names as a comma-separated list. If no tables are specified, all tables are included in the set.

**Note:**

Maintain a log or other record of the case-sensitive backup set names and the corresponding tables in each set on a separate or remote cluster, mirroring your backup strategy. This information can help you in case of failure on the primary cluster.

## Restore a backup image

You can perform restore operation to restore a backup image. You can only restore on a live HBase cluster because the data must be redistributed to the RegionServers to complete the restore operation successfully.

- Run the hbase restore command as hbase superuser to restore a backup image.

hbase restore /tmp/backup_incremental backupId_1467823988425 -t mytable1,mytable2

This command restores two tables of an incremental backup image. In this example:

- /tmp/backup_incremental is the path to the directory containing the backup image.
- backupId_1467823988425 is the backup ID.
- mytable1 and mytable2 are the names of the tables in the backup image to be restored.

## Command for restoring backup image

Use hbase restore command as hbase superuser to restore a backup image.

```
hbase restore <backup_path> <backup_id> [options]
```

Required command-line arguments

| | |
|---|---|
| **backup_path** | It specifies the path to a backup destination root, which is the full filesystem URI of where to store the backup image. Valid prefixes are hdfs:, webhdfs:, gpfs:, and s3fs:. |
| **backupId** | The backup ID that uniquely identifies the backup image to be restored. |

Optional command-line arguments

| | |
|---|---|
| **-c** | It checks the restore sequence and dependencies only, however it is not executed. It performs a dry-run of the restore. |
| **-d** | It enables debug loggings. |
| **-m <arg>** | A comma-separated list of target tables to restore into. |
| **-o** | Overwrite the data, if any of the restore target tables exists. |
| **-q <arg>** | It allows you to specify the Yarn queue name to run the backup restore command on. |
| **-t <arg>** | A comma-separated list of tables to restore. The values for this argument must be entered directly after the backupId argument. |
| **-s <arg>** | It specifies Backup set to restore. It is mutually exclusive with the -t (table list) option. |
| **-h,--help** | It shows the usage of the restore command. |

## Administering and deleting backup images

The hbase backup command has several subcommands that help you to administer backup images as they accumulate.

Most production environments require recurring backups, so it is necessary to have utilities to help manage the data of the backup repository. Some subcommands enable you to find information that can help identify backups that are relevant in a search for particular data. You can also delete backup images.

### HBase backup commands

You can use an appropriate hbase backup COMMAND to administer the hbase backups.

The following list details each hbase backup COMMAND [command-specific arguments]. Run the full command line as hbase superuser.

| | |
|---|---|
| **hbase backup create [ &lt;type&gt;&lt;backup_path&gt; [options]]** | Creates a new backup image. |
| **hbase backup delete [&lt;backup_id&gt;]** | Deletes the specified existing backup image from the system. The backup_id option is required. |
| **hbase backup describe [&lt;backup_id&gt;]** | Shows the detailed information of a backup image. The backup_id option is required. |
| **hbase backup history [options]** | Options:<br><br>-n  Number of records of backup history. Default is 10.<br><br>-p Backup destination root directory path.<br><br>-s  Backup set name<br><br>-t Table name. If specified, only backup images, which contain this table will be listed.<br><br>Shows the history of all successful backups. |
| **hbase backup progress [&lt;backup_id&gt;]** | Shows the progress of the latest backup request. Backup imageid is (optional). If no ID ia specified, the command will show the progress of the currently running backup session. |
| **hbase backup set COMMAND [[name] [tables]]** | Manages the backup sets. You can add, remove, delete, describe, and delete a backup set using this command. |
| **hbase backup progress [&lt;backup_id&gt;]** | Shows the progress of the latest backup request. Backup imageid is (optional). If no ID ia specified, the command will show the progress of the currently running backup session. |
| **hbase backup repair** | It repairs the backup system table.<br><br>It repairs the backup system table. This command attempts to correct any inconsistencies in persisted backup metadata, which exists as a result of software errors or unhandled failure scenarios. While the backup implementation tries to correct all errors on its own, this tool may be necessary in the cases where the system cannot automatically recover on its own. |
| **hbase backup merge [&lt;backup_id&gt;]** | Merges the backup images. Run merge command to merge incremental backup images into a single incremental backup image. The most recent backup |

image will be overwritten by resulting merged image, all other images will be deleted.

## Incremental backup-and-restore

HBase incremental backups enable more efficient capture of HBase table images than previous attempts at serial backup-and-restore solutions, such as those that only used HBase Export and Import APIs. Incremental backups use Write Ahead Logs (WALs) to capture the data changes since the previous backup was created. A roll log is executed across all RegionServers to track the WALs that need to be in the backup.

After the incremental backup image is created, the source backup files usually are on same node as the data source. A process similar to the DistCp (distributed copy) tool is used to move the source backup files to the target filesystems. When a table restore operation starts, a two-step process is initiated. First, the full backup is restored from the full backup image. Second, all WAL files from incremental backups between the last full backup and the incremental backup being restored are converted to HFiles, which the HBase Bulk Load utility automatically imports as restored data in the table.

You can only restore on a live HBase cluster because the data must be redistributed to complete the restore operation successfully.

## Example scenario: Safeguarding application datasets on Amazon S3

This scenario describes how a hypothetical retail business uses backups to safeguard application data and then restore the dataset after failure.

### About this task

The HBase administration team uses backup sets to store data from a group of tables that have interrelated information for an application called green. In this example, one table contains transaction records and the other contains customer details. The two tables need to be backed up and be recoverable as a group.

The admin team also wants to ensure daily backups occur automatically.

**Figure 6: Tables Composing the Backup Set**
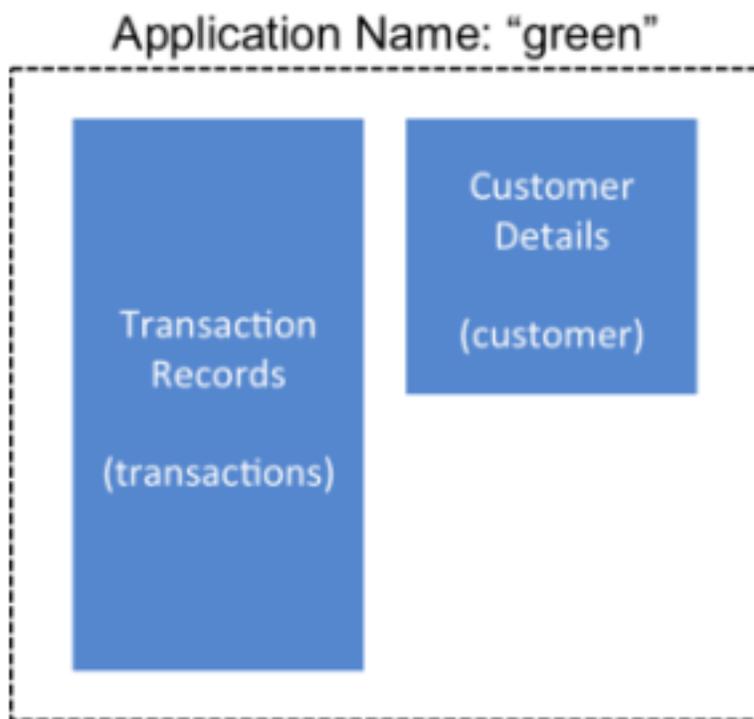
The following is an outline of the steps and examples of commands that are used to backup the data for the green application and to recover the data later. All commands are run when logged in as hbase superuser.

**Procedure**

1. A backup set called green_set is created as an alias for both the transactions table and the customer table. The backup set can be used for all operations to avoid typing each table name. The backup set name is case-sensitive and should be formed with only printable characters and without spaces.

```
$ hbase backup set add green_set transactions
$ hbase backup set add green_set customer
```

2. The first backup of green_set data must be a full backup. The following command example shows how credentials are passed to Amazon S3 and specifies the file system with the s3a: prefix.

```
hbase -D hadoop.security.credential.provider.path=jceks://
hdfs@prodhbasebackups/hbase/hbase/s3.jceks backup create full
s3a://green-hbase-backups/ -set green_set
```

3. Incremental backups should be run according to a schedule that ensures essential data recovery in the event of a catastrophe. At this retail company, the HBase admin team decides that automated daily backups secures the data sufficiently. The team decides that they can implement this by modifying an existing Cron job that is defined in / etc/crontab. Consequently, IT modifies the Cron job by adding the following line:

```
hbase -D hadoop.security.credential.provider.path=jceks://
hdfs@prodhbasebackups/hbase/daily/s3.jceks backup create incremental
s3a://green-hbase-backups/ -set green_set
```

4. A catastrophic IT incident disables the production cluster that the green application uses. An HBase system administrator of the backup cluster must restore the green_set dataset to the point in time closest to the recovery objective.

   **Note:**

   If the administrator of the backup HBase cluster has the backup ID with relevant details in accessible records, the following search with the hdfs dfs -ls command and manually scanning the backup ID list can be bypassed. Consider continuously maintaining and protecting a detailed log of backup IDs outside the production cluster in your environment.

   The HBase administrator runs the following command on the directory where backups are stored to print a list of successful backup IDs on the console:

```
hdfs dfs -ls -t s3a://green-hbase-backups/
```

5. The admin scans the list to see which backup was created at a date and time closest to the recovery objective. To do this, the admin converts the calendar timestamp of the recovery point in time to Unix time because backup IDs are uniquely identified with Unix time. The backup IDs are listed in reverse chronological order, meaning the most recent successful backup appears first.

   The admin notices that the following line in the command output corresponds with the green_set backup that needs to be restored:

```
s3a://green-hbase-backups//backupId_1467823988425
```

6. The admin restores green_set invoking the backup ID and the -overwrite option. The -overwrite option truncates all existing data in the destination and populates the tables with data from the backup dataset. Without this flag, the backup data is appended to the existing data in the destination. In this case, the admin decides to overwrite the data because it is corrupted.

```
hbase restore -D hadoop.security.credential.provider.path=jceks://
hdfs@prodhbasebackups/hbase/daily/s3.jceks restore -set green_set s3a://
green-hbase-backups/backupId_1467823988425 -overwrite
```

# Medium Object (MOB) storage support in Apache HBase

An HBase table becomes less efficient once any cell in the table exceeds 100 KB of data. Objects exceeding 100 KB are common when you store images and large documents, such as email attachments, in HBase tables. But, you can configure Hortonworks Data Platform (HDP) HBase to support tables with cells that have medium-size objects, also known as medium objects or more commonly as MOBs, to minimize the performance impact that objects over 100 KB can cause.

MOB support operates by storing a reference of the object data within the main table. The reference in the table points toward external HFiles that contain the actual data, which can be on disk or in HDFS.

To enable MOB storage support for a table column family, you can choose one of two methods. One way is to run the table create command or the table alter command with MOB options in the HBase shell. Alternatively, you can set MOB parameters in a Java API.

# Methods to enable MOB storage support

You can enable MOB storage support and configure the MOB threshold by using one of two different methods such as: configure options in the command line and invoke support parameters in a Java API.

If you do not specify a MOB size threshold, the default value of 100 KB is used.

> **Note:**
>
> While HBase enforces no maximum-size limit for a MOB column, generally the best practice for optimal performance is to limit the data size of each cell to 10 MB.

Prerequisites:

• hbase superuser privileges
• HFile version 3, which is the default format of HBase 0.98+.

# Method 1:Enable MOB Storage support using configure options in the command line

You can enable MOB storage support using configure options in the HBase shell.

**Procedure**

• Run the table create command or the table alter command and do the following.

    • Set the IS_MOB option to true.
    • Set the MOB_THRESHOLD option to the number of bytes for the threshold size above which an object is treated as a medium-size object.

Following are a couple of HBase shell command examples:

```
hbase> create 't1', {NAME => 'IMAGE_DATA', IS_MOB => true, MOB_THRESHOLD
 => 102400}
```

```
hbase> alter 't1', {NAME => 'IMAGE_DATA', IS_MOB => true, MOB_THRESHOLD =>
 102400}
```

# Method 2: Invoke MOB support parameters in a Java API

You can use the MOB support parameters in a Java API to enable and configure MOB storage support.

### About this task

If you invoke the MOB threshold parameter, substitute bytes with the value for the number of bytes for the threshold size at which an object is treated as a medium-size object. If you omit the parameter when you enable MOB storage, the threshold value defaults to 102400 (100 KB).

### Procedure

* Use the hcd.setMobEnabled(true); parameter to enable and configure MOB storage support. The parameter hcd.setMobThreshold(bytes); is optional.

  Following is a Java API example:

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

# Test the MOB storage support configuration

You can use the java utility org.apache.hadoop.hbase.IntegrationTestIngestWithMOBto help with testing the MOB feature.

### Procedure

* Run the org.apache.hadoop.hbase.IntegrationTestIngestWithMOB utility to the test the MOB storage configuration. Values in the command options are expressed in bytes.

  Following is an example that uses default values (in bytes):

  ```
  $ sudo -u hbase hbase org.apache.hadoop.hbase.IntegrationTestIngestWithMOB
   \
  -threshold 1024 \
  -minMobDataSize 512 \
  -maxMobDataSize threshold * 5 \
  ```

# MOB storage cache properties

Opening a MOB file places 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 might need to tune the MOB file reader cache to an appropriate size so that HBase scales appropriately.

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 can use either one of the two following methods.

**Related reference**
MOB cache properties

## Method 1: Enter property settings using Ambari

Using Ambari, you can specify the MOB property settings.

### Procedure

1. In Ambari select Advanced tab > Custom HBase-Site > Add Property.
2. Enter a MOB cache property in the **Type** field.
3. Complete the **Value** field with your configuration setting.

## Method 2: Enter property settings directly in the hbase-site.xml file

You can specify the MOB property settings directly in hbase-site.xml file.

### Procedure

1. Open the RegionServer's hbase-site.xml file. The file is usually located under /etc/hbase/conf .
2. Add the MOB cache properties to the RegionServer's hbase-site.xml file.
3. Adjust the parameters or use the default settings.
4. Initiate a restart or rolling restart of the RegionServer. For more information about rolling restarts, see the Rolling Restart section of the online Apache HBase Reference Guide.

**Related Information**
Rolling Restart

## MOB cache properties

The MOB cache properties are hbase.mob.file.cache.size, hbase.mob.cache.evict.period, and hbase.mob.cache.evict.remain.ratio.

**Table 1: MOB Cache Properties**

| Property and Default Value | Description |
|---|---|
| hbase.mob.file.cache.size<br><br>Default Value: 1000 | Number of opened file handlers to cache. A larger value enhances read operations by providing more file handlers per MOB file cache and reduce frequent file opening and closing. However, if the value is set too high, a "too many opened file handers" condition can occur. |

| Property and Default Value | Description |
|---|---|
| hbase.mob.cache.evict.period<br><br>Default Value: 3600 | The amount of time (in seconds) after which an unused file is evicted from the MOB cache. |
| hbase.mob.cache.evict.remain.ratio<br><br>Default Value: 0.5f | A multiplier (between 0.0 and 1.0) that determines how many files remain cached after the hbase.mob.file.cache.size property threshold is reached. The default value is 0.5f, which indicates that half the files (the least-recently used ones) are evicted. |

# HBase 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.

As of version 2.6, HDP has an additional quota type: a filesystem space quota. You can use file-system quotas to regulate the usage of filesystem space on namespaces or at the table level.

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 |
| Numbr of regions | Metadata | Limit number of regions for each namespace | Yes | No | No |

## Setting up 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 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:<br>• sec (second)<br>min (minute)<br>hour<br>day |
| **Request sizes and space limit** | Can be expressed in the following units:<br>• B: bytes<br>K: kilobytes<br>M: megabytes<br>G: gigabytes<br>P: petabytes<br>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<br>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'
```

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',
 LIMIT => '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',
 LIMIT => '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',
 LIMIT => '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',
 LIMIT => 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_INSERTS
```

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 =>
 DISABLE
```

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 coherent 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

## Live Write Access

As one would expect, every violation policy outlined disables the ability to write new data into the system. This means that any Mutation implementation other than DELETE operations could be rejected by the system. Depending on the violation policy, DELETE operations still might be accepted.

## Bulk Write Access

Bulk loading HFiles can be an extremely effective way to increase the overall throughput of ingest into HBase. Quota management is very relevant 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. A goal of applying these HBase quotas is to ensure that HDFS remains healthy and sustains a higher level of availability to HBase users. Guaranteeing that there is always free space in 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

Quotas should ideally be listed on the HBase Master UI. The list of defined quotas should be present as well as those quotas whose violation policy is being enforced. The list of tables/namespaces with enforced violation policies should also be presented via 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 n1t3exceeds 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.maxtables'=>'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.quota.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.maxregions'=>'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.quota.maxregions'}
```

# Understanding Apache HBase Hive integration

With Hortonworks Data Platform (HDP), you can use Hive HBase integration to perform READ and WRITE operations on the HBase tables. HBase is integrated with Hive using the StorageHandler. You can access the data through both Hive and HBase.

## Prerequisites

You must complete the following steps before configuring the Hive and HBase.

### Procedure

- Install ZooKeeper, HBase, and Hive through Ambari.
- Install the required version of Hadoop.
- Add all the required jars:

- Hive-hbase-handler.jar is available on the Hive client auxpath
- ZooKeeper jar
- Hbase server jar
- Hbase client jar

# Configuring HBase and Hive

Follow this step to complete the configuration:

### Procedure

Modify the hive-site.xml configuration file. Add the required path to the jars. The jars will be used by Hive to write data into the HBase. The full list of JARs to add can be seen by running the commandhbase mapredcp on the command-line.

Note: Each JAR name contains the HDP version -- the list of JARs will not work across different HDP releases, but it can be easily recreated.

```
<property>
<name>hive.aux.jars.path</name>
<value>
file:///usr/hdp/3.0.1.0-61/hbase/lib/commons-lang3-3.6.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-zookeeper-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-mapreduce-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/jackson-annotations-2.9.5.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-shaded-miscellaneous-2.1.0.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/jackson-databind-2.9.5.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-hadoop-
compat-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-metrics-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-client-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-protocol-
shaded-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/jackson-core-2.9.5.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/protobuf-java-2.5.0.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-shaded-netty-2.1.0.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/metrics-core-3.2.1.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-server-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-hadoop2-
compat-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-metrics-api-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-common-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-protocol-2.0.0.3.0.1.0-61.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/hbase-shaded-protobuf-2.1.0.jar,
file:///usr/hdp/3.0.1.0-61/hbase/lib/htrace-core4-4.2.0-incubating.jar,
file:///usr/hdp/3.0.1.0-61/zookeeper/zookeeper-3.4.6.3.0.1.0-61.jar
</value>
</property>
```

Note: Modify the file paths and file names to the path that appears on your machine.

# Using HBase Hive integration

Before you begin to use the Hive HBase integration, complete the following steps:

**Procedure**

- Use the HBaseStorageHandler to register the HBase tables with the Hive metastore. You can also register the Hbase tables directly in Hive using the HiveHBaseTableInputFormat and HiveHBaseTableOutputFormat classes.
- As part of the registration process, specify a column mapping. There are two SERDEPROPERTIES that controls the HBase column mapping to Hive:

    - Hbase.columns.mapping
    - Hbase.table.default.storage.type

## HBase Hive integration example

A change to Hive in HDP 3.0 is that all StorageHandlers must be marked as "external". There is no such thing as an non-external table created by a StorageHandler. If the corresponding HBase table exists when the Hive table is created, it will mimic the HDP 2.x semantics of an "external" table. If the corresponding HBase table does not exist when the Hive table is created, it will mimic the HDP 2.x semantics of a non-external table (e.g. the HBase table is dropped when the Hive table is dropped).

**Procedure**

1. From the Hive shell, create a HBase table:

```
CREATE EXTERNAL TABLE hbase_hive_table (key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
TBLPROPERTIES ("hbase.table.name" = "hbase_hive_table",
  "hbase.mapred.output.outputtable" = "hbase_hive_table");
```

The hbase.columns.mapping property is mandatory. The hbase.table.name property is optional.The hbase.mapred.output.outputtable  property is optional; It is needed, if you plan to insert data to the table

2. From the HBase shell, access the hbase_hive_table:

```
$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Version: 0.20.3, r902334, Mon Jan 25 13:13:08 PST 2010

hbase(main):001:0> list hbase_hive_table

1 row(s) in 0.0530 seconds
hbase(main):002:0> describe hbase_hive_table
Table hbase_hive_table is ENABLED
hbase_hive_table COLUMN FAMILIES DESCRIPTION{NAME => 'cf',
 DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE =>
 '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =>
 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY
 => 'false', BLOCKCACHE => 'true'} 1 row(s) in 0.2860 seconds

hbase(main):003:0> scan "hbase_hive_table "

ROW                              COLUMN+CELL

0 row(s) in 0.0060 seconds
```

3. Insert the data into the HBase table through Hive:

```
INSERT OVERWRITE TABLE HBASE_HIVE_TABLE SELECT * FROM pokes WHERE foo=98;
```

**4.** From the HBase shell, verify that the data got loaded:

```
hbase(main):009:0> scan "hbase_hive_table"
ROW                            COLUMN+CELL

 98                                column=cf1:val, timestamp=1267737987733,
 value=val_98
1 row(s) in 0.0110 seconds
```

**5.** From Hive, query the HBase data to view the data that is inserted in the hbase_hive_table:

```
hive> select * from HBASE_HIVE_TABLE;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
98 val_98
Time taken: 4.582 seconds
```

# Using Hive to access an existing HBase table example

Use the following steps to access the existing HBase table through Hive.

### Procedure

- You can access the existing HBase table through Hive using the CREATE EXTERNAL TABLE:

```
CREATE EXTERNAL TABLE hbase_table_2(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key
,cf1:val")
TBLPROPERTIES("hbase.table.name" = "some_existing_table",
  "hbase.mapred.output.outputtable" = "some_existing_table");
```

- You can use different type of column mapping to map the HBase columns to Hive:

  - Multiple Columns and Families

    To define four columns, the first being the rowkey: ":key,cf:a,cf:b,cf:c"
  - Hive MAP to HBase Column Family

    When the Hive datatype is a Map, a column family with no qualifier might be used. This will use the keys of the Map as the column qualifier in HBase: "cf:"
  - Hive MAP to HBase Column Prefix

    When the Hive datatype is a Map, a prefix for the column qualifier can be provided which will be prepended to the Map keys: "cf:prefix_.*"

    Note: The prefix is removed from the column qualifier as compared to the key in the Hive Map. For example, for the above column mapping, a column of "cf:prefix_a" would result in a key in the Map of "a".
- You can also define composite row keys. Composite row keys use multiple Hive columns to generate the HBase row key.

  - Simple Composite Row Keys

    A Hive column with a datatype of Struct will automatically concatenate all elements in the struct with the termination character specified in the DDL.
  - Complex Composite Row Keys and HBaseKeyFactory

Custom logic can be implemented by writing Java code to implement a KeyFactory and provide it to the DDL using the table property key "hbase.composite.key.factory".

## Understanding Bulk Loading

A common pattern in HBase to obtain high rates of data throughput on the write path is to use "bulk loading". This generates HBase files (HFiles) that have a specific format instead of shipping edits to HBase RegionServers. The Hive integration has the ability to generate HFiles, which can be enabled by setting the property "hive.hbase.generatehfiles" to true, for example, `set hive.hbase.generatehfiles=true`. Additionally, the path to a directory which to write the HFiles must also be provided, for example,`set hfile.family.path=/tmp/hfiles".

After the Hive query finishes, you must execute the "completebulkload" action in HBase to bring the files "online" in your HBase table. For example, to finish the bulk load for files in "/tmp/hfiles" for the table "hive_data", you might run on the command-line:

```
$ hbase completebulkload /tmp/hfiles hive_data
```

## Understanding HBase Snapshots

When an HBase snapshot exists for an HBase table which a Hive table references, you can choose to execute queries over the "offline" snapshot for that table instead of the table itself.

First, set the property to the name of the HBase snapshot in your Hive script: `set hive.hbase.snapshot.name=my_snapshot`. A temporary directory is required to run the query over the snapshot. By default, a directory is chosen inside of "/tmp" in HDFS, but this can be overridden by using the property "hive.hbase.snapshot.restoredir".

# HBase Best Practices

HBase RowKey length value best practice provides limitation information that you can apply to avoid error while indexing the table.

### HBase RowKey length value

You should limit the value of each HBase RowKey length to 32767 bytes. Exceeding this value causes an exceptional error while indexing the table. This restriction is applicable in addition to the maximum allowed size of an individual cell, inclusive of value and all key components as defined by the hbase.server.keyvalue.maxsize value. The default value for RowKey length is 1 MB, which avoids server memory errors.