Cloudera Runtime 7.0.2

Managing Apache HBase

Date published: 2019-09-23
Date modified:

CLOUDERA

https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 ("ASLv2"), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER'S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Starting and Stopping HBase using Cloudera Manager.................................... 5
Start HBase.. 5
Stop HBase... 5
Graceful HBase Shutdown.. 6
Gracefully shut down an HBase RegionServer... 6
Gracefully shut down the HBase service...6

Importing data into HBase...7
Choose the right import method.. 7
Use snapshots..7
Use CopyTable.. 9
Use BulkLoad.. 9
Use cases for BulkLoad... 11
Use cluster replication.. 11
Use Sqoop.. 12
Use Spark... 13
Use a custom MapReduce job.. 13
Use HashTable and SyncTable Tool..14
HashTable/SyncTable tool configuration... 14
Synchronize table data using HashTable/SyncTable tool.. 15

Writing data to HBase.. 15
Variations on Put... 16
Versions..16
Deletion.. 16
Examples.. 17

Reading data from HBase... 17
Perform scans using HBase Shell... 19

HBase filtering..19
Dynamically loading a custom filter..19
Logical operators, comparison operators and comparators...20
Compound operators...21
Filter types.. 21
HBase Shell example..24
Java API example...24

HBase online merge.. 28

Move HBase Master Role to another host..28

Expose HBase metrics to a Ganglia server..29

Starting and Stopping HBase using Cloudera Manager

You can start and stop your HBase cluster using Cloudera Manager.

Start HBase

You can start HBase clusters or individual hosts following these instructions.

Procedure

1. In Cloudera Manager, select the HBase service.
2. Click the Actions button.
3. Select Start.

If you want to restart a running cluster, do the following:

4. Click the Actions button.
5. Select Restart or Rolling Restart.

   A rolling restart restarts each RegionServer, on at a time, after a a grace period.

If you want to restart the Thrift service, do the following:

6. Click Instances.
7. Select the HBase Thrift Server instance.
8. Select Actions for Selected.
9. Select Restart.

Related Information

Configure the graceful shutdown timeout property

Stop HBase

You can stop HBase clusters or individual hosts following these instructions.

Procedure

If you want to stop a single RegionServer, do the following:

1. In Cloudera Manager, select the HBase service.
2. Click the Instance tab.
3. From the list of Role Instances, select the RegionServer or RegionServers you want to stop.
4. Select Actions for Selected.
5. Select Stop.

   Stop happens immediately and does not redistribute the regions. It issues a SIGTERM(kill -5) signal.

If you want to stop a single HMaster, do the following:

6. In Cloudera Manager, select the HBase service.
7. Click the Instance tab.
8. From the list of Role Instances, select the HMaster or HMasters you want to stop.
9. Select Actions for Selected.
10. Select Stop.

    Stop happens immediately and does not redistribute the regions. It issues a SIGTERM(kill -5) signal.

If you want to stop the entire cluster, do the following:

11. In Cloudera Manager, select the HBase service.
12. Click the Actions button.
13. Select Stop.

Related Information

Configure the graceful shutdown timeout property

Graceful HBase Shutdown

Graceful Shutdown causes the regions to be redistributed to other RegionServers, increasing availability during the RegionServer outage.

A graceful shutdown of an HBase RegionServer allows the regions hosted by that RegionServer to be moved to other RegionServers before stopping the RegionServer. You can perform a graceful shutdown of either an HBase RegionServer or the entire service.

Cloudera Manager waits for an interval determined by the Graceful Shutdown timeout interval, which defaults to three minutes. If the graceful stop does not succeed within this interval, the RegionServer is stopped with a SIGKILL (kill -9) signal. Recovery will be initiated on affected regions.

To increase the speed of a rolling restart of the HBase service, set the Region Mover Threads property to a higher value. This increases the number of regions that can be moved in parallel, but places additional strain on the HMaster. In most cases, Region Mover Threads should be set to 5 or lower.

Related Information

Configure the graceful shutdown timeout property

Gracefully shut down an HBase RegionServer

You can gracefully shut down an HBase RegionServer using Cloudera Manager.

Procedure

1. In Cloudera Manager, select the HBase service.
2. Click the Instance tab.
3. From the list of Role Instances, select the RegionServer or RegionServers you want to shut down gracefully.
4. Click Actions for Selected.
5. Select Decommission (Graceful Stop).

   If you cancel the graceful shutdown before the Graceful Shutdown Timeout expires, you can still manually stop a RegionServer by selecting Actions for Selected > Stop, which sends a SIGTERM (kill -5) signal.

Related Information

Configure the graceful shutdown timeout property

Gracefully shut down the HBase service

You can gracefully shut down the HBase Service using Cloudera Manager.

Procedure

1. In Clouder Manager, select the HBase service.
2. Click the Actions button.
3. Select Stop.
   Cloudera Manager tries to perform an HBase Master-driven graceful shutdown for the length of the configured Graceful Shutdown Timeout, after which it abruptly shuts down the whole service. The default value of Graceful Shutdown Timeout is three minutes.

Importing data into HBase

In different scenarios different methods can be used to import data into HBase.

Choose the right import method

Learn about how to choose the right data import method.

The method you use for importing data into HBase depends on several factors:

- The location, size, and format of your existing data
- Whether you need to import data once or periodically over time
- Whether you want to import the data in bulk or stream it into HBase regularly
- How fresh the HBase data needs to be

This topic helps you choose the correct method or composite of methods and provides example workflows for each method.

Always run HBase administrative commands as the HBase user (typically hbase).

If the data is already in an HBase table:

- To move the data from one HBase cluster to another, use snapshot and either the clone_snapshot or ExportSnapsh ot utility; or, use the CopyTable utility.
- To move the data from one HBase cluster to another without downtime on either cluster, use replication.

If the data currently exists outside HBase:

- If possible, write the data to HFile format, and use a BulkLoad to import it into HBase. The data is immediately available to HBase and you can bypass the normal write path, increasing efficiency.
- If you prefer not to use bulk loads, and you are using a tool such as Pig, you can use it to import your data.

If you need to stream live data to HBase instead of import in bulk:

- Write a Java client using the Java API, or use the Apache Thrift Proxy API to write a client in a language supported by Thrift.
- Stream data directly into HBase using the REST Proxy API in conjunction with an HTTP client such as wget or curl.
- Use Flume or Spark.

Most likely, at least one of these methods works in your situation. If not, you can use MapReduce directly. Test the most feasible methods with a subset of your data to determine which one is optimal.

Use snapshots

A snapshot captures the state of a table at the time the snapshot was taken

Cloudera recommends snapshots instead of CopyTable where possible. Because no data is copied when a snapshot is taken, the process is very quick. As long as the snapshot exists, cells in the snapshot are never deleted from HBase, even if they are explicitly deleted by the API. Instead, they are archived so that the snapshot can restore the table to its state at the time of the snapshot.

After taking a snapshot, use the clone_snapshot command to copy the data to a new (immediately enabled) table in the same cluster, or the Export utility to create a new table based on the snapshot, in the same cluster. This is a copy-on-write operation. The new table shares HFiles with the original table until writes occur in the new table but not the old table, or until a compaction or split occurs in either of the tables. This can improve performance in the short term compared to CopyTable.

To export the snapshot to a new cluster, use the ExportSnapshot utility, which uses MapReduce to copy the snapshot to the new cluster. Run the ExportSnapshot utility on the source cluster, as a user with HBase and HDFS write permission on the destination cluster, and HDFS read permission on the source cluster. This creates the expected amount of IO load on the destination cluster. Optionally, you can limit bandwidth consumption, which affects IO on the destination cluster. After the ExportSnapshot operation completes, you can see the snapshot in the new cluster using the list_snapshot command, and you can use the clone_snapshot command to create the table in the new cluster from the snapshot.

For full instructions for the snapshot and clone_snapshot HBase Shell commands, run the HBase Shell and type help snapshot. The following example takes a snapshot of a table, uses it to clone the table to a new table in the same cluster, and then uses the ExportSnapshot utility to copy the table to a different cluster, with 16 mappers and limited to 200 Mb/sec bandwidth.

```
$ bin/hbase shell
   hbase(main):005:0> snapshot 'TestTable', 'TestTableSnapshot'
   0 row(s) in 2.3290 seconds

   hbase(main):006:0> clone_snapshot 'TestTableSnapshot', 'NewTestTable'
   0 row(s) in 1.3270 seconds

   hbase(main):007:0> describe 'NewTestTable'
   DESCRIPTION                                          ENABLED
   'NewTestTable', {NAME => 'cf1', DATA_BLOCK_ENCODING true
   => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE
   => '0', VERSIONS => '1', COMPRESSION => 'NONE', MI
   N_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_C
   ELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY =>
   'false', BLOCKCACHE => 'true'}, {NAME => 'cf2', DA
   TA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
   REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESS
   ION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER
   ', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '655
   36', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
   1 row(s) in 0.1280 seconds
   hbase(main):008:0> quit

   $ hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot TestTa
bleSnapshot -copy-to file:///tmp/hbase -mappers 16 -bandwidth 200
   14/10/28 21:48:16 INFO snapshot.ExportSnapshot: Copy Snapshot Manifest
   14/10/28 21:48:17 INFO client.RMProxy: Connecting to ResourceManager at a
1221.example.com/192.0.2.121:8032
   14/10/28 21:48:19 INFO snapshot.ExportSnapshot: Loading Snapshot 'TestT
ableSnapshot' hfile list
   14/10/28 21:48:19 INFO Configuration.deprecation: hadoop.native.lib is d
eprecated. Instead, use io.native.lib.available
   14/10/28 21:48:19 INFO util.FSVisitor: No logs under directory:hdfs://a12
21.example.com:8020/hbase/.hbase-snapshot/TestTableSnapshot/WALs
   14/10/28 21:48:20 INFO mapreduce.JobSubmitter: number of splits:0
   14/10/28 21:48:20 INFO mapreduce.JobSubmitter: Submitting tokens for job:
 job_1414556809048_0001
   14/10/28 21:48:20 INFO impl.YarnClientImpl: Submitted application appli
cation_1414556809048_0001
   14/10/28 21:48:20 INFO mapreduce.Job: The url to track the job: http://
a1221.example.com:8088/proxy/application_1414556809048_0001/
   14/10/28 21:48:20 INFO mapreduce.Job: Running job: job_1414556809048_0001
   14/10/28 21:48:36 INFO mapreduce.Job: Job job_1414556809048_0001 running
 in uber mode : false
   14/10/28 21:48:36 INFO mapreduce.Job:  map 0% reduce 0%
   14/10/28 21:48:37 INFO mapreduce.Job: Job job_1414556809048_0001 comple
ted successfully
   14/10/28 21:48:37 INFO mapreduce.Job: Counters: 2
   Job Counters
```

```
   Total time spent by all maps in occupied slots (ms)=0
   Total time spent by all reduces in occupied slots (ms)=0
   14/10/28 21:48:37 INFO snapshot.ExportSnapshot: Finalize the Snapshot
Export
   14/10/28 21:48:37 INFO snapshot.ExportSnapshot: Verify snapshot integrity
   14/10/28 21:48:37 INFO Configuration.deprecation: fs.default.name is depr
ecated. Instead, use fs.defaultFS
   14/10/28 21:48:37 INFO snapshot.ExportSnapshot: Export Completed: TestT
ableSnapshot
```

The url to track the job: contains the URL from which you can track the ExportSnapshot job. When it finishes, a new set of HFiles, comprising all of the HFiles that were part of the table when the snapshot was taken, is created at the HDFS location you specified.

You can use the SnapshotInfo command-line utility included with HBase to verify or debug snapshots.

Use CopyTable

CopyTable uses HBase read and write paths to copy part or all of a table to a new table in either the same cluster or a different cluster.

CopyTable causes read load when reading from the source, and write load when writing to the destination. Region splits occur on the destination table in real time as needed. To avoid these issues, use snapshot and export commands instead of CopyTable. Alternatively, you can pre-split the destination table to avoid excessive splits. The destination table can be partitioned differently from the source table. See this section of the Apache HBase documentation for more information.

Edits to the source table after the CopyTable starts are not copied, so you may need to do an additional CopyTable operation to copy new data into the destination table. Run CopyTable as follows, using --help to see details about possible parameters.

```
$ ./bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --help
    Usage: CopyTable [general options] [--starttime=X] [--endtime=Y] [--new
.name=NEW] [--peer.adr=ADR] <tablename>
```

The starttime/endtime and startrow/endrow pairs function in a similar way: if you leave out the first of the pair, the first timestamp or row in the table is the starting point. Similarly, if you leave out the second of the pair, the operation continues until the end of the table. To copy the table to a new table in the same cluster, you must specify --new.na me, unless you want to write the copy back to the same table, which would add a new version of each cell (with the same data), or just overwrite the cell with the same value if the maximum number of versions is set to 1. To copy the table to a new table in a different cluster, specify --peer.adr and optionally, specify a new table name.

The following example creates a new table using HBase Shell in non-interactive mode, and then copies data in two ColumnFamilies in rows starting with timestamp 1265875194289 and including the last row before the CopyTable started, to the new table.

```
echo create 'NewTestTable', 'cf1', 'cf2', 'cf3' | bin/hbase shell --non-inte
ractive
    bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --starttime=126587
5194289  --families=cf1,cf2,cf3 --new.name=NewTestTable TestTable
```

Snapshots are recommended instead of CopyTable for most situations.

Use BulkLoad

In many situations, writing HFiles programmatically with your data, and bulk-loading that data into HBase on the RegionServer, has advantages over other data ingest mechanisms.

HBase uses an internal file format called HFile to store its data on disk. BulkLoad operations bypass the write path completely, providing the following benefits:

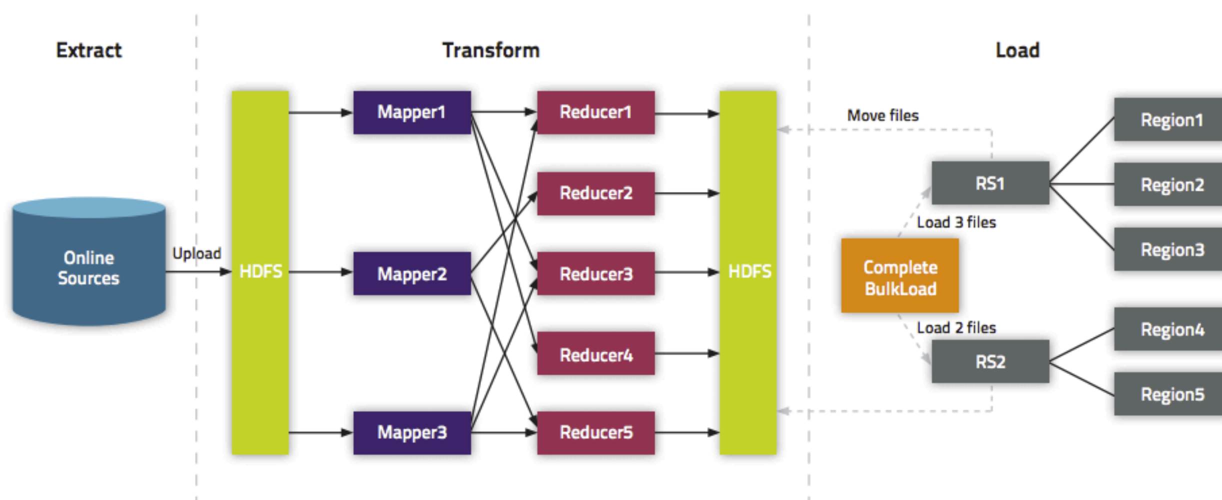
- The data is available to HBase immediately but does cause additional load or latency on the cluster when it appears.
- BulkLoad operations do not use the write-ahead log (WAL) and do not cause flushes or split storms.
- BulkLoad operations do not cause excessive garbage collection.

If you use BulkLoads with HBase, your workflow is similar to the following:

1. Extract your data from its existing source. For instance, if your data is in a MySQL database, you might run the mysqldump command. The process you use depends on your data. If your data is already in TSV or CSV format, skip this step and use the included ImportTsv utility to process your data into HFiles. See the ImportTsv documentation for details.
2. Process your data into HFile format. See http://hbase.apache.org/book.html#_hfile_format_2 for details about HFile format. Usually you use a MapReduce job for the conversion, and you often need to write the Mapper yourself because your data is unique. The job must to emit the row key as the Key, and either a KeyValue, a Put, or a Delete as the Value. The Reducer is handled by HBase; configure it using HFileOutputFormat.configureIncrementalLoad() and it does the following:
   - Inspects the table to configure a total order partitioner
   - Uploads the partitions file to the cluster and adds it to the DistributedCache
   - Sets the number of reduce tasks to match the current number of regions
   - Sets the output key/value class to match HFileOutputFormat requirements
   - Sets the Reducer to perform the appropriate sorting (either KeyValueSortReducer or PutSortReducer)
3. One HFile is created per region in the output folder. Input data is almost completely re-written, so you need available disk space at least twice the size of the original data set. For example, for a 100 GB output from mysq ldump, you should have at least 200 GB of available disk space in HDFS. You can delete the original input file at the end of the process.
4. Load the files into HBase. Use the LoadIncrementalHFiles command (more commonly known as the completebulkload tool), passing it a URL that locates the files in HDFS. Each file is loaded into the relevant region on the RegionServer for the region. You can limit the number of versions that are loaded by passing the --versions= N option, where N is the maximum number of versions to include, from newest to oldest (largest timestamp to smallest timestamp).

   If a region was split after the files were created, the tool automatically splits the HFile according to the new boundaries. This process is inefficient, so if your table is being written to by other processes, you should load as soon as the transform step is done.

The following illustration shows the full BulkLoad process.



For an explanation of how secure BulkLoad works, see Bulk Loading .

Extra Steps for BulkLoad With Encryption Zones

When using BulkLoad to import data into HBase in the a cluster using encryption zones, the following information is important.

- Both the staging directory and the directory into which you place your generated HFiles need to be within HBase's encryption zone (generally under the /hbase directory). Before you can do this, you need to change the permissions of /hbase to be world-executable but not world-readable (rwx--x--x, or numeric mode 711).
- You also need to configure the HMaster to set the permissions of the HBase root directory correctly. If you use Cloudera Manager, edit the Master Advanced Configuration Snippet (Safety Valve) for hbase-site.xml. Otherwise, edit hbase-site.xml on the HMaster. Add the following:

```
<property>
        <name>hbase.rootdir.perms</name>
        <value>711</value>
        </property>
```

  If you skip this step, a previously-working BulkLoad setup will start to fail with permission errors when you restart the HMaster.

Use cases for BulkLoad

There are three typical use cases when using BulkLoad can be advantageous.

- Loading your original dataset into HBase for the first time - Your initial dataset might be quite large, and bypassing the HBase write path can speed up the process considerably.
- Incremental Load - To load new data periodically, use BulkLoad to import it in batches at your preferred intervals. This alleviates latency problems and helps you to achieve service-level agreements (SLAs). However, one trigger for compaction is the number of HFiles on a RegionServer. Therefore, importing a large number of HFiles at frequent intervals can cause major compactions to happen more often than they otherwise would, negatively impacting performance. You can mitigate this by tuning the compaction settings such that the maximum number of HFiles that can be present without triggering a compaction is very high, and relying on other factors, such as the size of the Memstore, to trigger compactions.
- Data needs to originate elsewhere - If an existing system is capturing the data you want to have in HBase and needs to remain active for business reasons, you can periodically BulkLoad data from the system into HBase so that you can perform operations on it without impacting the system.

Use cluster replication

If your data is already in an HBase cluster, replication is useful for getting the data into additional HBase clusters.

In HBase, cluster replication refers to keeping one cluster state synchronized with that of another cluster, using the write-ahead log (WAL) of the source cluster to propagate the changes. Replication is enabled at column family granularity. Before enabling replication for a column family, create the table and all column families to be replicated, on the destination cluster.

Cluster replication uses an active-push methodology. An HBase cluster can be a source (also called active, meaning that it writes new data), a destination (also called passive, meaning that it receives data using replication), or can fulfill both roles at once. Replication is asynchronous, and the goal of replication is consistency.

When data is replicated from one cluster to another, the original source of the data is tracked with a cluster ID, which is part of the metadata. All clusters that have already consumed the data are also tracked. This prevents replication loops.

Related Information

Using HBase Replication

Use Sqoop

Sqoop can import records into a table in HBase. It has an out-of-the-box support for HBase

There are two mandatory options you must specify when using the sqoop import command to import data into HBase using Sqoop:

- --hbase-table: Specifies the name of the table in HBase to which you want to import your data.
- --column-family: Specifies into which column family Sqoop imports the data of your tables.

For example, you can import the table cities into an already existing HBase table with the same name and use the column family name world:

```
sqoop import --connect jdbc:mysql://mysql.example.com/sqoop --username sqoop
 --password sqoop --table cities --hbase-table cities --column-family world
```

If the target table and column family do not exist, the Sqoop job will exit with an error. You must create the target table and column family before running an import. If you specify --hbase-create-table, Sqoop creates the target table and column family if they do not exist, using the default parameters from your HBase configuration.

Sqoop needs to identify which RDBMS column is used as row key column in the HBase table. There are three ways to do this:

- By default, with the column name specified in the --split-by option
- With the primary key of the table, if it is available
- With the --hbase-row-key parameter, which overrides both the --split-by option and the primary key of the table

For more information on data insertion into HBase, see Sqoop User Guide.

Import NULL Column Updates into HBase

You can specify how Sqoop handles RDBMS table column updated to NULL during incremental import.

There are two modes for this, ignore and delete. You can specify the mode using the --hbase-null-incrementel-mode option:

- -ignore: This is the default value. If the source table's column is updated to NULL, the target HBase table will still show the previous value for that column.
- -delete: If the source table's column is updated to NULL, all previous versions of the column will be deleted from HBase. When checking the column in HBase using the Java API, a null value will be displayed.

Examples:

Execute an incremental import to an HBase table and ignore the columns which were updated to NULL in the relational database:

```
sqoop import --connect $CONN --username $USER --password $PASS --t
able "hbase_test" --hbase-table hbase_test --column-family data -m 1 --incr
emental lastmodified --check-column date_modified --last-value "2017-12-15
 10:58:44.0" --merge-key id --hbase-null-incremental-mode ignore
```

Execute an incremental import to an HBase table and delete all the versions of the columns which were updated to NULL in the relational database:

```
                sqoop import --connect $CONN --username $USER --passw
ord $PASS --table "hbase_test" --hbase-table hbase_test --column-family data
 -m 1 --incremental lastmodified --check-column date_modified --last-valu
e "2017-12-15 10:58:44.0" --merge-key id --hbase-null-incremental-mode d
elete
```

Use Spark

You can write data to HBase from Apache Spark using def saveAsHadoopDataset(conf: JobConf): Unit.

This example is adapted from a post on the spark-users mailing list.

```
// Note: mapred package is used, instead of the
                    // mapreduce package which contains new hadoop APIs.

                    import org.apache.hadoop.hbase.mapred.TableOutputFormat
                    import org.apache.hadoop.hbase.client
                    // ... some other settings

                    val conf = HBaseConfiguration.create()

                    // general hbase settings
                    conf.set("hbase.rootdir",
                    "hdfs://" + nameNodeURL + ":" + hdfsPort + "/hbase")
                    conf.setBoolean("hbase.cluster.distributed", true)
                    conf.set("hbase.zookeeper.quorum", hostname)
                    conf.setInt("hbase.client.scanner.caching", 10000)
                    // ... some other settings

                    val jobConfig: JobConf = new JobConf(conf, this.getClass
)

                    // Note:  TableOutputFormat is used as deprecated code
                    // because JobConf is an old hadoop API
                    jobConfig.setOutputFormat(classOf[TableOutputFormat])
                    jobConfig.set(TableOutputFormat.OUTPUT_TABLE, outputTabl
e)
```

Next, provide the mapping between how the data looks in Spark and how it should look in HBase. The following example assumes that your HBase table has two column families, col_1 and col_2, and that your data is formatted in sets of three in Spark, like (row_key, col_1, col_2).

```
def convert(triple: (Int, Int, Int)) = {
                val p = new Put(Bytes.toBytes(triple._1))
                p.add(Bytes.toBytes("cf"),
                Bytes.toBytes("col_1"),
                Bytes.toBytes(triple._2))
                p.add(Bytes.toBytes("cf"),
                Bytes.toBytes("col_2"),
                Bytes.toBytes(triple._3))
                (new ImmutableBytesWritable, p)
                }
```

To write the data from Spark to HBase, you might use:

```
new PairRDDFunctions(localData.map(convert)).saveAsHadoopDataset(jobConfig)
```

Use a custom MapReduce job

Many of the methods to import data into HBase use MapReduce implicitly. If none of those approaches fit your needs, you can use MapReduce directly to convert data to a series of HFiles or API calls for import into HBase

In this way, you can import data from Avro, Parquet, or another format into HBase, or export data from HBase into another format, using API calls such as TableOutputFormat, HFileOutputFormat, and TableInputFormat.

Use HashTable and SyncTable Tool

HashTable/SyncTable is a two steps tool for synchronizing table data without copying all cells in a specified row key/time period range.

The HashTable/SyncTable tool can be used for partial or entire table data synchronization, under the same or remote cluster. Both the HashTable and the SyncTable step are implemented as a MapReduce job.

The first step, HashTable, creates hashed indexes for batch of cells on the source table and output those as results. The source table is the table whose state is copied to its counterpart.

The second step, SyncTable, scans the target table and calculates hash indexes for table cells. Then these hashes are compared to the HashTable step outputs. So, SyncTable scans and compares cells for diverging hashes and updating only the mismatching cells.

This results in less network traffic or data transfers than other methods, for example CopyTable, which can impact performance when large tables are synchronized on remote clusters.

Remote clusters are often deployed on different Kerberos Realms. SyncTable support cross realm authentication, allowing a SyncTable process running on the target cluster to connect to the source cluster and read both the HashTable output files and the given HBase table when performing the required comparisons.

HashTable/SyncTable tool configuration

You can configure the HashTable/SyncTable tool for your specific needs.

Using the batchsize option

You can define the amount of cell data for a given region that is hashed together in a single hash value using the batchsize option, which sets the batchsize property. Sizing this property has a direct impact on the synchronization efficiency. If the batch size is increased, larger chunks are hashed.

If only a few differences are expected between the two tables, using a bit larger batch size can be beneficial, as less scans are executed by mapper tasks of SyncTable.

However, if relatively frequent differences are expected between the tables, using a large batch size can cause frequent mismatches of hash values, as the probability of finding at least one mismatch in a batch is increased.

The following is an example of sizing this property:

```
$ hbase org.apache.hadoop.hbase.mapreduce.HashTable --batchsize=32000 --numh
ashfiles=50 --starttime=1265875194289 --endtime=1265878794289 --families=cf2
,cf3 TestTableA /hashes/testTable
```

Creating a read-only report

You can use the dryrun option in the second, SyncTable, step to create a read only report. It produces only COUNTERS indicating the differences between the two tables, but does not perform any actual changes. It can be used as an alternative of the VerifyReplication tool.

The following is an example of using this option:

```
$ hbase org.apache.hadoop.hbase.mapreduce.SyncTable --dryrun=true --sourcezk
cluster=zk1.example.com,zk2.example.com,zk3.example.com:2181:/hbase hdfs://
nn:8020/hashes/testTable testTableA testTableB
```

Handling missing cells

By default, SyncTable changes the target table to make it an exact copy of the source table, at least for the specified startrow-stoprow or starttime-endtime.

Setting doDeletes to false modifies the default behaviour to not delete target cells that are missing on the source table. Similarly, setting doPuts to false modifies the default behaviour to not add missing cells to target table. If you set both doDeletes and doPuts to false, the result will be the same as setting dryrun to true.

In the case of two-way replication or other scenarios where both source and target clusters can have data ingested, Cloudera recommends to set doDeletes to false. Otherwise any additional cells inserted on the SyncTable target cluster and not yet replicated to the source cluster would be deleted, and potentially lost permanently.

Synchronize table data using HashTable/SyncTable tool

The HashTable/SyncTable tool can be used for partial or entire table data synchronization, under the same or remote cluster.

Before you begin

- Ensure that all RegionServers/DataNodes on the source cluster is accessible by the NodeManagers on the target cluster where SyncTable job tasks will be running.
- In the case of secured clusters, the user on the target cluster who executes the SyncTable job must be able to do the following on the HDFS and HBase services of the source cluster:
  - Authenticate: for example, using centralized authentication or cross-realm setup.
  - Be authorized: having at least read permission.

Procedure

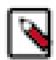
1. Run HashTable on the source table cluster: HashTable [options] <tablename> <outputpath>

   The following is an example to hash the TesTable in 32kB batches for a 1 hour window into 50 files:

   ```
   $ hbase org.apache.hadoop.hbase.mapreduce.HashTable --batchsize=32000 --n
   umhashfiles=50 --starttime=1265875194289 --endtime=1265878794289 --famil
   ies=cf2,cf3 TestTableA /hashes/testTable
   ```
2. Run SyncTable on the target cluster: SyncTable [options] <sourcehashdir> <sourcetable> <targettable>

   The following is an example for a dry run SyncTable of tableA from a remote source cluster to a local tableB on the target cluster:

   ```
   $ hbase org.apache.hadoop.hbase.mapreduce.SyncTable --dryrun=true --sour
   cezkcluster=zk1.example.com,zk2.example.com,zk3.example.com:2181:/hbase
    hdfs://nn:8020/hashes/testTable testTableA testTableB
   ```



Note:

For more detailed information regarding HashTable/SynTable tool options, use hbase org.apache.hadoop.hbase.mapreduce.SyncTable --help.

Writing data to HBase

To write data to HBase, you use methods of the Table class.

You can use the Java API directly, or use the HBase Shell, the REST API, the Thrift API, , or another client which uses the Java API indirectly. When you issue a Put, the coordinates of the data are the row, the column, and the timestamp. The timestamp is unique per version of the cell, and can be generated automatically or specified programmatically by your application, and must be a long integer.

Variations on Put

There are several different ways to write data into HBase.

- A Put operation writes data into HBase.
- A Delete operation deletes data from HBase. What actually happens during a Delete depends upon several factors.
- A CheckAndPut operation performs a Scan before attempting the Put, and only does the Put if a value matches what is expected, and provides row-level atomicity.
- A CheckAndDelete operation performs a Scan before attempting the Delete, and only does the Delete if a value matches what is expected.
- An Increment operation increments values of one or more columns within a single row, and provides row-level atomicity.

Refer to the API documentation for a full list of methods provided for writing data to HBase.Different methods require different access levels and have other differences.

Versions

When you put data into HBase, a timestamp is required.

The timestamp can be generated automatically by the RegionServer or can be supplied by you. The timestamp must be unique per version of a given cell, because the timestamp identifies the version. To modify a previous version of a cell, for instance, you would issue a Put with a different value for the data itself, but the same timestamp.

HBase's behavior regarding versions is highly configurable. The maximum number of versions defaults to 1. You can change the default value for HBase by configuring hbase.column.max.version in hbase-site.xml, either using an advanced configuration snippet if you use Cloudera Manager, or by editing the file directly otherwise.

You can also configure the maximum and minimum number of versions to keep for a given column, or specify a default time-to-live (TTL), which is the number of seconds before a version is deleted. The following examples all use alter statements in HBase Shell to create new column families with the given characteristics, but you can use the same syntax when creating a new table or to alter an existing column family. This is only a fraction of the options you can specify for a given column family.

```
hbase> alter ‘t1#, NAME => ‘f1#, VERSIONS => 5
    hbase> alter ‘t1#, NAME => ‘f1#, MIN_VERSIONS => 2
    hbase> alter ‘t1#, NAME => ‘f1#, TTL => 15
```

HBase sorts the versions of a cell from newest to oldest, by sorting the timestamps lexicographically. When a version needs to be deleted because a threshold has been reached, HBase always chooses the "oldest" version, even if it is in fact the most recent version to be inserted. Keep this in mind when designing your timestamps. Consider using the default generated timestamps and storing other version-specific data elsewhere in the row, such as in the row key. If MIN_VERSIONS and TTL conflict, MIN_VERSIONS takes precedence.

Deletion

When you request for HBase to delete data, either explicitly using a Delete method or implicitly using a threshold such as the maximum number of versions or the TTL, HBase does not delete the data immediately

Instead, it writes a deletion marker, called a tombstone, to the HFile, which is the physical file where a given RegionServer stores its region of a column family. The tombstone markers are processed during major compaction operations, when HFiles are rewritten without the deleted data included.

Even after major compactions, "deleted" data may not actually be deleted. You can specify the KEEP_DELETED_CELLS option for a given column family, and the tombstones will be preserved in the HFile even after major compaction. One scenario where this approach might be useful is for data retention policies.

Another reason deleted data may not actually be deleted is if the data would be required to restore a table from a snapshot which has not been deleted. In this case, the data is moved to an archive during a major compaction, and only deleted when the snapshot is deleted. This is a good reason to monitor the number of snapshots saved in HBase.

Examples

These abbreviated example writes data to an HBase table.

This abbreviated example writes data to an HBase table using HBase Shell and then scans the table to show the result

```
hbase> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.1770 seconds

hbase> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0160 seconds

hbase> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0260 seconds
hbase> scan 'test'
ROW                    COLUMN+CELL
 row1                  column=cf:a, timestamp=1403759475114, value=value1
 row2                  column=cf:b, timestamp=1403759492807, value=value2
 row3                  column=cf:c, timestamp=1403759503155, value=value3
3 row(s) in 0.0440 seconds
```

This abbreviated example uses the HBase API to write data to an HBase table, using the automatic timestamp created by the Region Server.

```
publicstaticfinalbyte[] CF = "cf".getBytes();
publicstaticfinalbyte[] ATTR = "attr".getBytes();
...
Put put = new Put(Bytes.toBytes(row));
put.add(CF, ATTR, Bytes.toBytes( data));
htable.put(put);
```

This example uses the HBase API to write data to an HBase table, specifying the timestamp.

```
publicstaticfinalbyte[] CF = "cf".getBytes();
publicstaticfinalbyte[] ATTR = "attr".getBytes();
...
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555;  // just an example
put.add(CF, ATTR, explicitTimeInMs, Bytes.toBytes(data));
htable.put(put);
```

Reading data from HBase

The Get and Scan are the two ways to read data from HBase, aside from manually parsing HFiles.

A Get is simply a Scan limited by the API to one row. A Scan fetches zero or more rows of a table. By default, a Scan reads the entire table from start to end. You can limit your Scan results in several different ways, which affect the Scan's load in terms of IO, network, or both, as well as processing load on the client side. This topic is provided as a quick reference. Refer to the API documentation for Scan for more in-depth information. You can also perform Get and Scan using the HBase Shell, the REST API, or the Thrift API.

- Specify a startrow or stoprow or both. Neither startrow nor stoprow need to exist. Because HBase sorts rows lexicographically, it will return the first row after startrow would have occurred, and will stop returning rows after stoprow would have occurred.The goal is to reduce IO and network.
    - The startrow is inclusive and the stoprow is exclusive. Given a table with rows a, b, c, d, e, f, and startrow of c and stoprow of f, rows c-e are returned.
    - If you omit startrow, the first row of the table is the startrow.
    - If you omit the stoprow, all results after startrow (including startrow) are returned.
    - If startrow is lexicographically after stoprow, and you set Scan setReversed(boolean reversed) to true, the results are returned in reverse order. Given the same table above, with rows a-f, if you specify c as the stoprow and f as the startrow, rows f, e, and d are returned.

```
Scan()
     Scan(byte[] startRow)
     Scan(byte[] startRow, byte[] stopRow)
```

- Specify a scanner cache that will be filled before the Scan result is returned, setting setCaching to the number of rows to cache before returning the result. By default, the caching setting on the table is used. The goal is to balance IO and network load.

```
public Scan setCaching(int caching)
```

- To limit the number of columns if your table has very wide rows (rows with a large number of columns), use setBatch(int batch) and set it to the number of columns you want to return in one batch. A large number of columns is not a recommended design pattern.

```
public Scan setBatch(int batch)
```

- To specify a maximum result size, use setMaxResultSize(long), with the number of bytes. The goal is to reduce IO and network.

```
public Scan setMaxResultSize(long maxResultSize)
```

- When you use setCaching and setMaxResultSize together, single server requests are limited by either number of rows or maximum result size, whichever limit comes first.
- You can limit the scan to specific column families or columns by using addFamily or addColumn. The goal is to reduce IO and network. IO is reduced because each column family is represented by a Store on each RegionServer, and only the Stores representing the specific column families in question need to be accessed.

```
public Scan addColumn(byte[] family,
     byte[] qualifier)

     public Scan addFamily(byte[] family)
```

- You can specify a range of timestamps or a single timestamp by specifying setTimeRange or setTimestamp.

```
public Scan setTimeRange(long minStamp,
     long maxStamp)
     throws IOException

     public Scan setTimeStamp(long timestamp)
     throws IOException
```

- You can retrieve a maximum number of versions by using setMaxVersions.

```
public Scan setMaxVersions(int maxVersions)
```

- You can use a filter by using setFilter. .

```
public Scan setFilter(Filter filter)
```

- You can disable the server-side block cache for a specific scan using the API setCacheBlocks(boolean). This is an expert setting and should only be used if you know what you are doing.

Related Information

HBase Schell Overview

HBase filtering

Perform scans using HBase Shell

You can perform scans using HBase Shell, for testing or quick queries.

Use the following guidelines or issue the scan command in HBase Shell with no parameters for more usage information. This represents only a subset of possibilities.

```
# Display usage information
hbase> scan
# Scan all rows of table 't1'
hbase> scan 't1'
# Specify a startrow, limit the result to 10 rows, and only return selected
 columns
hbase> scan 't1', {COLUMNS => ['c1', 'c2'], LIMIT => 10, STARTROW => 'xyz'}

# Specify a timerange
hbase> scan 't1', {TIMERANGE => [1303668804, 1303668904]}

# Specify a custom filter
hbase> scan 't1', {FILTER => org.apache.hadoop.hbase.filter.ColumnPaginatio
nFilter.new(1, 0)}

# Specify a row prefix filter and another custom filter
hbase> scan 't1', {ROWPREFIXFILTER => 'row2',
                         FILTER => (QualifierFilter (>=, 'binary:xyz')) AND
 (TimestampsFilter ( 123, 456))}

# Disable the block cache for a specific scan (experts only)
hbase> scan 't1', {COLUMNS => ['c1', 'c2'], CACHE_BLOCKS => false}
```

HBase filtering

When reading data from HBase using Get or Scan operations, you can use custom filters to return a subset of results to the client.

While this does not reduce server-side IO, it does reduce network bandwidth and reduces the amount of data the client needs to process. Filters are generally used using the Java API, but can be used from HBase Shell for testing and debugging purposes.

HBase filters take zero or more arguments, in parentheses. Where the argument is a string, it is surrounded by single quotes ('string').

Related Information

Reading data from HBase

Dynamically loading a custom filter

Cloudera Runtime by default has the ability to dynamically load a custom filter.

It adds a JAR with your filter to the directory specified by the hbase.dynamic.jars.dir property (which defaults to the lib/ directory under the HBase root directory).

To disable automatic loading of dynamic JARs, set hbase.use.dynamic.jars to false in the advanced configuration snippet for hbase-site.xml if you use Cloudera Manager, or to hbase-site.xmlotherwise.

Logical operators, comparison operators and comparators

Filters can be combined together with logical operators.

Some filters take a combination of comparison operators and comparators. Following is the list of each.

Logical Operators

- AND - the key-value must pass both the filters to be included in the results.
- OR - the key-value must pass at least one of the filters to be included in the results.
- SKIP - for a particular row, if any of the key-values do not pass the filter condition, the entire row is skipped.
- WHILE - For a particular row, it continues to emit key-values until a key-value is reached that fails the filter condition.
- Compound Filters - Using these operators, a hierarchy of filters can be created. For example:

```
(Filter1 AND Filter2)OR(Filter3 AND Filter4)
```

Comparison Operators

- LESS (<)
- LESS_OR_EQUAL (<=)
- EQUAL (=)
- NOT_EQUAL (!=)
- GREATER_OR_EQUAL (>=)
- GREATER (>)
- NO_OP (no operation)

Comparators

- BinaryComparator - lexicographically compares against the specified byte array using the Bytes.compareTo(byte [], byte[]) method.
- BinaryPrefixComparator - lexicographically compares against a specified byte array. It only compares up to the length of this byte array.
- RegexStringComparator - compares against the specified byte array using the given regular expression. Only EQUAL and NOT_EQUAL comparisons are valid with this comparator.
- SubStringComparator - tests whether or not the given substring appears in a specified byte array. The comparison is case insensitive. Only EQUAL and NOT_EQUAL comparisons are valid with this comparator.

Examples

```
Example1: >, 'binary:abc' will match everything that is lexicographically gr
eater than "abc"
   Example2: =, 'binaryprefix:abc' will match everything whose first 3 char
acters are lexicographically equal to "abc"
   Example3: !=, 'regexstring:ab*yz' will match everything that doesn't begi
n with "ab" and ends with "yz"
   Example4: =, 'substring:abc123' will match everything that begins with
 the substring "abc123"
```

Compound operators

Within an expression, parentheses can be used to group clauses together, and parentheses have the highest order of precedence.

SKIP and WHILE operators are next, and have the same precedence.

The AND operator is next.

The OR operator is next.

Examples

```
A filter string of the form: "Filter1 AND Filter2 OR Filter3" will be evalua
ted as: "(Filter1 AND Filter2) OR Filter3"
   A filter string of the form: "Filter1 AND SKIP Filter2 OR Filter3" will
 be evaluated as: "(Filter1 AND (SKIP Filter2)) OR Filter3"
```

Filter types

HBase includes several filter types, as well as the ability to group filters together and create your own custom filters.

- KeyOnlyFilter - takes no arguments. Returns the key portion of each key-value pair.

```
Syntax: KeyOnlyFilter ()
```

- FirstKeyOnlyFilter - takes no arguments. Returns the key portion of the first key-value pair.

```
Syntax: FirstKeyOnlyFilter ()
```

- PrefixFilter - takes a single argument, a prefix of a row key. It returns only those key-values present in a row that start with the specified row prefix

```
Syntax:  PrefixFilter ('<row_prefix>')

      Example: PrefixFilter ('Row')
```

- ColumnPrefixFilter - takes a single argument, a column prefix. It returns only those key-values present in a column that starts with the specified column prefix.

```
Syntax:  ColumnPrefixFilter ('<column_prefix>')

      Example: ColumnPrefixFilter ('Col')
```

- MultipleColumnPrefixFilter - takes a list of column prefixes. It returns key-values that are present in a column that starts with any of the specified column prefixes.

```
Syntax:  MultipleColumnPrefixFilter ('<column_prefix>', '<column_prefix>',
 …, '<column_prefix>')

      Example: MultipleColumnPrefixFilter ('Col1', 'Col2')
```

- ColumnCountGetFilter - takes one argument, a limit. It returns the first limit number of columns in the table.

```
Syntax:  ColumnCountGetFilter ('<limit>')

      Example: ColumnCountGetFilter (4)
```

- PageFilter - takes one argument, a page size. It returns page size number of rows from the table.

```
Syntax:  PageFilter ('<page_size>')

      Example: PageFilter (2)
```

- ColumnPaginationFilter - takes two arguments, a limit and offset. It returns limit number of columns after offset number of columns. It does this for all the rows.

```
Syntax:  ColumnPaginationFilter ('<limit>', '<offset>')

      Example: ColumnPaginationFilter (3, 5)
```

- InclusiveStopFilter - takes one argument, a row key on which to stop scanning. It returns all key-values present in rows up to and including the specified row.

```
Syntax:  InclusiveStopFilter ('<stop_row_key>')

      Example: InclusiveStopFilter ('Row2')
```

- TimeStampsFilter - takes a list of timestamps. It returns those key-values whose timestamps matches any of the specified timestamps.

```
Syntax:  TimeStampsFilter (<timestamp>, <timestamp>, ... ,<timestamp>)

      Example: TimeStampsFilter (5985489, 48895495, 58489845945)
```

- RowFilter - takes a compare operator and a comparator. It compares each row key with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in that row.

```
Syntax:  RowFilter (<compareOp>, '<row_comparator>')

      Example: RowFilter (<=, 'binary:xyz)
```

- FamilyFilter - takes a compare operator and a comparator. It compares each family name with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in that family.

```
Syntax:  FamilyFilter (<compareOp>, '<family_comparator>')

      Example: FamilyFilter (>=, 'binaryprefix:FamilyB')
```

- QualifierFilter - takes a compare operator and a comparator. It compares each qualifier name with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in that column.

```
Syntax:  QualifierFilter (<compareOp>, '<qualifier_comparator>')

      Example: QualifierFilter (=, 'substring:Column1')
```

- ValueFilter - takes a compare operator and a comparator. It compares each value with the comparator using the compare operator and if the comparison returns true, it returns that key-value.

```
Syntax:  ValueFilter (<compareOp>, '<value_comparator>')

      Example: ValueFilter (!=, 'binary:Value')
```

- DependentColumnFilter - takes two arguments required arguments, a family and a qualifier. It tries to locate this column in each row and returns all key-values in that row that have the same timestamp. If the row does not contain the specified column, none of the key-values in that row will be returned.

  The filter can also take an optional boolean argument, dropDependentColumn. If set to true, the column used for the filter does not get returned.

  The filter can also take two more additional optional arguments, a compare operator and a value comparator, which are further checks in addition to the family and qualifier. If the dependent column is found, its value should also pass the value check. If it does pass the value check, only then is its timestamp taken into consideration.

```
Syntax:  DependentColumnFilter (‘<family>’, ‘<qualifier>’, <boolean>, <c
ompare operator>, ‘<value comparator’)
        DependentColumnFilter (‘<family>’, ‘<qualifier>’, <boolean>)
        DependentColumnFilter (‘<family>’, ‘<qualifier>’)

        Example: DependentColumnFilter (‘conf’, ‘blacklist’, false, >=, ‘
zebra’)
        DependentColumnFilter (‘conf’, ‘blacklist’, true)
        DependentColumnFilter (‘conf’, ‘blacklist’)
```

- SingleColumnValueFilter - takes a column family, a qualifier, a compare operator and a comparator. If the specified column is not found, all the columns of that row will be emitted. If the column is found and the comparison with the comparator returns true, all the columns of the row will be emitted. If the condition fails, the row will not be emitted.

  This filter also takes two additional optional boolean arguments, filterIfColumnMissing and setLatestVersion Only.

  If the filterIfColumnMissing flag is set to true, the columns of the row will not be emitted if the specified column to check is not found in the row. The default value is false.

  If the setLatestVersionOnly flag is set to false, it will test previous versions (timestamps) in addition to the most recent. The default value is true.

  These flags are optional and dependent on each other. You must set neither or both of them together.

```
Syntax:  SingleColumnValueFilter (‘<family>’, ‘<qualifier>’, <compare op
erator>, ‘<comparator>’, <filterIfColumnMissing_boolean>, <latest_versio
n_boolean>)
          Syntax:  SingleColumnValueFilter (‘<family>’, ‘<qualifier>’, <
compare operator>, ‘<comparator>’)

          Example: SingleColumnValueFilter (‘FamilyA’, ‘Column1’, <=, ‘abc
’, true, false)
          Example: SingleColumnValueFilter ('FamilyA’, ‘Column1’, <=, ‘a
bc’)
```

- SingleColumnValueExcludeFilter - takes the same arguments and behaves same as SingleColumnValueFilter. However, if the column is found and the condition passes, all the columns of the row will be emitted except for the tested column value.

```
Syntax:  SingleColumnValueExcludeFilter (<family>, <qualifier>, <compare
 operators>, <comparator>, <latest_version_boolean>, <filterIfColumnMiss
ing_boolean>)
      Syntax:  SingleColumnValueExcludeFilter (<family>, <qualifier>, <com
pare operator> <comparator>)

      Example: SingleColumnValueExcludeFilter (‘FamilyA’, ‘Column1’, ‘<=’,
 ‘abc’, ‘false’, ‘true’)
      Example: SingleColumnValueExcludeFilter (‘FamilyA’, ‘Column1’, ‘<=’,
 ‘abc’)
```

- ColumnRangeFilter - takes either minColumn, maxColumn, or both. Returns only those keys with columns that are between minColumn and maxColumn. It also takes two boolean variables to indicate whether to include the minColumn and maxColumn or not. If you don’t want to set the minColumn or the maxColumn, you can pass in an empty argument.

```
Syntax:  ColumnRangeFilter (‘<minColumn >’, <minColumnInclusive_bool>, ‘
<maxColumn>’, <maxColumnInclusive_bool>)

      Example: ColumnRangeFilter (‘abc’, true, ‘xyz’, false)
```

- Custom Filter - You can create a custom filter by implementing the Filter class. The JAR must be available on all RegionServers.

HBase Shell example

This example scans the 'users' table for rows where the contents of the cf:name column equals the string 'abc'.

```
hbase>
          scan 'users', { FILTER =>
 SingleColumnValueFilter.new(Bytes.toBytes('cf'),
          Bytes.toBytes('name'), CompareFilter::CompareOp.valueOf('EQUAL'),
          BinaryComparator.new(Bytes.toBytes('abc')))}
```

Java API example

This example shows how to use the Java API to implement several different filters.

This example is taken from the HBase unit test found in hbase-server/src/test/java/org/apache/hadoop/hbase/filter/TestSingleColumnValueFilter.java.

```
/**
      *
      * Licensed to the Apache Software Foundation (ASF) under one
      * or more contributor license agreements.  See the NOTICE file
      * distributed with this work for additional information
      * regarding copyright ownership.  The ASF licenses this file
      * to you under the Apache License, Version 2.0 (the
      * "License"); you may not use this file except in compliance
      * with the License.  You may obtain a copy of the License at
      *
      *     http://www.apache.org/licenses/LICENSE-2.0
      *
      * Unless required by applicable law or agreed to in writing, software
      * distributed under the License is distributed on an "AS IS" BASIS,
      * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or im
plied.
      * See the License for the specific language governing permissions and
      * limitations under the License.
      */
      package org.apache.hadoop.hbase.filter;

      import static org.junit.Assert.assertFalse;
      import static org.junit.Assert.assertTrue;

      import java.util.regex.Pattern;

      import org.apache.hadoop.hbase.KeyValue;
      import org.apache.hadoop.hbase.SmallTests;
```

```
import org.apache.hadoop.hbase.filter.CompareFilter.CompareOp;
import org.apache.hadoop.hbase.util.Bytes;
import org.junit.Before;
import org.junit.Test;
import org.junit.experimental.categories.Category;

/**
* Tests the value filter
*/
@Category(SmallTests.class)
public class TestSingleColumnValueFilter {
private static final byte[] ROW = Bytes.toBytes("test");
private static final byte[] COLUMN_FAMILY = Bytes.toBytes("test");
private static final byte [] COLUMN_QUALIFIER = Bytes.toBytes("foo");
private static final byte[] VAL_1 = Bytes.toBytes("a");
private static final byte[] VAL_2 = Bytes.toBytes("ab");
private static final byte[] VAL_3 = Bytes.toBytes("abc");
private static final byte[] VAL_4 = Bytes.toBytes("abcd");
private static final byte[] FULLSTRING_1 =
Bytes.toBytes("The quick brown fox jumps over the lazy dog.");
private static final byte[] FULLSTRING_2 =
Bytes.toBytes("The slow grey fox trips over the lazy dog.");
private static final String QUICK_SUBSTR = "quick";
private static final String QUICK_REGEX = ".+quick.+";
private static final Pattern QUICK_PATTERN = Pattern.compile("QuIcK",
 Pattern.CASE_INSENSITIVE | Pattern.DOTALL);

Filter basicFilter;
Filter nullFilter;
Filter substrFilter;
Filter regexFilter;
Filter regexPatternFilter;

@Before
public void setUp() throws Exception {
basicFilter = basicFilterNew();
nullFilter = nullFilterNew();
substrFilter = substrFilterNew();
regexFilter = regexFilterNew();
regexPatternFilter = regexFilterNew(QUICK_PATTERN);
}

private Filter basicFilterNew() {
return new SingleColumnValueFilter(COLUMN_FAMILY, COLUMN_QUALIFIER,
CompareOp.GREATER_OR_EQUAL, VAL_2);
}

private Filter nullFilterNew() {
return new SingleColumnValueFilter(COLUMN_FAMILY, COLUMN_QUALIFIER,
CompareOp.NOT_EQUAL,
new NullComparator());
}

private Filter substrFilterNew() {
return new SingleColumnValueFilter(COLUMN_FAMILY, COLUMN_QUALIFIER,
CompareOp.EQUAL,
new SubstringComparator(QUICK_SUBSTR));
}

private Filter regexFilterNew() {
return new SingleColumnValueFilter(COLUMN_FAMILY, COLUMN_QUALIFIER,
CompareOp.EQUAL,
new RegexStringComparator(QUICK_REGEX));
}
```

```
      private Filter regexFilterNew(Pattern pattern) {
      return new SingleColumnValueFilter(COLUMN_FAMILY, COLUMN_QUALIFIER,
      CompareOp.EQUAL,
      new RegexStringComparator(pattern.pattern(), pattern.flags()));
      }

      private void basicFilterTests(SingleColumnValueFilter filter)
      throws Exception {
      KeyValue kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_
2);
      assertTrue("basicFilter1", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_3);
      assertTrue("basicFilter2", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_4);
      assertTrue("basicFilter3", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
      assertFalse("basicFilterNotNull", filter.filterRow());
      filter.reset();
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_1);
      assertTrue("basicFilter4", filter.filterKeyValue(kv) == Filter.Retu
rnCode.NEXT_ROW);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_2);
      assertTrue("basicFilter4", filter.filterKeyValue(kv) == Filter.ReturnC
ode.NEXT_ROW);
      assertFalse("basicFilterAllRemaining", filter.filterAllRemaining());
      assertTrue("basicFilterNotNull", filter.filterRow());
      filter.reset();
      filter.setLatestVersionOnly(false);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_1);
      assertTrue("basicFilter5", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, VAL_2);
      assertTrue("basicFilter5", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
      assertFalse("basicFilterNotNull", filter.filterRow());
      }

      private void nullFilterTests(Filter filter) throws Exception {
      ((SingleColumnValueFilter) filter).setFilterIfMissing(true);
      KeyValue kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER, FUL
LSTRING_1);
      assertTrue("null1", filter.filterKeyValue(kv) == Filter.ReturnCode.INC
LUDE);
      assertFalse("null1FilterRow", filter.filterRow());
      filter.reset();
      kv = new KeyValue(ROW, COLUMN_FAMILY, Bytes.toBytes("qual2"), FULLST
RING_2);
      assertTrue("null2", filter.filterKeyValue(kv) == Filter.ReturnCode.IN
CLUDE);
      assertTrue("null2FilterRow", filter.filterRow());
      }

      private void substrFilterTests(Filter filter)
      throws Exception {
      KeyValue kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER,
      FULLSTRING_1);
      assertTrue("substrTrue",
      filter.filterKeyValue(kv) == Filter.ReturnCode.INCLUDE);
      kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER,
      FULLSTRING_2);
```

```
        assertTrue("substrFalse", filter.filterKeyValue(kv) == Filter.ReturnC
ode.INCLUDE);
        assertFalse("substrFilterAllRemaining", filter.filterAllRemaining());
        assertFalse("substrFilterNotNull", filter.filterRow());
        }

        private void regexFilterTests(Filter filter)
        throws Exception {
        KeyValue kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER,
        FULLSTRING_1);
        assertTrue("regexTrue",
        filter.filterKeyValue(kv) == Filter.ReturnCode.INCLUDE);
        kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER,
        FULLSTRING_2);
        assertTrue("regexFalse", filter.filterKeyValue(kv) == Filter.Return
Code.INCLUDE);
        assertFalse("regexFilterAllRemaining", filter.filterAllRemaining());
        assertFalse("regexFilterNotNull", filter.filterRow());
        }

        private void regexPatternFilterTests(Filter filter)
        throws Exception {
        KeyValue kv = new KeyValue(ROW, COLUMN_FAMILY, COLUMN_QUALIFIER,
        FULLSTRING_1);
        assertTrue("regexTrue",
        filter.filterKeyValue(kv) == Filter.ReturnCode.INCLUDE);
        assertFalse("regexFilterAllRemaining", filter.filterAllRemaining());
        assertFalse("regexFilterNotNull", filter.filterRow());
        }

        private Filter serializationTest(Filter filter)
        throws Exception {
        // Decompose filter to bytes.
        byte[] buffer = filter.toByteArray();

        // Recompose filter.
        Filter newFilter = SingleColumnValueFilter.parseFrom(buffer);
        return newFilter;
        }

        /**
        * Tests identification of the stop row
        * @throws Exception
        */
        @Test
        public void testStop() throws Exception {
        basicFilterTests((SingleColumnValueFilter) basicFilter);
        nullFilterTests(nullFilter);
        substrFilterTests(substrFilter);
        regexFilterTests(regexFilter);
        regexPatternFilterTests(regexPatternFilter);
        }

        /**
        * Tests serialization
        * @throws Exception
        */
        @Test
        public void testSerialization() throws Exception {
        Filter newFilter = serializationTest(basicFilter);
        basicFilterTests((SingleColumnValueFilter)newFilter);
        newFilter = serializationTest(nullFilter);
        nullFilterTests(newFilter);
        newFilter = serializationTest(substrFilter);
```

```
        substrFilterTests(newFilter);
        newFilter = serializationTest(regexFilter);
        regexFilterTests(newFilter);
        newFilter = serializationTest(regexPatternFilter);
        regexPatternFilterTests(newFilter);
        }

        }
```

HBase online merge

Cloudera Runtime supports online merging of regions.

HBase splits big regions automatically but does not support merging small regions automatically. To complete an online merge of two regions of a table, use the HBase shell to issue the online merge command. By default, both regions to be merged should be neighbors; that is, one end key of a region should be the start key of the other region. Although you can "force merge" any two regions of the same table, this can create overlaps and is not recommended.

The Master and RegionServer both participate in online merges. When the request to merge is sent to the Master, the Master moves the regions to be merged to the same RegionServer, usually the one where the region with the higher load resides. The Master then requests the RegionServer to merge the two regions. The RegionServer processes this request locally. Once the two regions are merged, the new region will be online and available for server requests, and the old regions are taken offline.

For merging two consecutive regions use the following command:

```
hbase> merge_region 'ENCODED_REGIONNAME', 'ENCODED_REGIONNAME'
```

For merging regions that are not adjacent, passing true as the third parameter forces the merge.

```
hbase> merge_region 'ENCODED_REGIONNAME', 'ENCODED_REGIONNAME', true
```

Note: This command is slightly different from other region operations. You must pass the encoded region name (ENCODED_REGIONNAME), not the full region name . The encoded region name is the hash suffix on region names. For example, if the region name is TestTable,0094429456,1289497600452.527db22f95c8a9e0116f0cc13c680396, the encoded region name portion is 527db22f95c8a9e0116f0cc13c680396.

Move HBase Master Role to another host

You can move the HBase Master Role from one host to another using Cloudera Manager.

Procedure

1. In Cloudera Manager, select the HBase service.
2. Click the Instances tab.
3. Click Add Role Instances and add role instances to HBase.
4. Choose the new host under Master.
5. Click Continue.
6. Start the newly added HBase Master role.
   The state of the role becomes Started.
7. Wait until the type of the newly added HBase Master role bacomes Master (Backup).

8. Stop any other non-active HBase Master role instances.

   This step does not impact HBase. It is required to ensure that the newly created HBase Master role backup will be chosen to be the new active HBase Master role.
9. Stop the remaining active HBase Master role.
   The type of the newly added HBase Master role automatically bacomes Master (Active).
10. Delete the old HBase Master role instances on hosts that are not wanted.

Expose HBase metrics to a Ganglia server

You can expose HBase metrics to Ganglia instance so that Ganglia, an open-source monitoring framework, can detect potential problems with your HBase cluster.

Procedure

1. In Cloudera Manager, select the HBase service.
2. Click the Configuration tab.
3. Select the HBase Master or RegionServer role.

Configure each role to monitor both:

4. Search for metrics2.
5. Find the Hadoop Metrics2 Advanced Configuration Snippet (Safety Valve) property.
6. Add the following snippet to the property, substituting the server information with your own:

```
hbase.sink.ganglia.class=org.apache.hadoop.metrics2.sink.ganglia.Ganglia
Sink31
hbase.sink.ganglia.servers=<Ganglia server>:<port>
hbase.sink.ganglia.period=10
```

7. To apply this configuration property to other role groups as needed, edit the value for the appropriate role group.
8. Click Save Changes.
9. Restart the role.
10. Restart the HBase service.