Configuring Fault Tolerance

Date of Publish: 2019-12-17

https://docs.hortonworks.com

Contents

Configuring Fault Tolerance

You can configure fault tolerance in Hortonworks Data Platform using Ambari and also on non-Ambari clusters.

You can configure fault tolerance in Hortonworks Data Platform in the following ways:

- Configure high availability using Apache Ambari - Use the Ambari's wizard-driven interface to configure high availability of the components.
- Configure high availability on non-Ambari clusters - You can configure HA on Hive Metastore, HiveServer2 instances, HBase, and NameNodes on HDFS. You can also configure HA for Resource Manager on YARN and Apache Ranger.
- Protect HDFS data and metadata - You can configure accidental deletion of files and use HDFS snapshots for data protection.

Apache Ambari High Availability

Ambari web provides a wizard-driven user experience that enables you to configure high availability of the components in many Hortonworks Data Platform (HDP) stack services. High availability is assured through establishing primary and secondary components. In the event that the primary component fails or becomes unavailable, the secondary component is available. After configuring high availability for a service, Ambari enables you to manage and disable (roll back) high availability of components in that service.

See the Ambari documentation for more information about configuring High Availability using Ambari.

High Availability on Non-Ambari Clusters

Configuring High Availability (HA) on your cluster is essential for avoiding instances of service failures or cluster downtimes.

Configuring High Availability for the Hive Metastore

Configuring and deploying the Hive Metastore service in High Availability mode can address instances of service failure.

Important:

The relational database that backs the Hive Metastore itself should also be made highly available using best practices defined for the database system in use.

Use Cases and Failover Scenarios

Deploying the Metastore service in High Availability (HA) mode can help in handling service failures. You should deploy the Metastore service on multiple systems concurrently.

Use Cases

The Metastore HA solution is designed to handle Metastore service failures. Whenever a deployed Metastore service goes down, Metastore service can remain unavailable for a considerable time until service is brought back up. To avoid such outages, deploy the Metastore service in HA mode.

Deployment Scenarios

Hortonworks recommends deploying the Metastore service on multiple systems concurrently. Each Hive Metastore client will read the configuration property hive.metastore.uris to get a list of Metastore servers with which it can try to communicate.

```
<property>
 <name> hive.metastore.uris </name>
 <value> thrift://$Hive_Metastore_Server_Host_Machine_FQDN </value>
 <description> A comma separated list of Metastore uris on which Metastore
 service is running </description>
</property>
```

Note that the relational database that backs the Hive Metastore itself should also be made highly available using the best practices defined for the database system in use.

In the case of a secure cluster, add the following configuration property to the hive-site.xml file for each Metastore server:

```
<property>
 <name> hive.cluster.delegation.token.store.class</name>
 <value>org.apache.hadoop.hive.thrift.ZooKeeperTokenStore</value>
</property>
```

Failover Scenario

The Hive metastore client randomly chooses a metastore URI when multiple metastores are configured, which helps in load-balancing.

Software Configuration

To configure high availability for Hive, you must install HDP, update the Hive Metastore, and validate the Hive configuration.

About this task

Complete the following tasks to configure Hive High Availability solution:

Procedure

1. Install Hortonworks Data Platform.
2. Update the Hive Metastore.
3. Validate the configuration.

Install Hortonworks Data Platform

Install Hortonworks Data Platform on your cluster hardware and later configure high availability.

Procedure

1. Make sure that you specify the virtual machine as your NameNode.
2. Download the Apache Ambari repository.

   Note:

   Do not start the Ambari server until you have configured the relevant templates as outlined in the following steps.
3. Edit the <master-install-machine-for-Hive-Metastore>/etc/hive/conf.server/hive-site.xml configuration file to add the following properties:

   - Provide the URI for the client to contact Metastore server. The following property can have a comma separated list when your cluster has multiple Hive Metastore servers.

     ```
     <property>
     ```

```
 <name>hive.metastore.uris</name>
 <value>thrift://$Hive_Metastore_Server_Host_Machine_FQDN</value>
 <description>URI for client to contact Metastore server</description>
</property>
```

- Configure Hive cluster delegation token storage class.

```
<property>
 <name>hive.cluster.delegation.token.store.class</name>
 <value>org.apache.hadoop.hive.thrift.ZooKeeperTokenStore</value>
</property>
```

- Complete Hortonworks Data Platform installation.

  See the Ambari documentation for installing, configuring, and deploying the HDP cluster.

Update the Hive Metastore

Hortonworks Data Platform components configured for High Availability must use a NameService rather than a NameNode.

About this task

Use the following instructions to update the Hive Metastore to reference the NameService rather than a NameNode.

Note: Hadoop administrators also often use the following procedure to update the Hive Metastore with the new URI for a node in a Hadoop cluster. For example, administrators sometimes rename an existing node as their cluster grows.

Procedure

1. Open a command prompt on the machine hosting the Hive Metastore.
2. Set the HIVE_CONF_DIR environment variable:

```
export HIVE_CONF_DIR=/etc/hive/conf.server
```

3. Execute the following command to retrieve a list of URIs for the filesystem roots, including the location of the NameService:

```
hive --service metatool -listFSRoot
```

4. Execute the following command with the -dryRun option to test your configuration change before implementing it:

```
hive --service metatool -updateLocation <nameservice-uri> <namenode-uri> -
 dryRun
```

5. Execute the command again, this time without the -dryRun option:

```
hive --service metatool -updateLocation <nameservice-uri> <namenode-uri>
```

Validate configuration

You must test various failover scenarios to validate your configuration.

Deploying Multiple HiveServer2 Instances for High Availability

Deploy and configure a second instance of HiveServer2 (HS2) that runs in parallel with your primary instance of HS2 to enhance availability.

Adding an Additional HiveServer2 to Your Cluster Manually

You must add the additional HiveServer2 to your cluster manually.

Procedure

1. Install Hive on the new node. For example, use one of the following commands in RHEL/CentOS/Oracle Linux environments:
   a) If the new node is part of a cluster where Hadoop and HDFS have not been installed, use the following command:

   ```
   yum install hive-hcatalog hadoop hadoop-hdfs hadoop-libhdfs hadoop-yarn
    hadoop-mapreduce hadoop-client openssl
   ```

   b) If the new node is part of a cluster where Hadoop and HDFS are installed, you need only install the hive-hcatalog package. For example, in RHEL/CentOS/Oracle Linux environments use the following command:
2. Copy the following configuration files from your existing HS2 instance to the new HS2 instance:
   a) Under /etc/hive/conf, copy the hive-site.xml file.

   For HDP version 2.2 and later, you must also copy the hiveserver2-site.xml file.
   b) Under /etc/hadoop/conf, copy the core-site.xml, hdfs-site.xml, mapred-site.xml, and yarn-site.xml files.
3. Copy the database driver file for the Metastore database from the /usr/hdp/current/hive-server2/lib folder of the existing HS2 instance to the /usr/hdp/current/hive-server2/lib folder of the new instance. For example, postgresql-jdbc.jar is the database driver file for a PostgreSQL database, and mysql-connector-java.jar is the database driver file for a MySQL database.

   Note:

   Before HDP version 2.2.0, the database driver file for the Metastore database is located in the /usr/lib folder.
4. Start the HS2 service:

   ```
   su $HIVE_USER
   /usr/lib/hive/bin/hiveserver2 -hiveconf hive.metastore.uris=" "
   -hiveconf hive.log.file=hiveserver2.log
   >$HIVE_LOG_DIR/hiveserver2.out 2
   >$HIVE_LOG_DIR/hiveserver2.log &
   ```

   If you are using HDP 2.1.15 and earlier, the HS2 service startup script is located in the /usr/lib/hive/bin directory. For more information about starting the HS2 service, see the

   a) If you are using HDP 2.1.15 and earlier, the HS2 service startup script is located in the /usr/lib/hive/bin directory.
   b) Specifying –hiveconf hive.metastore.uris=" " when you start the HS2 service causes HS2 to use an embedded Metastore, which improves the performance when HS2 retrieves data from the back-end data store (RDBMS). If you are using HDP 2.3.0 or later and have added the hive.metastore.uris=" " property to the hiveserver2-site.xml file, it is not necessary to specify it on the command line when you start the service.
5. Validate your installation by connecting to the new HS2 instance using Beeline.
   a) Open Beeline command-line shell to interact with HS2:

   ```
   /usr/hdp/current/hive-server2/bin/beeline
   ```

   b) Establish a connection to HS2:

   ```
   !connect jdbc:hive2://$hive.server.full.hostname:<port_number>
    $HIVE_USER password org.apache.hive.jdbc.HiveDriver
   ```

c) Run sample commands

```
show databases;
create table test2(a int, b string);
show tables;
```

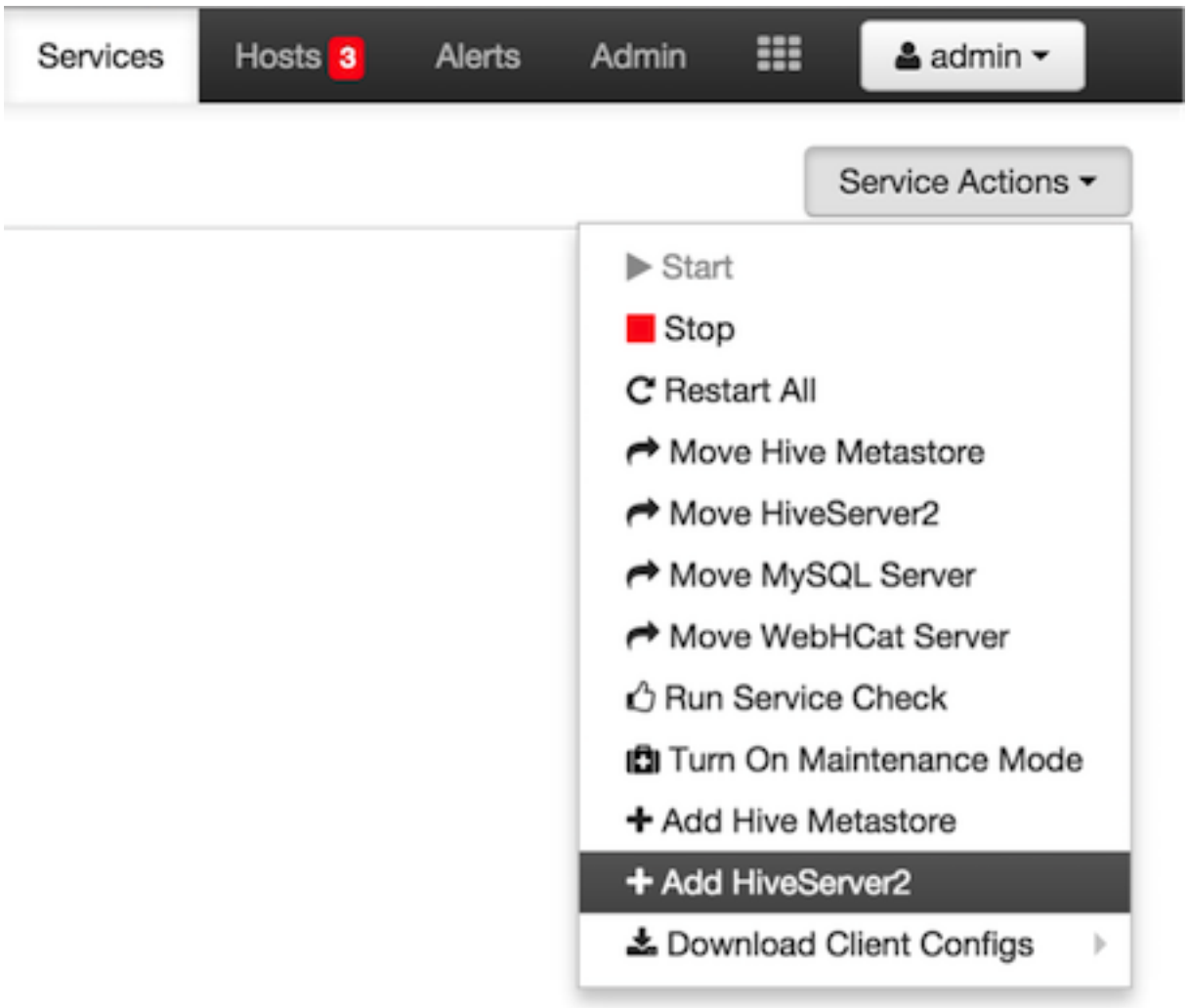
This completes the manual installation of an additional HiveServer2 on a cluster.

Adding an Additional HiveServer2 to a Cluster with Ambari

If you have a new HiveServer2 (HS2) instance installed on a new host that has not been added to your cluster, you can add it with Ambari.

Procedure

1. Open Ambari in a browser and click the Hosts tab.
2. On the Hosts page, click Actions, and select Add New Hosts.
3. Follow the Add Host Wizard instructions.
4. When you have completed adding the host to your cluster, click the Services tab.
5. On the Services page, click Hive in the list of services on the left side of the browser.
6. On the upper right side of the Hive Services page, click Service Actions, and select Add HiveServer2:



7. In the Confirmation dialog box, select the host that you added in Steps 1 -3, and click Confirm Add.

   The progress of adding the HS2 is displayed in the Background Operation Running dialog box. Click OK when it is finished to dismiss the dialog box.

Configuring HiveServer2 High Availability Using ZooKeeper

You can implement HiveServer2 High Availability using ZooKeeper.

This section describes how to implement HiveServer2 High Availability through ZooKeeper.

- How ZooKeeper Manages HiveServer2 Requests
- Dynamic Service Discovery Through ZooKeeper
- Rolling Upgrade for HiveServer2 Through ZooKeeper

How ZooKeeper Manages HiveServer2 Requests

Multiple HiveServer2 (HS2) instances can register themselves with ZooKeeper and then the client (client driver) can find a HS2 through ZooKeeper.

When a client requests an HS2 instance, ZooKeeper returns one randomly-selected registered HS2.

This enables the following scenarios:

- High Availability

  If more than one HS2 instance is registered with ZooKeeper, and all instances fail except one, ZooKeeper passes the link to the instance that is running and the client can connect successfully. (Failed instances must be restarted manually.)
- Load Balancing

  If there is more than one HS2 instance registered with ZooKeeper, ZooKeeper responds to client requests by randomly passing a link to one of the HS2 instances. This ensures that all HS2 instances get roughly the same workload.

The following situation is not handled:

- Automatic Failover

  If an HS2 instance failed while a client is connected, the session is lost. Since this situation need to be handed at the client, there is no automatic failover; the client needs to reconnect using ZooKeeper.

Dynamic Service Discovery Through ZooKeeper

You can implement dynamic service discovery by including an additional indirection step through ZooKeeper.

The HS2 instances register with ZooKeeper under a namespace. When a HiveServer2 instance comes up, it registers itself with ZooKeeper by adding a znode in ZooKeeper. The znode name has the format:

/<hiveserver2_namespace>/serverUri=<host:port>;version=<versionInfo>; sequence=<sequence_number>,

The znode stores the server host:port as its data.

The server instance sets a watch on the znode; when the znode is modified, that watch sends a notification to the server. This notification helps the server instance keep track of whether or not it is on the list of servers available for new client connections.

When a HiveServer2 instance is de-registered from ZooKeeper, it is removed from the list of servers available for new client connections. (Client sessions on the server are not affected.) When the last client session on a server is closed, the server is closed.

To de-register a single HiveServer2, enter hive --service hiveserver2 --deregister <package ID>

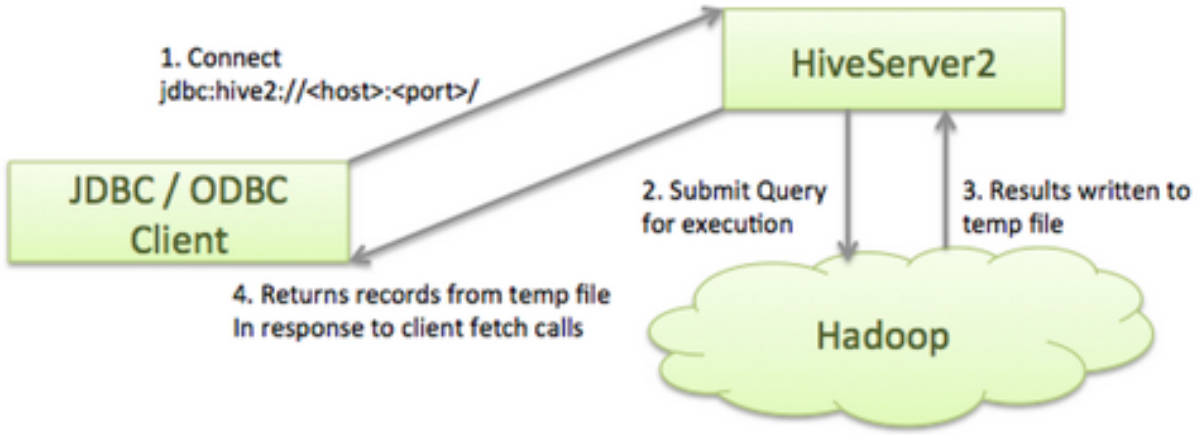
Query Execution Path Without ZooKeeper

As shown in the illustration below, query execution without ZooKeeper happens in the traditional client/server model used by most databases:

1. The JDBC / ODBC driver is given a host:port to an existing HS2 instance.

   This establishes a session where multiple queries can be executed.

   For each query...
2. Client submits a query to HS2 which in turn submits it for execution to Hadoop.
3. The results of query are written to a temporary file.
4. The client driver retrieves the records from HS2 which returns them from the temporary file.



Query Execution Path With ZooKeeper

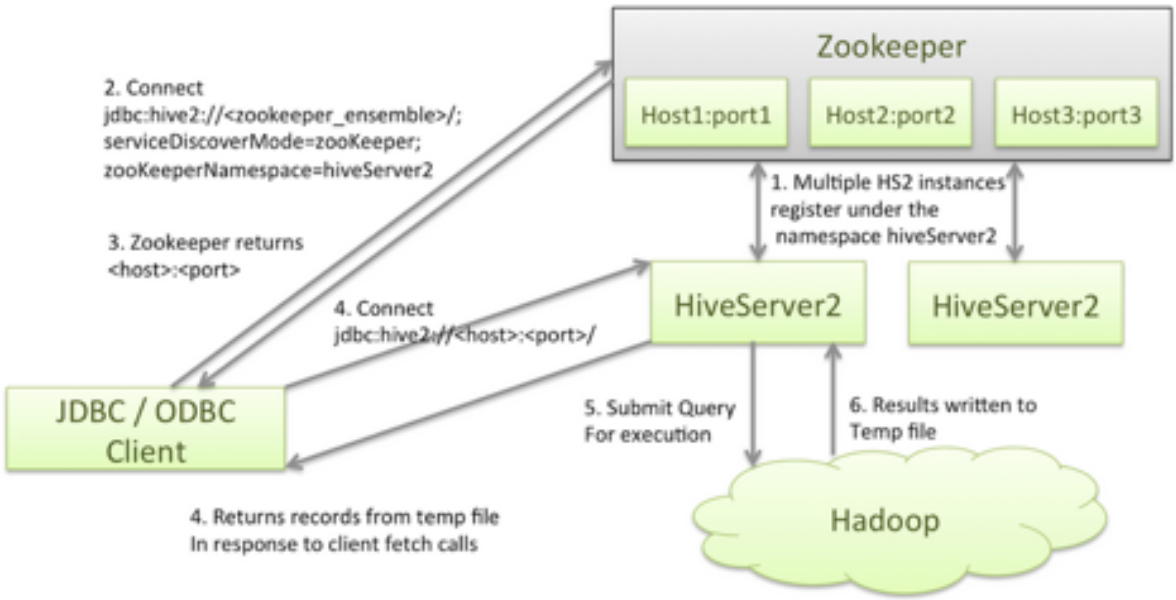
Query execution with ZooKeeper takes advantage of dynamic discovery. Thus, the client driver needs to know how to use this capability, which is available in HDP 2.2 and later with the JDBC driver and ODBC driver 2.0.0.

Dynamic discovery is implemented by including an additional indirection step through ZooKeeper. As shown in the figure below...

1. Multiple HiveServer2 instances are registered with ZooKeeper.
2. The client driver connects to the ZooKeeper ensemble:

   ```
   jdbc:hive2://<zookeeper_ensemble>;serviceDiscoveryMode=zooKeeper;
    zooKeeperNamespace=<hiveserver2_namespace
   ```

   In the figure below, <zookeeper_ensemble> is Host1:Port1, Host2:Port2, Host3:Port3; <hiveserver_namespace) is hiveServer2.
3. ZooKeeper randomly returns <host>:<port> for one of the registered HiveServer2 instances.
4. The client driver can not connect to the returned HiveServer instance and proceed as shown in the previous section (as if ZooKeeper was not present).



Rolling Upgrade for HiveServer2 Through ZooKeeper

There are additional configuration settings and procedures that need to be implemented to support rolling upgrade for HiveServer.

Set Configuration Parameters for HiveServer2 Rolling Upgrade

Set the Zookeeper Quorum Configuration parameters before performing rolling upgrade for HiveServer2.

Procedure

1. Set hive.zookeeper.quorum to the ZooKeeper ensemble (a comma separated list of ZooKeeper server host:ports running at the cluster)
2. Customize hive.zookeeper.session.timeout so that it closes the connection between the HiveServer2's client and ZooKeeper if a heartbeat is not received within the timeout period.
3. Set hive.server2.support.dynamic.service.discovery to true
4. Set hive.server2.zookeeper.namespace to the value that you want to use as the root namespace on ZooKeeper. The default value is hiveserver2.
5. The administrator should ensure that the ZooKeeper service is running on the cluster, and that each HiveServer2 instance gets a unique host:port combination to bind to upon startup.

Perform Rolling Upgrade for HiveServer2

After setting the required configuration parameters, perform the rolling upgrade for HiveServer2.

Procedure

1. Without altering the old version of HiveServer2, bring up instances of the new version of HiveServer2. Make sure they start up successfully.
2. To de-register instances of the old version of HiveServer2, enter hive service hiveserver2 deregister
3. Do not shut down the older instances of HiveServer2, as they might have active client sessions. When sessions complete and the last client connection is closed, the server instances shut down on their own. Eventually all instances of the older version of HiveServer2 will become inactive.

Perform Rollback of HiveServer2

Based on your requirements, you can roll back HiveServer2 upgrades.

Procedure

1. Bring up instances of the older version of HiveServer2. Make sure they start up successfully.
2. To explicitly de-register the instances of the newer version of HiveServer2, enter: hive service hiveserver2 deregister
3. Do not shut down the newer instances of HiveServer2, as they might have active client sessions. When sessions complete and the last client connection is closed, the server instances shut down on their own. Eventually all instances of the newer version of HiveServer2 will become inactive.

Configuring High Availability for HBase

HDP enables HBase administrators to configure HBase clusters with read-only High Availability, or HA.

This feature benefits HBase applications that require low-latency queries and can tolerate minimal (near-zero-second) staleness for read operations. Examples include queries on remote sensor data, distributed messaging, object stores, and user profile management.

High Availability for HBase features the following functionality:

- Data is safely protected in HDFS
- Failed nodes are automatically recovered

- No single point of failure
- All HBase API and region operations are supported, including scans, region split/merge, and META table support (the META table stores information about regions)

However, HBase administrators should carefully consider the following costs associated with using High Availability features:

- Double or triple MemStore usage
- Increased BlockCache usage
- Increased network traffic for log replication
- Extra backup RPCs for secondary region replicas

HBase is a distributed key-value store designed for fast table scans and read operations at petabyte scale. Before configuring HA for HBase, you should understand the concepts in the following table.

Table 1: Basic HBase Concepts

HBase Concept	Description
Region	A group of contiguous rows in an HBase table. Tables start with one region; additional regions are added dynamically as the table grows. Regions can be spread across multiple hosts to balance workloads and recover quickly from failure. There are two types of regions: primary and secondary. A secondary region is a copy of a primary region, replicated on a different RegionServer.
RegionServer	A 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. When region replication is enabled, a RegionServer can serve regions in primary and secondary mode concurrently.
Column family	A column family is a group of semantically related columns that are stored together.
Memstore	Memstore is 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)	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.
Compaction	When operations stored in the MemStore are flushed to disk, 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 hit the data limit (specified by the MemStore) at the same time, HBase performance may degrade from the large number of simultaneous major compactions. Administrators can avoid this by manually splitting tables over time.

Introduction to HBase High Availability

HBase provides a feature called region replication to achieve high availability for reads.

HBase, architecturally, has had a strong consistency guarantee from the start. All reads and writes are routed through a single RegionServer, which guarantees that all writes happen in order, and all reads access the most recently committed data.

However, because of this "single homing" of reads to a single location, if the server becomes unavailable, the regions of the table that are hosted in the RegionServer become unavailable for some time until they are recovered. There are three phases in the region recovery process: detection, assignment, and recovery. Of these, the detection phase is

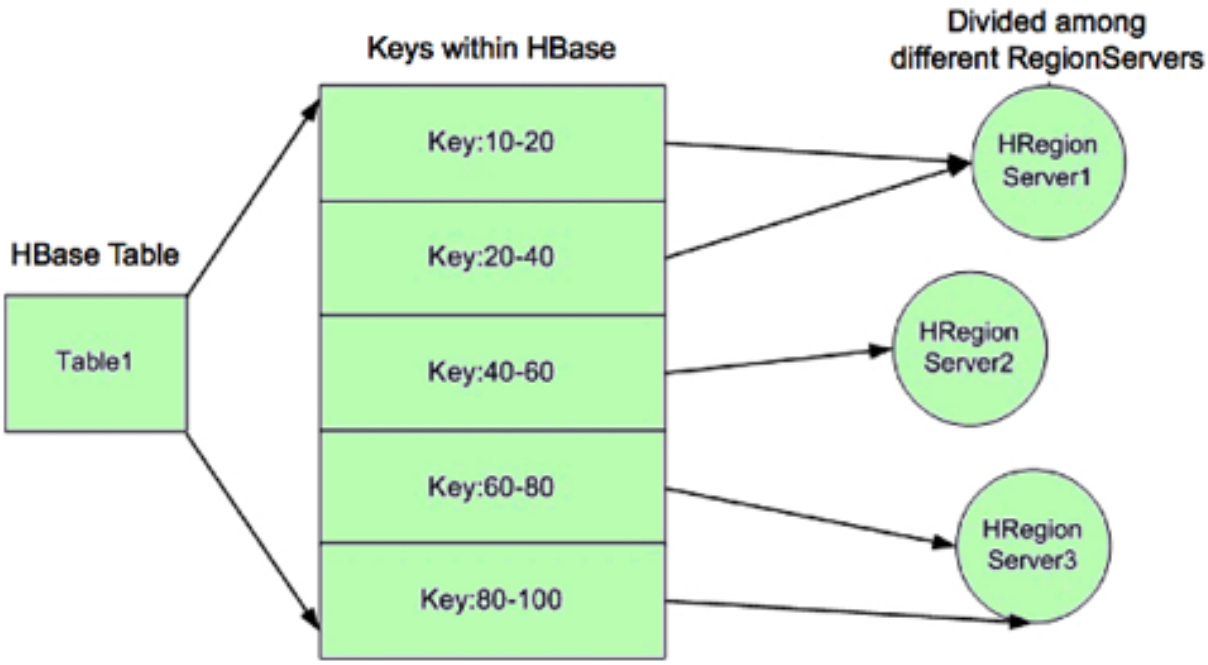
usually the longest, currently on the order of 20 to 30 seconds depending on the ZooKeeper session timeout setting (if the RegionServer became unavailable but the ZooKeeper session is alive). After that we recover data from the Write Ahead Log and assign the region to a different server. During this time -- until the recovery is complete -- clients are not able to read data from that region.

For some use cases the data may be read-only, or reading some amount of stale data is acceptable. With timeline-consistent highly available reads, HBase can be used for these kind of latency-sensitive use cases where the application can expect to have a time bound on the read completion.

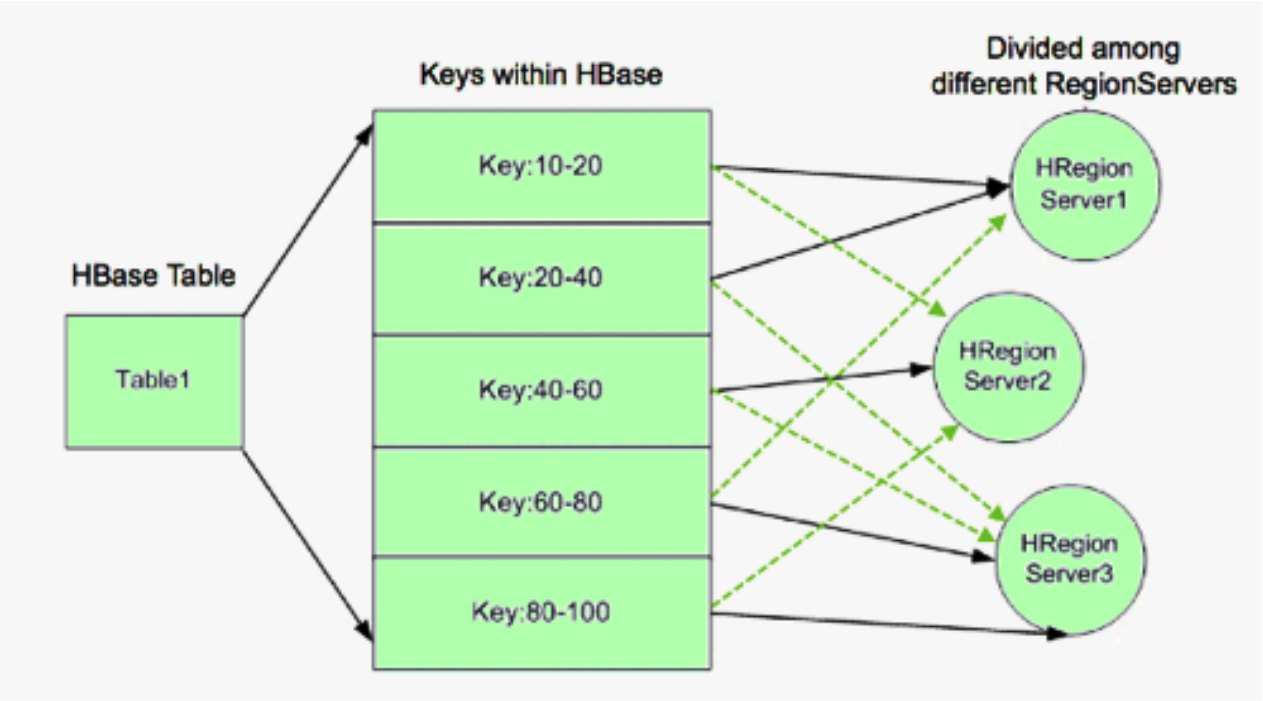
For achieving high availability for reads, HBase provides a feature called “region replication”. In this model, for each region of a table, there can be multiple replicas that are opened in different RegionServers. By default, the region replication is set to 1, so only a single region replica is deployed and there are no changes from the original model. If region replication is set to 2 or more, then the master assigns replicas of the regions of the table. The Load Balancer ensures that the region replicas are not co-hosted in the same Region Servers and also in the same rack (if possible).

All of the replicas for a single region have a unique replica ID, starting with 0. The region replica with replica ID = 0 is called the "primary region." The others are called “secondary region replicas,” or "secondaries". Only the primary region can accept writes from the client, and the primary always contains the latest changes. Since all writes must go through the primary region, the writes are not highly available (meaning they might be blocked for some time if the region becomes unavailable).

In the following image, for example, RegionServer 1 is responsible for responding to queries and scans for keys 10 through 40. If RegionServer 1 crashes, the region holding keys 10-40 is unavailable for a short time until the region recovers.



HA provides a way to access keys 10-40 even if RegionServer 1 is not available, by hosting replicas of the region and assigning the region replicas to other RegionServers as backups. In the following image, RegionServer 2 hosts secondary region replicas for keys 10-20, and RegionServer 3 hosts the secondary region replica for keys 20-40. RegionServer 2 also hosts the secondary region replica for keys 80-100. There are no separate RegionServer processes for secondary replicas. Rather, RegionServers can serve regions in primary or secondary mode. When RegionServer 2 services queries and scans for keys 10-20, it acts in secondary mode.



Note:

Regions acting in secondary mode are also known as Secondary Region Replicas. However, there is no separate RegionServer process. A region in secondary mode can read but cannot write data. In addition, the data it returns may be stale, as described in the following section.

Timeline and Strong Data Consistency

HBase guarantees timeline consistency for all data served from RegionServers in secondary mode, meaning all HBase clients see the same data in the same order, but that data may be slightly stale. Only the primary RegionServer is guaranteed to have the latest data. Timeline consistency simplifies the programming logic for complex HBase queries and provides lower latency than quorum-based consistency.

In contrast, strong data consistency means that the latest data is always served. However, strong data consistency can greatly increase latency in case of a RegionServer failure, because only the primary RegionServer is guaranteed to have the latest data. The HBase API allows application developers to specify which data consistency is required for a query.

Note:

The HBase API contains a method called Result.isStale(), which indicates whether data returned in secondary mode is "stale" -- the data has not been updated with the latest write operation to the primary RegionServer.

Propagating Writes to Region Replicas

StoreFile Refresher and Async WAL Replication are the mechanisms used to propagate writes from the primary replica to secondary replicas.

Writes are written only to the primary region replica.

The following two mechanisms are used to propagate writes from the primary replica to secondary replicas.

Note:

By default, HBase tables do not use High Availability features. After configuring your cluster for High Availability, designate tables as HA by setting region replication to a value greater than 1 at table creation time.

For read-only tables, you do not need to use any of the following methods. Disabling and enabling the table should make the data available in all region replicas.

StoreFile Refresher

The first mechanism is the store file refresher, which was introduced in Phase 1 (Apache HBase 1.0.0 and HDP 2.1).

Store file refresher is a thread per RegionServer, which runs periodically, and does a refresh operation for the store files of the primary region for the secondary region replicas. If enabled, the refresher ensures that the secondary region replicas see the new flushed, compacted or bulk loaded files from the primary region in a timely manner. However, this means that only flushed data can be read back from the secondary region replicas, and after the refresher is run, making the secondaries lag behind the primary for an a longer time.

To enable this feature, configure hbase.regionserver.storefile.refresh.period to a value greater than zero.

Async WAL Replication

The second mechanism for propagating writes to secondaries is done via the Async WAL Replication feature.

Async WAL replication works similarly to HBase's multi-datacenter replication, but the data from a region is replicated to its secondary regions. Each secondary replica always receives writes in the same order that the primary region committed them. In some sense, this design can be thought of as "in-cluster replication"; instead of replicating to a different datacenter, the data goes to secondary regions. This process keeps the secondary region's in-memory state up to date. Data files are shared between the primary region and the other replicas, so there is no extra storage overhead. However, secondary regions have recent non-flushed data in their MemStores, which increases memory overhead. The primary region writes flush, compaction, and bulk load events to its WAL as well, which are also replicated through WAL replication to secondaries. When secondary replicas detect a flush/compaction or bulk load event, they replay the event to pick up the new files and drop the old ones.

Committing writes in the same order as in the primary region ensures that the secondaries won't diverge from the primary region's data, but because the log replication is asynchronous, the data might still be stale in secondary regions. Because this feature works as a replication endpoint, performance and latency characteristics should be similar to inter-cluster replication.

Async WAL Replication is disabled by default. To enable this feature, set hbase.region.replica.replication.enabled to true.

When you create a table with High Availability enabled, the Async WAL Replication feature adds a new replication peer (named region_replica_replication).

Once enabled, to disable this feature you'll need to perform the following two steps:

- Set hbase.region.replica.replication.enabled to false in hbase-site.xml.
- In your cluster, disable the replication peer named region_replica_replication, using hbase shell or ReplicationAdmin class: hbase> disable_peer 'region_replica_replication'

Store File TTL

In phase 1 and 2 of the write propagation approaches mentioned above, store files for the primary replica are opened in secondaries independent of the primary region. Thus, for files that the primary region compacted and archived, the secondaries might still refer to these files for reading.

Both features use HFileLinks to refer to files, but there is no guarantee that the file is not deleted prematurely. To prevent I/O exceptions for requests to replicas, set the configuration property hbase.master.hfilecleaner.ttl to a sufficient time range such as 1 hour.

Region Replication for the META Table's Region

Currently, Async WAL Replication is not done for the META table's WAL -- the META table's secondary replicas still refresh themselves from the persistent store files. To ensure that the META store files are refreshed, set hbase.regionserver.meta.storefile.refresh.period to a non-zero value. This is configured differently than hbase.regionserver.storefile.refresh.period.

Related Tasks

Creating Highly Available HBase Tables with the HBase Java API

Creating Highly Available HBase Tables with the HBase Shell
Configuring HA Reads for HBase

Timeline Consistency

With timeline consistency, HBase introduces a consistency definition that can be provided per read operation.

The Consistency definition that can be provided per read operation (get or scan) is as follows:

```
public enum Consistency {
   STRONG,
   TIMELINE
 }
```

Consistency.STRONG is the default consistency model provided by HBase. If a table has region replication = 1, or has region replicas but the reads are done with time consistency enabled, the read is always performed by the primary regions. This preserves previous behavior; the client receives the latest data.

If a read is performed with Consistency.TIMELINE, then the read RPC is sent to the primary RegionServer first. After a short interval (hbase.client.primaryCallTimeout.get, 10ms by default), a parallel RPC for secondary region replicas is sent if the primary does not respond back. HBase returns the result from whichever RPC finishes first. If the response is from the primary region replica, the data is current. You can use Result.isStale() API to determine the state of the resulting data:

- If the result is from a primary region, Result.isStale() is set to false.
- If the result is from a secondary region, Result.isStale() is set to true.

TIMELINE consistency as implemented by HBase differs from pure eventual consistency in the following respects:

- Single homed and ordered updates: Whether region replication is enabled or not, on the write side, there is still only one defined replica (primary) that can accept writes. This replica is responsible for ordering the edits and preventing conflicts. This guarantees that two different writes are not committed at the same time by different replicas, resulting in divergent data. With this approach, there is no need to do read-repair or last-timestamp-wins types of of conflict resolution.
- The secondary replicas also apply edits in the order that the primary committed them, thus the secondaries contain a snapshot of the primary's data at any point in time. This is similar to RDBMS replications and HBase's own multi-datacenter replication, but in a single cluster.
- On the read side, the client can detect whether the read is coming from up-to-date data or is stale data. Also, the client can issue reads with different consistency requirements on a per-operation basis to ensure its own semantic guarantees.
- The client might still read stale data if it receives data from one secondary replica first, followed by reads from another secondary replica. There is no stickiness to region replicas, nor is there a transaction ID-based guarantee. If required, this can be implemented later.

Memory Accounting

Secondary region replicas refer to data files in the primary region replica, but they have their own MemStores (in HA Phase 2) and use block cache as well. However, one distinction is that secondary region replicas cannot flush data when there is memory pressure for their MemStores. They can only free up MemStore memory when the primary region does a flush and the flush is replicated to the secondary.

Because a RegionServer can host primary replicas for some regions and secondaries for others, secondary replicas might generate extra flushes to primary regions in the same host. In extreme situations, there might be no memory for new writes from the primary, via WAL replication.

To resolve this situation, the secondary replica is allowed to do a "store file refresh." A file system list operation picks up new files from the primary, possibly dropping its MemStore. This refresh is only performed if the MemStore size of the biggest secondary region replica is at least hbase.region.replica.storefile.refresh.memstore.multiplier times bigger than the biggest MemStore of a primary replica. (The default value for hbase.region.replica.storefile.refresh.memstore.multiplier is 4.)

Note:

If this operation is performed, the secondary replica might obtain partial row updates across column families (because column families are flushed independently). We recommend that you configure HBase to not do this operation frequently.

You can disable this feature by setting the value to a large number, but this might cause replication to be blocked without resolution.

Secondary Replica Failover

When a secondary region replica first comes online, or after a secondary region fails over, it may have contain edits from its MemStore. The secondary replica must ensure that it does accesss stale data (data that has been overwritten) before serving requests after assignment. Therefore, the secondary waits until it detects a full flush cycle (start flush, commit flush) or a “region open event” replicated from the primary.

Until the flush cycle occurs, the secondary region replica rejects all read requests via an IOException with the following message:

The region's reads are disabled

Other replicas are probably still be available to read, thus not causing any impact for the RPC with TIMELINE consistency.

To facilitate faster recovery, the secondary region triggers a flush request from the primary when it is opened. The configuration property hbase.region.replica.wait.for.primary.flush (enabled by default) can be used to disable this feature if needed.

Configuring HA Reads for HBase

To enable High Availability for HBase reads, specify the server-side and client-side configuration properties in your hbase-site.xml configuration file, and then restart the HBase Master and RegionServers.

Procedure

1. Set the server-side properties in your hbase-site.xml configuration file for all servers in your HBase cluster that use region replicas. The following table describes server-side properties.

Table 2: Server-Side Configuration Properties for HBase HA

Property	Example value	Description
hbase.regionserver. storefile.refresh.period	30000	Specifies the period (in milliseconds) for refreshing the store files for secondary regions. The default value is 0, which indicates that the feature is disabled. Secondary regions receive new files from the primary region after the secondary replica refreshes the list of files in the region. Note: Too-frequent refreshes might cause extra Namenode pressure. If files cannot be refreshed for longer than HFile TTL, specified with hbase.master.hfilecleaner.ttl, the requests are rejected. Refresh period should be a non-zero number if META replicas are enabled (see hbase.meta.replica.count). If you specify refresh period, we recommend configuring HFile TTL to a larger value than its default.

Property	Example value	Description
hbase.region.replica. replication.enabled	true	Determines whether asynchronous WAL replication is enabled or not. The value can be true or false. The default is false. If this property is enabled, a replication peer named region_replica_replication is created. The replication peer replicates changes to region replicas for any tables that have region replication set to 1 or more. After enabling this property, disabling it requires setting it to false and disabling the replication peer using the shell or the ReplicationAdmin java class. When replication is explicitly disabled and then re-enabled, you must set hbase.replication to true.
hbase.master. hfilecleaner.ttl	3600000	Specifies the period (in milliseconds) to keep store files in the archive folder before deleting them from the file system.
hbase.master. loadbalancer.class	org.apache.hadoop.hbase. master.balancer. StochasticLoadBalancer	Specifies the Java class used for balancing the load of all HBase clients. The default value is org.apache.hadoop.hbase. master.balancer. StochasticLoadBalancer, which is the only load balancer that supports reading data from RegionServers in secondary mode.
hbase.meta.replica.count	3	Region replication count for the meta regions. The default value is 1.
hbase.regionserver. meta.storefile.refresh.period	30000	Specifies the period in milliseconds for refreshing the store files for the HBase META tables secondary regions. If this is set to 0, the feature is disabled. When the secondary region refreshes the list of files in the region, the secondary regions see new files that are flushed and compacted from the primary region. There is no notification mechanism. Note: If the secondary region is refreshed too frequently, it may cause Namenode pressure. Requests are rejected if the files cannot be refreshed for longer than HFile TTL, which is specified with hbase.master.hfilecleaner.ttl. Configuring HFile TTL to a larger value is recommended with this setting. If META replicas are enabled, set this to a non-zero number by setting hbase.meta.replica.count to a value greater than 1.
hbase.region.replica.wait. for.primary.flush	true	Specifies whether to wait for a full flush cycle from the primary before starting to serve data in a secondary replica. Disabling this feature might cause secondary replicas to read stale data when a region is transitioning to another RegionServer.

Property	Example value	Description
hbase.region.replica. storefile.refresh. memstore.multiplier	4	Multiplier for a “store file refresh” operation for the secondary region replica. This multiplier is used to refresh a secondary region instead of flushing a primary region. The default value (4) configures the file refresh so that the biggest secondary region replica is 4 times bigger than the biggest primary region. Disabling this feature is not recommended. However, if you want to do so, set this property to a large value.

2. Set the client-side properties in your hbase-site.xml configuration file for all clients, applications, and servers in your HBase cluster that use region replicas. The following table lists client-side properties.

Table 3: Client-Side Properties for HBase HA

Property	Example value	Description
hbase.ipc.client. specificThreadForWriting	true	Specifies whether to enable interruption of RPC threads at the client side. This is required for region replicas with fallback RPC’s to secondary regions.
hbase.client. primaryCallTimeout.get	10000	Specifies the timeout (in microseconds) before secondary fallback RPC’s are submitted for get requests with Consistency.TIMELINE to the secondary replicas of the regions. The default value is 10ms. Setting this to a smaller value increases the number of RPC’s, but lowers 99th-percentile latencies.
hbase.client. primaryCallTimeout. multiget	10000	Specifies the timeout (in microseconds) before secondary fallback RPC’s are submitted for multi-get requests (HTable.get(List<Get>)) with Consistency.TIMELINE to the secondary replicas of the regions. The default value is 10ms. Setting this to a smaller value increases the number of RPC’s, but lowers 99th-percentile latencies.
hbase.client. primaryCallTimeout.scan	1000000	Specifies the timeout (in microseconds) before secondary fallback RPC’s are submitted for scan requests with Consistency.TIMELINE to the secondary replicas of the regions. The default value is 1 second. Setting this to a smaller value increases the number of RPC’s, but lowers 99th-percentile latencies.
hbase.meta.replicas.use	true	Specifies whether to use META table replicas or not. The default value is false.

3. Restart the HBase Master and RegionServers.

Related Concepts

Propagating Writes to Region Replicas

Creating Highly Available HBase Tables with the HBase Java API

HBase tables are not highly available by default. To enable high availability, designate a table as HA during table creation.

HBase Java API

Create highly available HBase tables programmatically, using the Java API, as shown in the following example:

```
HTableDescriptor htd =
    new HTableDesscriptor(TableName.valueOf("test_table"));
htd.setRegionReplication(2);
...
admin.createTable(htd);
```

This example creates a table named test_table that is replicated to one secondary region. To replicate test_table to two secondary replicas, pass 3 as a parameter to the setRegionReplication() method.

Related Concepts

Propagating Writes to Region Replicas

Creating Highly Available HBase Tables with the HBase Shell

Create HA tables using the HBase shell using the REGION_REPLICATION keyword.

HBase Shell

Valid values are 1, 2, and 3, indicating the total number of copies. The default value is 1.

The following example creates a table named t1 that is replicated to one secondary replica:

```
CREATE 't1', 'f1', {REGION_REPLICATION => 2}
```

To replicate t1 to two secondary regions, set REGION_REPLICATION to 3:

```
CREATE 't1', 'f1', {REGION_REPLICATION => 3}
```

Related Concepts

Propagating Writes to Region Replicas

Querying Secondary Regions

You can query HA-enabled HBase tables with the Java API or HBase Shell.

This section describes how to query HA-enabled HBase tables.

Querying HBase with the Java API

The HBase Java API allows application developers to specify the desired data consistency for a query using the setConsistency() method, as shown in the following example. A new enum, CONSISTENCY, specifies two levels of data consistency: TIMELINE and STRONG.

```
Get get = new Get(row);
get.setConsistency(CONSISTENCY.TIMELINE);
...
Result result = table.get(get);
```

HBase application developers can also pass multiple gets:

```
Get get1 = new Get(row);
get1.setConsistency(Consistency.TIMELINE);
...
ArrayList<Get> gets = new ArrayList<Get>();
...
```

```
Result[] results = table.get(gets);
```

The setConsistency() method is also available for Scan objects:

```
Scan scan = new Scan();
scan.setConsistency(CONSISTENCY.TIMELINE);
...
ResultScanner scanner = table.getScanner(scan);
```

In addition, you can use the Result.isStale() method to determine whether the query results arrived from the primary or a secondary replica:

```
Result result = table.get(get);
if (result.isStale()) {
   ...
}
```

Querying HBase Interactively

To specify the desired data consistency for each query, use the HBase shell:

hbase(main):001:0> get 't1', 'r6', {CONSISTENCY => "TIMELINE"}

Interactive scans also accept this syntax:

hbase(main):001:0> scan 't1', {CONSISTENCY => 'TIMELINE'}

Note:

This release of HBase does not provide a mechanism to determine if the results from an interactive query arrived from the primary or a secondary replica.

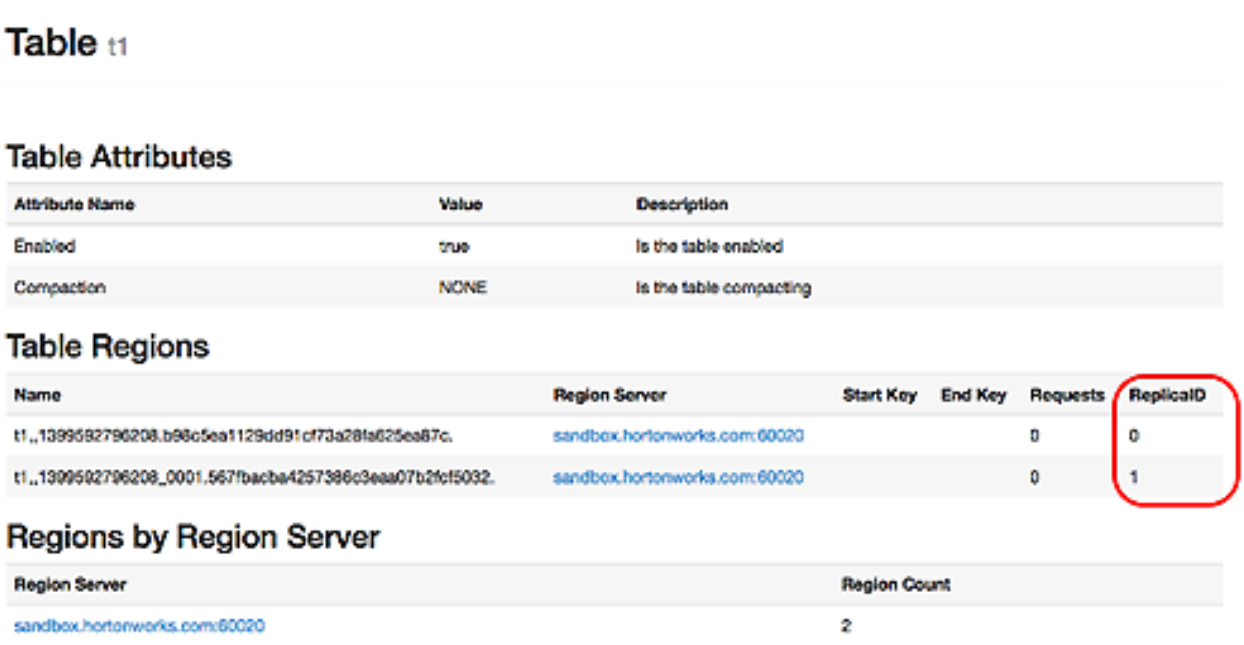
You can also request a specific region replica for debugging:

```
hbase> get 't1', 'r6', {REGION_REPLICA_ID=>0, CONSISTENCY=>'TIMELINE'}
hbase> get 't1', 'r6', {REGION_REPLICA_ID=>2, CONSISTENCY=>'TIMELINE'}
```

Monitoring Secondary Region Replicas

You can access the HBase Master server user interface and monitor the secondary region replicas.

HBase provides highly available tables by replicating table regions. All replicated regions have a unique replica ID. The replica ID for a primary region is always 0. The HBase web-based interface displays the replica IDs for all defined table regions. In the following example, the table t1 has two regions. The secondary region is identified by a replica ID of 1.



To access the HBase Master Server user interface, point your browser to port 16010.

HBase Cluster Replication for Geographic Data Distribution

HBase provides a cluster replication mechanism which allows you to keep one cluster's state synchronized with that of another cluster, using the write-ahead log (WAL) of the source cluster to propagate the changes.

The use cases for cluster replication include the following scenarios:

- Backup and disaster recovery
- Data aggregation
- Geographic data distribution, such as data centers
- Online data ingestion combined with offline data analytics

Note:

Replication is enabled at the granularity of the column family. Before enabling replication for a column family, create the table and all column families to be replicated on the destination cluster.

HBase Cluster Replication Overview

An HBase cluster can either be an active cluster or passive cluster to ensure cluster replication.

Cluster replication uses a source-push methodology. An HBase cluster can be a 'source' cluster, which means it is the source of the new data (also known as a 'master' or 'active' cluster), a 'destination' cluster, which means that it is the cluster that receives the new data by way of replication (also known as a 'slave' or 'passive' cluster), or an HBase cluster can fulfill both roles at once. Replication is asynchronous, and the goal of replication is eventual consistency. When the source receives an edit to a column family with replication enabled, that edit is propagated to all destination clusters using the WAL for that column family on the RegionServer that manages the relevant region.

When data is replicated from one cluster to another, the original source of the data is tracked by using a cluster ID which is part of the metadata. In HBase 0.96 and newer, all clusters that have already consumed the data are also tracked. This prevents replication loops.

The WALs for each RegionServer must be kept in HDFS as long as they are needed to replicate data to a slave cluster. Each RegionServer reads from the oldest log it needs to replicate and keeps track of its progress by processing WALs inside ZooKeeper to simplify failure recovery. The position marker which indicates a slave cluster's progress, as well as the queue of WALs to process, may be different for every slave cluster.

The clusters participating in replication can be of different sizes. The master cluster relies on randomization to attempt to balance the stream of replication on the slave clusters. It is expected that the slave cluster has storage capacity to hold the replicated data, as well as any data it is responsible for ingesting. If a slave cluster runs out of room, or is inaccessible for other reasons, it throws an error, the master retains the WAL, and then retries the replication at intervals.

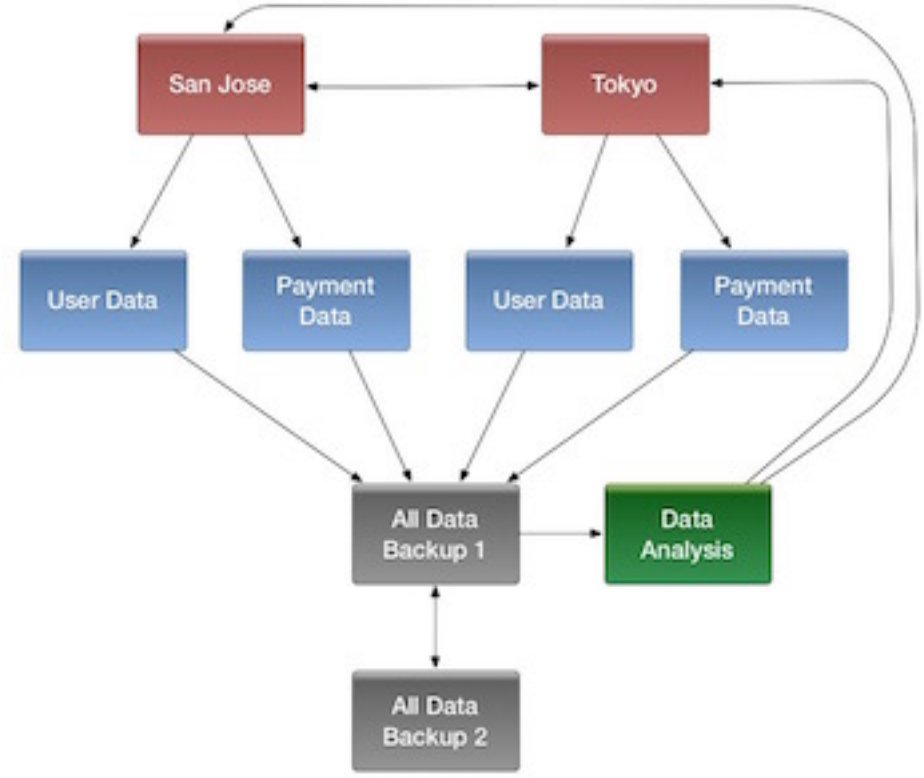
HBase Cluster Topologies

Clusters can propagate changes to single or multiple destination clusters.

- A central source cluster might propagate changes out to multiple destination clusters, for failover or due to geographic distribution.
- A source cluster might push changes to a destination cluster, which might also push its own changes back to the original cluster.
- Many different low-latency clusters might push changes to one centralized cluster for backup or resource-intensive data analytics jobs. The processed data might then be replicated back to the low-latency clusters.

Multiple levels of replication may be chained together to suit your organization's needs. The following diagram shows a hypothetical scenario for a complex cluster replication configuration. The arrows indicate the data paths.

Example of a Complex Cluster Replication Configuration



HBase replication borrows many concepts from the statement-based replication design used by MySQL. Instead of SQL statements, entire WALEdits, which consist of multiple cell inserts that come from Put and Delete operations on the clients, are replicated in order to maintain atomicity.

Managing and Configuring HBase Cluster Replication

Implementing HBase cluster replication enables you to achieve High Availability (HA). HBase supports replication across multiple clusters. This can help you setup HA and enable disaster recovery.

Related Tasks

Setting Up HBase Replication Among Kerberos Secured Clusters

Manually Enable HBase Replication

After ensuring that HBase is running on both the source and destination clusters, you must configure certain parameters to manually enable replication.

Procedure

1. Configure the source and destination clusters and ensure that you have HBase running in both clusters. HBase master and region servers in the source cluster must be able to communicate with the master and all region servers in the destination cluster.
2. On both clusters, create tables with the same names and column families, so that the destination cluster stores the data that it receives in a logical location:

   ```
   hbase shell>create "t1","cf1"
   ```

3. All hosts in the source and destination clusters should be reachable to each other. If both clusters use the same ZooKeeper cluster, you must use a different zookeeper.znode.parent, because they cannot write in the same folder.
4. On the source cluster, in HBase shell, add the destination cluster as a peer:

   ```
   hbase shell>add_peer “us_east”,”hostname.of.zookeeper:2181:/path-to-hbase”
   ```

5. On HDP, path-to-hbase is either “/hbase-unsecure” or “/hbase-secure”. On the destination cluster, open the hbase-site.xml file and look at the value of zookeeper.znode.parent to find out the HBase directory.
6. Ensure that replication has not been disabled. Ensure that the hbase.replication setting is set to true.
7. On the source cluster, in HBase shell, enable the table replication:

   Run

   ```
   hbase shell>enable_table_replication "t1"
   ```

8. Copy the HBase data from the source cluster to the destination cluster:

   ```
   $>hbase org.apache.hadoop.hbase.mapreduce.CopyTable --
   peer.adr=hostname.of.zookeeper:2181:/hbase-unsecure t1
   ```

Pause and Stop HBase Replication

Run HBase table replication commands to pause or stop HBase replication.

Procedure

1. To pause HBase cluster replication, use the disable_table_replication command:

   Run

   ```
   hbase shell>disable_table_replication "t1"
   ```

   With this, you can temporarily stop replication. To re-enable the replication, use the enable_table_replication command.
2. To permanently disable replication, remove the replication relationship:

   Run

   ```
   hbase shell>remove_peer “us_east”
   ```

HBase Cluster Management Commands

You can use the common HBase Cluster Management commands to perform operations related to replication.

Table 4: Table of HBase Cluster Management Commands and Descriptions

Command	Description
add_peer <ID> <CLUSTER_KEY>	Adds a replication relationship between two clusters: • ID: A unique string, which must not contain a hyphen. • CLUSTER_KEY: Composed using the following format: hbase.zookeeper.quorum:hbase.zookeeper.property.clientPort:zookeeper.znode.parent
list_peers	Lists all replication relationships known by the cluster.
enable_peer <ID>	Enables a previously-disabled replication relationship.
disable_peer <ID>	Disables a replication relationship. After disabling, HBase no longer sends edits to that peer cluster, but continues to track the new WALs that are required for replication to commence again if it is re-enabled.
remove_peer <ID>	Disables and removes a replication relationship. After removal, HBase no longer sends edits to that peer cluster nor does it track WALs.
enable_table_replication <TABLE_NAME>	Enables the table replication switch for all of the column families associated with that table. If the table is not found in the destination cluster, one is created with the same name and column families.
disable_table_replication <TABLE_NAME>	Disables the table replication switch for all of the column families associated with that table.

Verifying Replicated HBase Data

The VerifyReplication MapReduce job, which is included in HBase, performs a systematic comparison of replicated data between two different clusters.

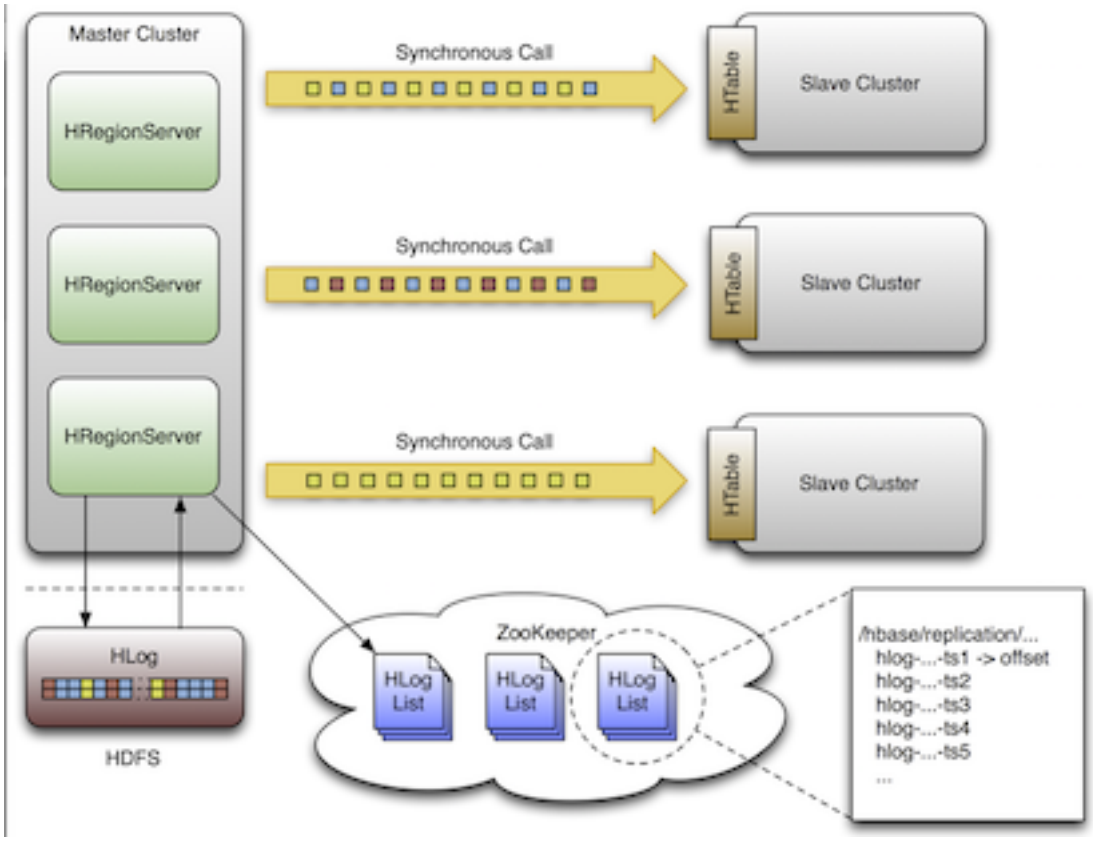
Run the VerifyReplication job on the master cluster, supplying it with the peer ID and table name to use for validation. You can limit the verification further by specifying a time range or specific column families. The job short name is verifyrep. To run the job, use a command like the following:

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath`
"${HADOOP_HOME}/bin/hadoop" jar "${HBASE_HOME}/hbase-server-VERSION.jar"
verifyrep --starttime=<timestamp> --stoptime=<timestamp> --families=<myFam>
 <ID> <tableName>
```

The VerifyReplication command prints out GOODROWS and BADROWS counters to indicate rows that did and did not replicate correctly.

HBase Cluster Replication Details

A Write Ahead Log edit goes through a series of steps to complete the replication process.



WAL (Write Ahead Log) Edit Process

A single WAL edit goes through the following steps when it is replicated to a slave cluster:

1. An HBase client uses a Put or Delete operation to manipulate data in HBase.
2. The RegionServer writes the request to the WAL in such a way that it can be replayed if the write operation is not successful.
3. If the changed cell corresponds to a column family that is scoped for replication, the edit is added to the queue for replication.
4. In a separate thread, the edit is read from the log as part of a batch process. Only the KeyValues that are eligible for replication are kept. KeyValues that are eligible for replication are those KeyValues that are:
   - Part of a column family whose schema is scoped GLOBAL.
   - Not part of a catalog such as hbase:meta.
   - Have not originated from the target slave cluster.
   - Have not already been consumed by the target slave cluster.
5. The WAL edit is tagged with the master's UUID and added to a buffer. When the buffer is full or the reader reaches the end of the file, the buffer is sent to a random RegionServer on the slave cluster.
6. The RegionServer reads the edits sequentially and separates them into buffers, one buffer per table. After all edits are read, each buffer is flushed using Table, the HBase client. The UUID of the master RegionServer and the UUIDs of the slaves, which have already consumed the data, are preserved in the edits when they are applied. This prevents replication loops.
7. The offset for the WAL that is currently being replicated in the master is registered in ZooKeeper.
8. The edit is inserted as described in Steps 1, 2, and 3.
9. In a separate thread, the RegionServer reads, filters, and edits the log edits as described in Step 4. The slave RegionServer does not answer the RPC call.
10. The master RegionServer sleeps and tries again. The number of attempts can be configured.
11. If the slave RegionServer is still not available, the master selects a new subset of RegionServers to replicate to, and tries again to send the buffer of edits.

12. Meanwhile, the WALs are rolled and stored in a queue in ZooKeeper. Logs that are archived by their RegionServer (by moving them from the RegionServer log directory to a central log directory) update their paths in the in-memory queue of the replicating thread.
13. When the slave cluster is finally available, the buffer is applied in the same way as during normal processing. The master RegionServer then replicates the backlog of logs that accumulated during the outage.

Related Information

Table HBase Client

Spreading Queue Failover Load

Set configuration parameters to maintain an even distribution of replication activity over the servers in the cluster.

When replication is active, a subset of RegionServers in the source cluster is responsible for shipping edits to the sink. This responsibility must be failed over like all other RegionServer functions if a process or node crashes. The following configuration settings are recommended for maintaining an even distribution of replication activity over the remaining live servers in the source cluster:

- Set replication.source.maxretriesmultiplier to 300.
- Set replication.source.sleepforretries to 1 (1 second). This value, combined with the value of replication.source.maxretriesmultiplier, causes the retry cycle to last about 5 minutes.
- Set replication.sleep.before.failover to 30000 (30 seconds) in the source cluster site configuration.

Preserving Tags During Replication

Configure the codecs to prevent stripping of the tags during replication.

By default, the codec used for replication between clusters strips tags, such as cell-level ACLs, from cells. To prevent the tags from being stripped, you can use a different codec which does not strip them. Configure hbase.replication.rpc.codec to use org.apache.hadoop.hbase.codec.KeyValueCodecWithTags, on both the source and the sink RegionServers which are involved in the replication.

HBase Replication Internals

The state of HBase replication is contained in the base node /hbase/replication. The peers znode contains a list of all peer replication clusters with their respective statuses. The rs znode contains a list of WAL logs to be replicated.

Replication State in ZooKeeper

HBase replication maintains its state in ZooKeeper. By default, the state is contained in the base node /hbase/replication. This node contains two child nodes, the Peers znode and the RS znode.

The Peers Znode

The peers znode is stored in /hbase/replication/peers by default. It consists of a list of all peer replication clusters along with the status of each of them. The value of each peer is its cluster key, which is provided in the HBase shell. The cluster key contains a list of ZooKeeper nodes in the cluster quorum, the client port for the ZooKeeper quorum, and the base znode for HBase in HDFS on that cluster.

The RS Znode

The rs znode contains a list of WAL logs which need to be replicated. This list is divided into a set of queues organized by Region Server and the peer cluster that the RegionServer is shipping the logs to. The rs znode has one child znode for each RegionServer in the cluster. The child znode name is the RegionServer hostname, client port, and start code. This list includes both live and dead RegionServers.

Choosing RegionServers to Replicate to

The master cluster RegionServers chooses potential recipients for replication and the Zookeeper watcher monitors changes in the composition of the slave clusters.

When a master cluster RegionServer initiates a replication source to a slave cluster, it first connects to the ZooKeeper ensemble of the slave using the provided cluster key. Then it scans the rs/ directory to discover all the available 'sinks' (RegionServers that are accepting incoming streams of edits to replicate) and randomly chooses a subset of them using a configured ratio which has a default value of 10 per cent. For example, if a slave cluster has 150 servers, 15 are chosen as potential recipients for edits sent by the master cluster RegionServer. Because this selection is performed by each master RegionServer, the probability that all slave RegionServers are used is very high. This

method works for clusters of any size. For example, a master cluster of 10 servers replicating to a slave cluster of 5 servers with a ratio of 10 per cent causes the master cluster RegionServers to choose one server each at random.

A ZooKeeper watcher is placed on the ${zookeeper.znode.parent}/rs node of the slave cluster by each of the master cluster RegionServers. This watcher monitors changes in the composition of the slave cluster. When nodes are removed from the slave cluster, or if nodes go down or come back up, the master cluster RegionServers respond by selecting a new pool of slave Region Servers to which to replicate.

Keeping Track of Logs

Replication of logs is done atomically and does not generate an exception when the reader is currently reading a log.

Each master cluster RegionServer has its own znode in the replication znodes hierarchy. It contains one znode per peer cluster. For example, if there are 5 slave clusters, 5 znodes are created, and each of these contain a queue of WALs to process. Each of these queues tracks the WALs created by that RegionServer, but they can differ in size. For example, if one slave cluster becomes unavailable for some time, the WALs should not be deleted. They need to stay in the queue while the others are processed.

When a source is instantiated, it contains the current WAL that the RegionServer is writing to. During log rolling, the new file is added to the queue of each slave cluster znode just before it is made available. This ensures that all the sources are aware that a new log exists before the RegionServer is able to append edits into it. However, this operation is now more expensive. The queue items are discarded when the replication thread cannot read more entries from a file because it reached the end of the last block and there are other files in the queue. This means that if a source is up to date and replicates from the log that the RegionServer writes to, reading up to the end of the current file does not delete the item in the queue.

A log can be archived if it is no longer used or if the number of logs exceeds the hbase.regionserver.maxlogs setting because the insertion rate is faster than the rate at which the regions are flushed. When a log is archived, the source threads are notified that the path for that log changed. If a particular source has already finished with an archived log, it ignores the message. If the log is in the queue, the path is updated in memory. If the log is currently being replicated, the change is done atomically so that the reader does not attempt to open the file when it has already been moved. Because moving a file is a NameNode operation, if the reader is currently reading the log, it does not generate an exception.

Related Concepts

RegionServer Failover

Reading, Filtering, and Sending Edits

The rate at which the source attempts to read the data depends on the filtering of log entries and and the total size of the list edits to replicate per slave.

By default, a source attempts to read from a WAL and ships log entries to a sink as quickly as possible. Speed is limited by the filtering of log entries. Only Key-Values that are scoped GLOBAL and that do not belong to catalog tables are retained. Speed is limited by total size of the list of edits to replicate per slave, which is limited to 64 MB by default. With this configuration, a master cluster Region Server with three slaves would use at most 192 MB to store data to replicate. This does not account for the data which was filtered but not garbage collected.

Once the maximum size of edits has been buffered or the reader reaches the end of the WAL, the source thread stops reading and chooses at random a sink to replicate to from the list that was generated by keeping only a subset of slave Region Servers. It directly issues an RPC to the chosen RegionServer and waits for the method to return. If the RPC is successful, the source determines whether the current file has been emptied or whether it contains more data that needs to be read. If the file has been emptied, the source deletes the znode in the queue. Otherwise, it registers the new offset in the log znode. If the RPC throws an exception, the source retries 10 times before trying to find a different sink.

Cleaning Logs

If replication is not enabled, the log-cleaning thread of the master deletes old logs using a configured TTL (Time To Live).

This TTL-based method does not work well with replication because archived logs that have exceeded their TTL may still be in a queue. The default behavior is augmented so that if a log is past its TTL, the cleaning thread looks up

every queue until it finds the log. During this process, queues that are found are cached. If the log is not found in any queues, the log is deleted. The next time the cleaning process needs to look for a log, it starts by using its cached list.

RegionServer Failover

When no RegionServers are failing, keeping track of the logs in ZooKeeper adds no value. Unfortunately, RegionServers do fail, and since ZooKeeper is highly available, it is useful for managing the transfer of the queues in the event of a failure.

Each of the master cluster RegionServers keeps a watcher on every other RegionServer, in order to be notified when one becomes unavailable just as the master does. When a failure happens, they all race to create a znode called lock inside the unavailable RegionServer znode that contains its queues. The RegionServer that creates it successfully then transfers all the queues to its own znode, one at a time since ZooKeeper does not support renaming queues. After all queues are transferred, they are deleted from the old location. The recovered znodes are then renamed with the slave cluster ID appended to the name of the server that failed.

Next, the master cluster RegionServer creates one new source thread per copied queue. Each of the source threads follows the 'read/filter/ship pattern.' Those queues never receive new data because they do not belong to their new RegionServer. When the reader hits the end of the last log, the queue znode is deleted and the master cluster RegionServer closes that replication source.

For example, the following hierarchy represents what the znodes layout might be for a master cluster with 3 RegionServers that are replicating to a single slave with the ID of 2. The RegionServer znodes contain a peers znode that contains a single queue. The znode names in the queues represent the actual file names on HDFS in the form address,port.timestamp:

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1234  (Contains a position)
      1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    2/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
      1.1.1.2,60020.1312
  1.1.1.3,60020,    123456630/
    2/
      1.1.1.3,60020.1280  (Contains a position)
```

Assume that 1.1.1.2 loses its ZooKeeper session. The survivors race to create a lock, and, arbitrarily, 1.1.1.3 wins. It then starts transferring all the queues to the znode of its local peers by appending the name of the server that failed. Right before 1.1.1.3 is able to clean up the old znodes, the layout looks like the following:

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1234  (Contains a position)
      1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    lock
    2/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
      1.1.1.2,60020.1312
  1.1.1.3,60020,123456630/
    2/
      1.1.1.3,60020.1280  (Contains a position)

    2-1.1.1.2,60020,123456790/
      1.1.1.2,60020.1214  (Contains a position)
      1.1.1.2,60020.1248
```

```
      1.1.1.2,60020.1312
```

Some time later, but before 1.1.1.3 is able to finish replicating the last WAL from 1.1.1.2, it also becomes unavailable. Some new logs were also created in the normal queues. The last RegionServer then tries to lock 1.1.1.3's znode and begins transferring all the queues. Then the new layout is:

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    2/
      1.1.1.1,60020.1378  (Contains a position)

    2-1.1.1.3,60020,123456630/
      1.1.1.3,60020.1325  (Contains a position)
      1.1.1.3,60020.1401

    2-1.1.1.2,60020,123456790-1.1.1.3,60020,123456630/
      1.1.1.2,60020.1312  (Contains a position)
  1.1.1.3,60020,123456630/
    lock
    2/
      1.1.1.3,60020.1325  (Contains a position)
      1.1.1.3,60020.1401

    2-1.1.1.2,60020,123456790/
      1.1.1.2,60020.1312  (Contains a position)
```

Related Concepts

Keeping Track of Logs

HBase Replication Metrics

Replication metrics are exposed at the global RegionServer level and at the peer level (since HBase 0.95).

Metric	Description
source.sizeOfLogQueue	Number of WALs to process (excludes the one which is being processed) at the replication source.
source.shippedOps	Number of of mutations shipped.
source.logEditsRead	Number of mutations read from WALs at the replication source.
source.ageOfLastShippedOp	Age of last batch shipped by the replication source.

Replication Configuration Options

You can configure replication using the various available options.

Option	Description	Default Setting
zookeeper.znode.parent	Name of the base ZooKeeper znode that is used for HBase.	/hbase
zookeeper.znode.replication	Name of the base znode used for replication.	replication
zookeeper.znode.replication. peers	Name of the peer znode.	peers
zookeeper.znode.replication. peers.state	Name of the peer-state znode.	peer-state
zookeeper.znode.replication. rx	Name of the rs znode.	rs
hbase.replication	Whether replication is enabled or disabled on the cluster.	false

Option	Description	Default Setting
replication.sleep.before. failover	Number of milliseconds a worker should sleep before attempting to replicate the WAL queues for a dead RegionServer.	--
replication.executor.workers	Number of RegionServers a RegionServer should attempt to failover simultaneously.	1

Monitoring Replication Status

You can use the HBase shell command status 'replication' to monitor the replication status on your cluster.

The command has three variations:

Command	Description
* status 'replication'	Prints the status of each source and its sinks, sorted by hostname.
* status 'replication', 'source'	Prints the status for each replication source, sorted by hostname.
* status 'replication', 'sink'	Prints the status for each replication sink, sorted by hostname.

Setting Up HBase Replication Among Kerberos Secured Clusters

HBase replication supports Kerberos, if you want to ensure secure communication between two clusters.

Before you begin

Prerequisite

You have configured HBase replication in two separate clusters.

Prior to configuring secure HBase, you must configure cross realm authentication for Kerberos, ZooKeeper, and Apache Hadoop.

Procedure

1. Create krbtgt principals for the two realms.

   For example, if you have two realms called HDP1.COM and HDP2.COM, the realms must share a key. In this case, you add the following principles in both the realms:

   krbtgt/HDP1.COM@HDP2.COM and krbtgt/HDP2.COM@HDP1.COM

   There must be at least one common encryption mode between these two realms:

   ```
   HDP1 Cluster
            kadmin.local:addprinc krbtgt/HDP1.COM@HDP2.COM
            kadmin.local:addprinc krbtgt/HDP2.COM@HDP1.COM
   ```

   ```
   HDP2 Cluster
            kadmin.local:addprinc krbtgt/HDP1.COM@HDP2.COM
            kadmin.local:addprinc krbtgt/HDP2.COM@HDP1.COM
   ```

   Note:

   To ensure, there is at least one common encryption mode between the realms, you can use the -e option in addprinc to specify the list of encryption types. Refer to the “Supported Encryption Types” in mit kerberos manual in your deployment to view all possible options.

   kadmin.local:addprinc -e "<enc_type_list >" krbtgt/HDP1.COM@HDP2.COM

2. Add rules in the slave ZooKeeper to create short names based on the incoming principal.

   To do this, add a system level property in java.env, as defined in the conf directory.

a) On the HDP1 cluster, add support for the realm called HDP2.COM, and have two members in the principal (such as service/instance@HDP2.com):

```
-Dzookeeper.security.auth_to_local=RULE:[2:\$1@\$0](.*@\\QHDP2.COM\\E
$)s/@\\QHDP2.COM\\E$//DEFAULT
```

b) On the HDP2 cluster, add support for the realm called HDP1.COM, and have two members in the principal (such as service/instance@HDP1.com):

```
-Dzookeeper.security.auth_to_local=RULE:[2:\$1@\$0](.*@\\QHDP1.COM\\E
$)s/@\\QHDP1.COM\\E$//DEFAULT
```

The DEFAULT value defines the default rule.

3. Add rules for creating short names in the Hadoop processes. To do this, add the hadoop.security.auth_to_local property in the core-site.xml file in the replica cluster.

a) On the HDP1 cluster, add the following:

```
<property>
 <name>hadoop.security.auth_to_local</name>
 <value>
 RULE:[2:\$1@\$0](.*@\\QHDP2.COM\\E$)s/@\\QHDP2.COM\\E$//DEFAULT
 </value>
```

b) On the HDP2 cluster, add the following:

```
<property>
 <name>hadoop.security.auth_to_local</name>
 <value>
 RULE:[2:\$1@\$0](.*@\\QHDP1.COM\\E$)s/@\\QHDP1.COM\\E$//DEFAULT
 </value>
```

4. Manage and configure HBase cluster replication to complete the replication process.

Related Tasks

Managing and Configuring HBase Cluster Replication

Configuring NameNode High Availability

The HDFS NameNode High Availability feature enables you to run redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby. This eliminates the NameNode as a potential single point of failure (SPOF) in an HDFS cluster. As of Hadoop 3.0, you can configure more than one backup NameNode.

Prior to Hadoop 2.0, the NameNode represented a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine. This situation impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in periods of cluster downtime.

HDFS NameNode HA avoids this by facilitating either a fast failover to one or more backup NameNodes during machine crash, or a graceful administrator-initiated failover during planned maintenance.

This guide provides an overview of the HDFS NameNode High Availability (HA) feature, instructions on how to deploy Hue with an HA cluster, and instructions on how to enable HA on top of an existing HDP cluster using the Quorum Journal Manager (QJM) and ZooKeeper Failover Controller for configuration and management. Using

the QJM and ZooKeeper Failover Controller enables the sharing of edit logs between the Active and Standby NameNodes.

Note:

This guide assumes that an existing HDP cluster has been manually installed and deployed. If your existing HDP cluster was installed using Ambari, configure NameNode HA using the Ambari wizard, as described in the Ambari documentation.

NameNode Architecture

In a typical HA cluster, two or more separate machines are configured as NameNodes. In a working cluster, one of the NameNode machine is in the Active state, and the others are in the Standby state.

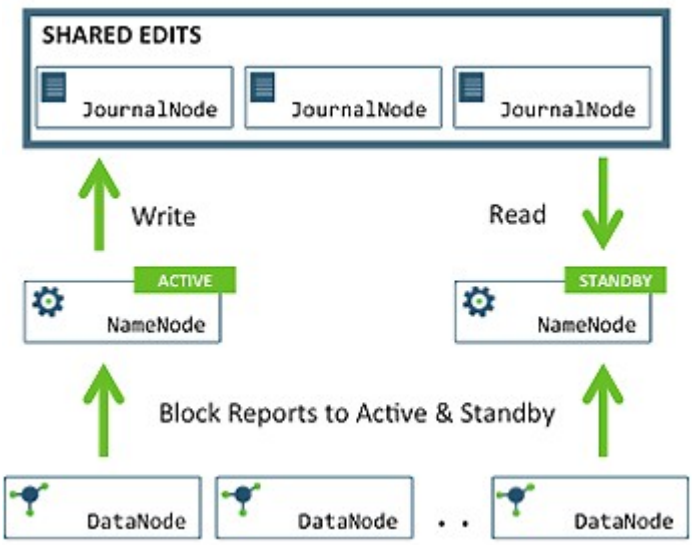
The Active NameNode is responsible for all client operations in the cluster, while the Standby NameNode acts as a backup. The Standby machine maintains enough state to provide a fast failover (if required).

In order for the Standby node to keep its state synchronized with the Active node, both nodes communicate with a group of separate daemons called JournalNodes (JNs). When the Active node performs any namespace modification, the Active node durably logs a modification record to a majority of these JNs. The Standby node reads the edits from the JNs and continuously watches the JNs for changes to the edit log. Once the Standby Node observes the edits, it applies these edits to its own namespace. When using QJM, JournalNodes act as the shared editlog storage. In a failover event, the Standby ensures that it has read all of the edits from the JounalNodes before promoting itself to the Active state. (This mechanism ensures that the namespace state is fully synchronized before a failover completes.)

Note:

Secondary NameNode is not required in HA configuration because the Standby node also performs the tasks of the Secondary NameNode.

To provide a fast failover, it is also necessary that the Standby node have up-to-date information on the location of blocks in your cluster. To get accurate information about the block locations, DataNodes are configured with the location of all the NameNodes, and send block location information and heartbeats to all the NameNode machines.



It is vital for the correct operation of an HA cluster that only one of the NameNodes should be Active at a time. Failure to do so would cause the namespace state to quickly diverge between the NameNode machines, thus causing potential data loss. (This situation is referred to as a split-brain scenario.)

To prevent the split-brain scenario, the JournalNodes allow only one NameNode to be a writer at a time. During failover, the NameNode, that is chosen to become active, takes over the role of writing to the JournalNodes. This process prevents the other NameNode from continuing in the Active state and thus lets the new Active node proceed with the failover safely.

Preparing the Hardware Resources for NameNode High Availability

Make sure that you prepare the required hardware resources for High Availability.

- NameNode machines: The machines where you run Active and Standby NameNodes, should have exactly the same hardware. .
- JournalNode machines: The machines where you run the JournalNodes. The JournalNode daemon is relatively lightweight, so these daemons may reasonably be co- located on machines with other Hadoop daemons, for example the NameNodes or the YARN ResourceManager.

  Note:

  There must be at least three JournalNode daemons, because edit log modifications must be written to a majority of JNs. This lets the system tolerate failure of a single machine. You may also run more than three JournalNodes, but in order to increase the number of failures that the system can tolerate, you must run an odd number of JNs (3, 5, 7, and so on).

  Note that when running with N JournalNodes, the system can tolerate at most (N - 1) / 2 failures and continue to function normally.
- ZooKeeper machines: For automated failover functionality, there must be an existing ZooKeeper cluster available. The ZooKeeper service nodes can be co-located with other Hadoop daemons.

In an HA cluster, the Standby NameNode also performs checkpoints of the namespace state. Therefore, do not deploy a Secondary NameNode, CheckpointNode, or BackupNode in an HA cluster.

Deploying the NameNode HA Cluster

HA configuration is backward compatible and works with your existing single NameNode configuration.

The following instructions describe how to set up NameName HA on a manually-installed cluster. If you installed with Ambari and manage HDP on Ambari 2.0.0 or later, instead of the manual instructions use the Ambari documentation for the NameNode HA wizard.

Note:

HA cannot accept HDFS cluster names that include an underscore (_).

To deploy a NameNode HA cluster, use the steps in the following subsections.

Related Tasks

Configure and Deploy Automatic Failover

Configuring the NameNode HA Cluster

First, add High Availability configurations to your HDFS configuration files. Start by taking the HDFS configuration files from the original NameNode in your HDP cluster, and use that as the base, adding various properties to those files.

About this task

After you have added the configurations below, ensure that the same set of HDFS configuration files are propagated to all nodes in the HDP cluster. This ensures that all the nodes and services are able to interact with the highly available NameNode.

Add the following configuration options to your hdfs-site.xml file:

Procedure

- dfs.nameservices

  Choose an arbitrary but logical name (for example, mycluster) as the value for dfs.nameservices option. This name will be used for both configuration and authority component of absolute HDFS paths in the cluster.

```
<property>
 <name>dfs.nameservices</name>
```

```
 <value>mycluster</value>
 <description>Logical name for this new nameservice</description>
</property>
```

If you are also using HDFS Federation, this configuration setting should also include the list of other nameservices, HA or otherwise, as a comma-separated list.

- dfs.ha.namenodes.[$nameservice ID]

  Provide a list of comma-separated NameNode IDs. DataNodes use this this property to determine all the NameNodes in the cluster.

  For example, for the nameservice ID mycluster and individual NameNode IDs nn1, nn2, and nn3, the value of this property is:

```
<property>
 <name>dfs.ha.namenodes.mycluster</name>
 <value>nn1,nn2,nn3</value>
 <description>Unique identifiers for each NameNode in the
    nameservice</description>
</property>
```

  Note:

  The minimum number of NameNodes for HA is two, but you can configure more. You should not exceed five NameNodes due to communication overhead. Three NameNodes are recommended.

- dfs.namenode.rpc-address.[$nameservice ID].[$name node ID]

  Use this property to specify the fully-qualified RPC address for each NameNode to listen on.

  Continuing with the previous example, set the full address and IPC port of the NameNode process for the NameNode IDs above -- nn1, nn2, and nn3.

  Note that there will be three separate configuration options.

```
<property>
 <name>dfs.namenode.rpc-address.mycluster.nn1</name>
 <value>machine1.example.com:8020</value>
</property>

<property>
 <name>dfs.namenode.rpc-address.mycluster.nn2</name>
 <value>machine2.example.com:8020</value>
</property>

<property>
  <name>dfs.namenode.rpc-address.mycluster.nn3</name>
  <value>machine3.example.com:9820</value>
</property>
```

- dfs.namenode.http-address.[$nameservice ID].[$name node ID]

  Use this property to specify the fully-qualified HTTP address for each NameNode to listen on.

  Set the addresses for the NameNodes HTTP servers to listen on. For example:

```
<property>
  <name>dfs.namenode.http-address.mycluster.nn1</name>
  <value>machine1.example.com:9870</value>
</property>

<property>
  <name>dfs.namenode.http-address.mycluster.nn2</name>
  <value>machine2.example.com:9870</value>
</property>
```

```
<property>
  <name>dfs.namenode.http-address.mycluster.nn3</name>
  <value>machine3.example.com:9870</value>
</property>
```

Note:

If you have Hadoop security features enabled, set the https-address for each NameNode.

- dfs.namenode.shared.edits.dir

  Use this property to specify the URI that identifies a group of JournalNodes (JNs) where the NameNode will write/read edits.

  Configure the addresses of the JNs that provide the shared edits storage. The Active nameNode writes to this shared storage and the Standby NameNode reads from this location to stay up-to-date with all the file system changes.

  Although you must specify several JournalNode addresses, you must configure only one of these URIs for your cluster.

  The URI should be of the form:

  ```
  qjournal://host1:port1;host2:port2;host3:port3/journalId
  ```

  The Journal ID is a unique identifier for this nameservice, which allows a single set of JournalNodes to provide storage for multiple federated namesystems. You can reuse the nameservice ID for the journal identifier.

  For example, if the JournalNodes for a cluster were running on node1.example.com, node2.example.com, and node3.example.com, and the nameservice ID were mycluster, you would use the following value for this setting:

  ```
  <property>
   <name>dfs.namenode.shared.edits.dir</name>
   <value>qjournal://node1.example.com:8485;node2.example.com:
        8485;node3.example.com:8485/mycluster</value>
  </property>
  ```

  Note:

  Note that the default port for the JournalNode is 8485.

- dfs.client.failover.proxy.provider.[$nameservice ID]

  This property specifies the Java class that HDFS clients use to contact the Active NameNode. DFS Client uses this Java class to determine which NameNode is the current Active and therefore which NameNode is currently serving client requests.

  Use the ConfiguredFailoverProxyProvider implementation if you are not using a custom implementation.

  For example:

  ```
  <property>
   <name>dfs.client.failover.proxy.provider.mycluster</name>
   <value>org.apache.hadoop.hdfs.server.namenode.ha.
        ConfiguredFailoverProxyProvider</value>
  </property>
  ```

- dfs.ha.fencing.methods

  This property specifies a list of scripts or Java classes that will be used to fence the Active NameNode during a failover.

  For maintaining system correctness, it is important to have only one NameNode in the Active state at any given time. When using the Quorum Journal Manager, only one NameNode will ever be allowed to write to the JournalNodes, so there is no potential for corrupting the file system metadata from a split-brain scenario.

However, when a failover occurs, it is still possible that the previous Active NameNode could serve read requests to clients, which may be out of date until that NameNode shuts down when trying to write to the JournalNodes.

For this reason, it is still recommended to configure some fencing methods even when using the Quorum Journal Manager. To improve the availability of the system in the event the fencing mechanisms fail, it is advisable to configure a fencing method which is guaranteed to return success as the last fencing method in the list. Note that if you choose to use no actual fencing methods, you must set some value for this setting, for example shell(/bin/true).

The fencing methods used during a failover are configured as a carriage-return-separated list, which will be attempted in order until one indicates that fencing has succeeded. The following two methods are packaged with Hadoop: shell and sshfence. For information on implementing custom fencing method, see the org.apache.hadoop.ha.NodeFencer class.

sshfence: SSH to the Active NameNode and kill the process.

The sshfence option SSHes to the target node and uses fuser to kill the process listening on the service's TCP port. In order for this fencing option to work, it must be able to SSH to the target node without providing a passphrase. Ensure that you configure the dfs.ha.fencing.ssh.private-key-files option, which is a comma-separated list of SSH private key files.

For example:

```
<property>
 <name>dfs.ha.fencing.methods</name>
 <value>sshfence</value>
</property>

<property>
 <name>dfs.ha.fencing.ssh.private-key-files</name>
 <value>/home/exampleuser/.ssh/id_rsa</value>
</property>
```

Optionally, you can also configure a non-standard username or port to perform the SSH. You can also configure a timeout, in milliseconds, for the SSH, after which this fencing method will be considered to have failed. To configure non-standard username or port and timeout, see the properties given below:

```
<property>
 <name>dfs.ha.fencing.methods</name>
 <value>sshfence([[username][:port]])</value>
</property>

<property>
 <name>dfs.ha.fencing.ssh.connect-timeout</name>
 <value>30000</value>
</property>
```

shell: Run an arbitrary shell command to fence the Active NameNode.

The shell fencing method runs an arbitrary shell command:

```
<property>
 <name>dfs.ha.fencing.methods</name>
 <value>shell(/path/to/my/script.sh arg1 arg2 ...)</value>
</property>
```

The string between '(' and ')' is passed directly to a bash shell and may not include any closing parentheses.

The shell command will be run with an environment set up to contain all of the current Hadoop configuration variables, with the '_' character replacing any '.' characters in the configuration keys. The configuration used has already had any namenode-specific configurations promoted to their generic forms -- for example

dfs_namenode_rpc-address will contain the RPC address of the target node, even though the configuration may specify that variable as dfs.namenode.rpc-address.ns1.nn1.

Additionally, the following variables (referring to the target node to be fenced) are also available:

- $target_host: Hostname of the node to be fenced.
- $target_port: IPC port of the node to be fenced
- $target_address: The combination of $target_host and $target_port as host:port
- $target_nameserviceid: The nameservice ID of the NN to be fenced
- $target_namenodeid: The namenode ID of the NN to be fenced

These environment variables may also be used as substitutions in the shell command. For example:

```
<property>
 <name>dfs.ha.fencing.methods</name>
 <value>shell(/path/to/my/script.sh --nameservice=$target_nameserviceid
 $target_host:$target_port)</value>
</property>
```

If the shell command returns an exit code of 0, the fencing is successful.

Note:

This fencing method does not implement any timeout. If timeouts are necessary, they should be implemented in the shell script itself (for example, by forking a subshell to kill its parent in some number of seconds).

- fs.defaultFS The default path prefix used by the Hadoop FS client. Optionally, you may now configure the default path for Hadoop clients to use the new HA-enabled logical URI. For example, for mycluster nameservice ID, this will be the value of the authority portion of all of your HDFS paths. Configure this property in the core-site.xml file:

```
<property>
 <name>fs.defaultFS</name>
 <value>hdfs://mycluster</value>
</property>
```

- dfs.journalnode.edits.dir This is the absolute path on the JournalNode machines where the edits and other local state (used by the JNs) will be stored. You may only use a single path for this configuration. Redundancy for this data is provided by either running multiple separate JournalNodes or by configuring this directory on a locally-attached RAID array. For example:

```
<property>
 <name>dfs.journalnode.edits.dir</name>
 <value>/path/to/journal/node/local/data</value>
</property>
```

Note:

NameNode and NameNode HA failure may occur if the hadoop.security.authorization property in the core-site.xml file is set to true without Kerberos enabled on a NameNode HA cluster. Therefore you should only set this property to true when configuring HDP for Kerberos.

Deploying a NameNode HA Cluster

To deploy a NameNode HA cluster, you must initialize JournalNodes, run the required configurations for HA on the NameNodes, and validate the HA configuration.

About this task

In this task, we convert a non-HA cluster to HA. We use NN1 to denote the original NameNode in the non-HA setup, and NN2 to denote the other NameNode that is to be added in the HA setup. If you are using more than one standby NameNode, repeat the steps for NN2 on NN3.

Note:

HA clusters reuse the NameService ID to identify a single HDFS instance (that may consist of multiple HA NameNodes).

A new abstraction called NameNode ID is added with HA. Each NameNode in the cluster has a distinct NameNode ID to distinguish it.

To support a single configuration file for all of the NameNodes, the relevant configuration parameters are suffixed with both the NameService ID and the NameNode ID.

Procedure

1. Start the JournalNode daemons on those set of machines where the JNs are deployed. On each machine, execute the following command:

```
su –l hdfs –c "/usr/hdp/current/hadoop-hdfs-journalnode/../hadoop/sbin/
hdfs-daemon.sh start journalnode"
```

2. Wait for the daemon to start on each of the JN machines.
3. Initialize JournalNodes.
   - At the NN1 host machine, execute the following command:

     ```
     su –l hdfs –c "hdfs namenode -initializeSharedEdits -force"
     ```

     This command formats all the JournalNodes. This by default happens in an interactive way: the command prompts users for “Y/N” input to confirm the format. You can skip the prompt by using option -force or -nonInteractive.

     It also copies all the edits data after the most recent checkpoint from the edits directories of the local NameNode (NN1) to JournalNodes.
   - Initialize HA state in ZooKeeper. Execute the following command on NN1:

     ```
     hdfs zkfc -formatZK -force
     ```

     This command creates a znode in ZooKeeper. The failover system stores uses this znode for data storage.
   - Check to see if ZooKeeper is running. If not, start ZooKeeper by executing the following command on the ZooKeeper host machine(s).

     ```
     su - zookeeper -c "export ZOOCFGDIR=/usr/hdp/current/zookeeper-server/
     conf ; export ZOOCFG=zoo.cfg; source /usr/hdp/current/zookeeper-
     server/conf/zookeeper-env.sh ; /usr/hdp/current/zookeeper-server/bin/
     zkServer.sh start"
     ```
   - At the standby namenode host, execute the following command:

     ```
     su -l hdfs -c "hdfs namenode -bootstrapStandby -force"
     ```
4. Start NN1. At the NN1 host machine, execute the following command:

   ```
   su -l hdfs -c "/usr/hdp/current/hadoop-hdfs-namenode/../hadoop/sbin/hdfs-
   daemon.sh start namenode"
   ```

   Make sure that NN1 is running correctly.
5. Format NN2 and copy the latest checkpoint (FSImage) from NN1 to NN2 by executing the following command:

   ```
   su -l hdfs -c "hdfs namenode -bootstrapStandby -force"
   ```

This command connects with HH1 to get the namespace metadata and the checkpointed fsimage. This command also ensures that NN2 receives sufficient editlogs from the JournalNodes (corresponding to the fsimage). This command fails if JournalNodes are not correctly initialized and cannot provide the required editlogs.

6. Start NN2. Execute the following command on the NN2 host machine:

```
su -l hdfs -c "/usr/hdp/current/hadoop-hdfs-namenode/../hadoop/sbin/
hadoop-daemon.sh start namenode"
```

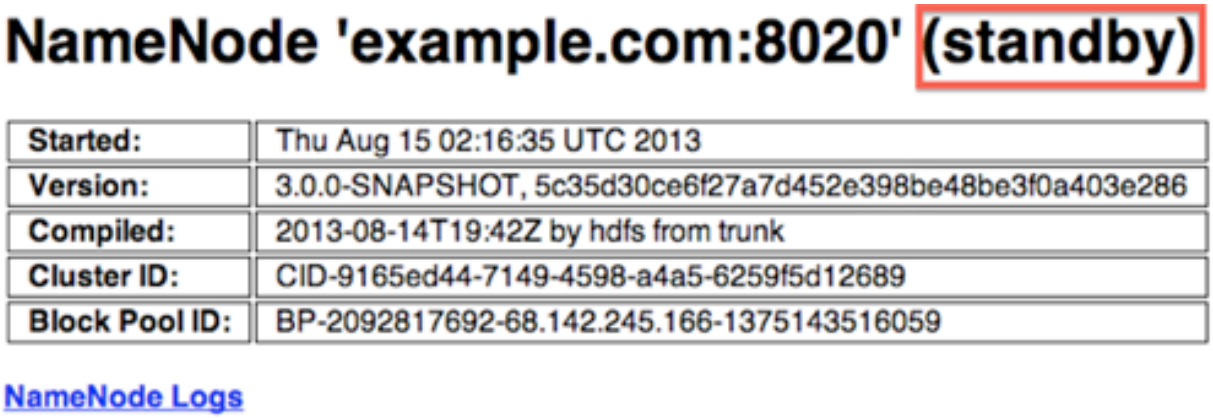
   Ensure that NN2 is running correctly.

7. If you are deploying an additional NameNode NN3, repeat steps 5 and 6 for NN3.
8. Start DataNodes. Execute the following command on all the DataNodes:

```
su -l hdfs -c "/usr/hdp/current/hadoop-hdfs-datanode/../hadoop/sbin/
hadoop-daemon.sh start datanode"
```

9. Validate the HA configuration.

   Go to the NameNodes' web pages separately by browsing to their configured HTTP addresses. Under the configured address label, you should see that HA state of the NameNode. The NameNode can be either in "standby" or "active" state.



   Note:

   The HA NameNode is initially in the Standby state after it is bootstrapped. You can also use either JMX (tag.HAState) to query the HA state of a NameNode. The following command can also be used to query the HA state of a NameNode:

   hdfs haadmin -getServiceState

10. Transition one of the HA NameNodes to the Active state.

    Initially, all the NameNodes are in the Standby state. Therefore you must transition one of the NameNodes to the Active state. This transition can be performed using one of the following options (ZooKeeper is not required for Option I).

    - Option I: Using CLI – Use the command line interface (CLI) to transition one of the NameNode to the Active State. Execute the following command on that NameNode host machine:

```
hdfs haadmin -failover --forcefence --forceactive <serviceId>
 <namenodeId>
```

    - Option II: Deploying Automatic Failover – You can configure and deploy automatic failover.

Deploying Hue with an HA Cluster

If you are going to use Hue with an HA Cluster, make changes to /etc/hue/conf/hue.ini file.

Procedure

1. Install the Hadoop HttpFS component on the Hue server.

   For RHEL/CentOS/Oracle Linux:

   ```
   yum install hadoop-httpfs
   ```

   For SLES:

   ```
   zypper install hadoop-httpfs
   ```

2. Modify /etc/hadoop-httpfs/conf/httpfs-env.sh to add the JDK path. In the file, ensure that JAVA_HOME is set:
   export JAVA_HOME=/usr/jdk64/jdk1.7.0_67
3. Configure the HttpFS service script by setting up the symlink in /etc/init.d:

   ```
   > ln -s /usr/hdp/{HDP2.4.x version number}
   /hadoop-httpfs/etc/rc.d/init.d/hadoop-httpfs /etc/init.d/hadoop-httpfs
   ```

4. Modify /etc/hadoop-httpfs/conf/httpfs-site.xml to configure HttpFS to talk to the cluster, by confirming that the following properties are correct:

   ```
   <property>
    <name>httpfs.proxyuser.hue.hosts</name>
    <value>*</value>
   </property>

   <property>
    <name>httpfs.proxyuser.hue.groups</name>
    <value>*</value>
   </property>
   ```

5. Start the HttpFS service.

   ```
   service hadoop-httpfs start
   ```

6. Modify the core-site.xml file. On the NameNodes and all the DataNodes, add the following properties to the $HADOOP_CONF_DIR /core-site.xml file, where $HADOOP_CONF_DIR is the directory for storing the Hadoop configuration files. For example, /etc/hadoop/conf.

   ```
   <property>
    <name>hadoop.proxyuser.httpfs.groups</name>
    <value>*</value>
   </property>

   <property>
    <name>hadoop.proxyuser.httpfs.hosts</name>
    <value>*</value>
   </property>
   ```

7. In the hue.ini file, under the [hadoop][[hdfs_clusters]][[[default]]] subsection, use the following variables to configure the cluster:

Property	Description	Example
fs_defaultfs	NameNode URL using the logical name for the new name service. For reference, this is the dfs.nameservices property in hdfs-site.xml in your Hadoop configuration.	hdfs://mycluster
webhdfs_url	URL to the HttpFS server.	http://c6401.apache.org:14000/ webhdfs/v1/

8. Restart Hue for the changes to take effect.

```
service hue restart
```

Deploying Oozie with an HA Cluster

You can configure multiple Oozie servers against the same database to provide High Availability (HA) for the Oozie service.

About this task

You need the following prerequisites:

Procedure

- A database that supports multiple concurrent connections. In order to have full HA, the database should also have HA support, or it becomes a single point of failure.

  Note:

  The default derby database does not support this.
- A ZooKeeper ensemble. Apache ZooKeeper is a distributed, open-source coordination service for distributed applications; the Oozie servers use it for coordinating access to the database and communicating with each other. In order to have full HA, there should be at least 3 ZooKeeper servers.
- Multiple Oozie servers.

  Important:

  While not strictly required, you should configure all ZooKeeper servers to have identical properties.
- A Loadbalancer, Virtual IP, or Round-Robin DNS. This is used to provide a single entry-point for users and for callbacks from the JobTracker.

  The load balancer should be configured for round-robin between the Oozie servers to distribute the requests. Users (using either the Oozie client, a web browser, or the REST API) should connect through the load balancer. In order to have full HA, the load balancer should also have HA support, or it becomes a single point of failure.

Related Tasks

Configure Oozie Failover

Related Information

Apache ZooKeeper

Operating a NameNode HA cluster

The dfsadmin command can be run on both active and standby NameNodes to operate the HA cluster.

- While operating an HA cluster, the Active NameNode cannot commit a transaction if it cannot write successfully to a quorum of the JournalNodes.
- When restarting an HA cluster, the steps for initializing JournalNodes and NN2 can be skipped.
- Start the services in the following order:

  1. JournalNodes
  2. NameNodes

     Note:

     Verify that the ZKFailoverController (ZKFC) process on each node is running so that one of the NameNodes can be converted to active state.
  3. DataNodes
- In a NameNode HA cluster, the following dfsadmin command options will run only on the active NameNode:

```
-rollEdits
-setQuota
```

```
-clrQuota
-setSpaceQuota
-clrSpaceQuota
-setStoragePolicy
-getStoragePolicy
-finalizeUpgrade
-rollingUpgrade
-printTopology
-allowSnapshot <snapshotDir>
-disallowSnapshot <snapshotDir>
```

The following dfsadmin command options will run on both the active and standby NameNodes:

```
-safemode enter
-saveNamespace
-restoreFailedStorage
-refreshNodes
-refreshServiceAcl
-refreshUserToGroupsMappings
-refreshSuperUserGroupsConfiguration
-refreshCallQueue
-metasave
-setBalancerBandwidth
```

The -refresh <host:ipc_port> <key> arg1..argn command will be sent to the corresponding host according to its command arguments.

The -fetchImage <local directory> command attempts to identify the active NameNode through a RPC call, and then fetch the fsimage from that NameNode. This means that usually the fsimage is retrieved from the active NameNode, but it is not guaranteed because a failover can happen between the two operations.

The following dfsadmin command options are sent to the DataNodes:

```
-refreshNamenodes
-deleteBlockPool
-shutdownDatanode <datanode_host:ipc_port> upgrade
-getDatanodeInfo <datanode_host:ipc_port>
```

Configuring and Deploying NameNode Automatic Failover

Automatic Failover adds Zookeper Quorum and ZK FailoverController process components to an HDFS deployment.

The preceding sections describe how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby NameNode, even if the active node has failed. This section describes how to configure and deploy automatic failover.

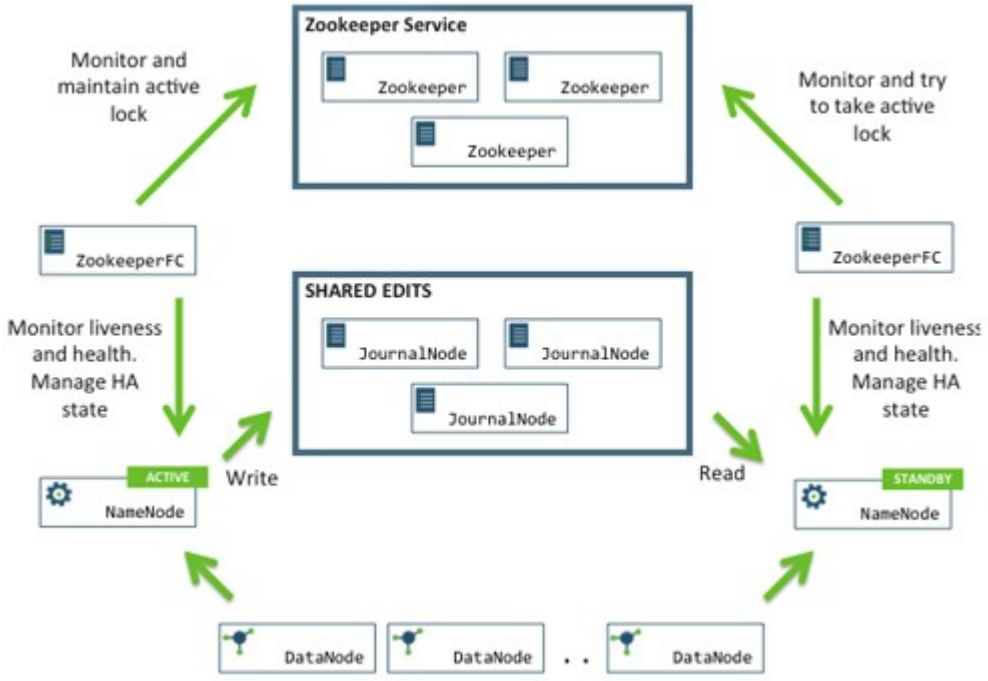
Automatic failover adds following components to an HDFS deployment

- ZooKeeper quorum
- ZKFailoverController process (abbreviated as ZKFC).

The ZKFailoverController (ZKFC) is a ZooKeeper client that monitors and manages the state of the NameNode. Each of the machines which run NameNode service also runs a ZKFC. ZKFC is responsible for:

- Health monitoring: ZKFC periodically pings its local NameNode with a health-check command.
- ZooKeeper session management: When the local NameNode is healthy, the ZKFC holds a session open in ZooKeeper. If the local NameNode is active, it also holds a special "lock" znode. This lock uses ZooKeeper's support for "ephemeral" nodes; if the session expires, the lock node will be automatically deleted.
- ZooKeeper-based election: If the local NameNode is healthy and no other node currently holds the lock znode, ZKFC will try to acquire the lock. If ZKFC succeeds, then it has "won the election" and will be responsible for running a failover to make its local NameNode active. The failover process is similar to the manual failover

described above: first, the previous active is fenced if necessary and then the local NameNode transitions to active state.



Prerequisites for Configuring NameNode Automatic Failover

Make sure you have a working Zookeeper service and you shut down your HA cluster before you try to configure and deploy automatic failover.

Complete the following prerequisites:

- Make sure that you have a working ZooKeeper service. If you have an Ambari-deployed HDP cluster with ZooKeeper, you can use that.

  Note:

  In a typical deployment, ZooKeeper daemons are configured to run on three or five nodes. It is however acceptable to co-locate the ZooKeeper nodes on the same hardware as the HDFS NameNode and Standby Node. Many operators choose to deploy the third ZooKeeper process on the same node as the YARN ResourceManager. To achieve performance and improve isolation, Hortonworks recommends configuring the ZooKeeper nodes such that the ZooKeeper data and HDFS metadata is stored on separate disk drives.
- Shut down your HA cluster (configured for manual failover).

  Currently, you cannot transition from a manual failover setup to an automatic failover setup while the cluster is running.

Configure and Deploy Automatic Failover

Configure automatic failover, initialize HA state in Zookeeper, and start the nodes in the cluster.

Procedure

1. Configure automatic failover.
   - Set up your cluster for automatic failover. Add the following property to the hdfs-site.xml file for all of the NameNode machines:

```
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
```

```
 </property>
```

- List the host-port pairs running the ZooKeeper service. Add the following property to the core-site.xml file for all of the NameNode machines:

```
<property>
 <name>ha.zookeeper.quorum</name>
 <value>zk1.example.com:2181,zk2.example.com:2181,
    zk3.example.com:2181</value>
</property>
```

Note:

Suffix the configuration key with the nameservice ID to configure the above settings on a per-nameservice basis. For example, in a cluster with federation enabled, you can explicitly enable automatic failover for only one of the nameservices by setting dfs.ha.automatic-failover.enabled.$my-nameservice-id.

2. Initialize HA state in ZooKeeper.

   Execute the following command on NN1:

```
hdfs zkfc -formatZK -force
```

   This command creates a znode in ZooKeeper. The automatic failover system stores uses this znode for data storage.

3. Check to see if ZooKeeper is running. If not, start ZooKeeper by executing the following command on the ZooKeeper host machines.

```
su - zookeeper -c "export ZOOCFGDIR=/usr/hdp/current/zookeeper-server/
conf ; export ZOOCFG=zoo.cfg; source /usr/hdp/current/zookeeper-server/
conf/zookeeper-env.sh ; /usr/hdp/current/zookeeper-server/bin/zkServer.sh
 start"
```

4. Start the JournalNodes, NameNodes, and DataNodes using the instructions provided in the Controlling HDP Services Manually chapter of the *HDP Administration Guide*.
5. Start the ZooKeeper Failover Controller (ZKFC) by executing the following command:

```
su -l hdfs -c "/usr/hdp/current/hadoop-hdfs-namenode/../hadoop/sbin/
hadoop-daemon.sh start zkfc"
```

   The sequence of starting ZKFC determines which NameNode will become Active. For example, if ZKFC is started on NN1 first, it will cause NN1 to become Active.

Note:

To convert a non-HA cluster to an HA cluster, Hortonworks recommends that you run the bootstrapStandby command (this command is used to initialize NN2) before you start ZKFC on any of the NameNode machines.

6. Verify automatic failover.

   a. Locate the Active NameNode.

      Use the NameNode web UI to check the status for each NameNode host machine.

   b. Cause a failure on the Active NameNode host machine.

      For example, you can use the following command to simulate a JVM crash:

```
kill -9 $PID_of_Active_NameNode
```

      Or, you could power cycle the machine or unplug its network interface to simulate outage.

   c. The Standby NameNode should now automatically become Active within several seconds.

Note:

The amount of time required to detect a failure and trigger a failover depends on the configuration of ha.zookeeper.session-timeout.ms property (default value is 5 seconds).

d. If the test fails, your HA settings might be incorrectly configured.

Check the logs for the zkfc daemons and the NameNode daemons to diagnose the issue.

Related Concepts

Deploying the NameNode HA Cluster

Configure Oozie Failover

Set up the database and configure Oozie on two or more servers and start the Oozie servers.

Procedure

1. Set up your database for High Availability. (For details, see the documentation for your Oozie database.)

   Oozie database configuration properties may need special configuration. (For details, see the JDBC driver documentation for your database.)
2. Configure Oozie identically on two or more servers.
3. Set the OOZIE_HTTP_HOSTNAME variable in oozie-env.sh to the Load Balancer or Virtual IP address.
4. Start all Oozie servers.
5. Use either a Virtual IP Address or Load Balancer to direct traffic to Oozie servers.
6. Access Oozie via the Virtual IP or Load Balancer address.

Related Tasks

Deploying Oozie with an HA Cluster

Related Information

Apache ZooKeeper

Administrative Commands

The subcommands of hdfs haadmin are extensively used for administering an HA cluster.

Running the hdfs haadmin command without any additional arguments will display the following usage information:

```
Usage: HAAdmin [-ns <nameserviceId>]
  [-transitionToActive <serviceId>]
  [-transitionToStandby <serviceId>]
  [-failover [--forcefence] [--forceactive] <serviceId> <serviceId>]
  [-getServiceState <serviceId>]
  [-checkHealth <serviceId>]
  [-help <command>
```

This section provides high-level uses of each of these subcommands.

- transitionToActive and transitionToStandby: Transition the state of the given NameNode to Active or Standby.

  These subcommands cause a given NameNode to transition to the Active or Standby state, respectively. These commands do not attempt to perform any fencing, and thus should be used rarely. Instead, Hortonworks recommends using the following subcommand:

  ```
  hdfs haadmin -failover
  ```
- failover: Initiate a failover between two NameNodes.

  This subcommand causes a failover from the first provided NameNode to the second.

  - If the first NameNode is in the Standby state, this command transitions the second to the Active state without error.

    - If the first NameNode is in the Active state, an attempt will be made to gracefully transition it to the Standby state. If this fails, the fencing methods (as configured by dfs.ha.fencing.methods) will be attempted in order until one succeeds. Only after this process will the second NameNode be transitioned to the Active state. If the fencing methods fail, the second NameNode is not transitioned to Active state and an error is returned.
- getServiceState: Determine whether the given NameNode is Active or Standby.

  This subcommand connects to the provided NameNode, determines its current state, and prints either "standby" or "active" to STDOUT appropriately. This subcommand might be used by cron jobs or monitoring scripts.
- checkHealth: Check the health of the given NameNode.

  This subcommand connects to the NameNode to check its health. The NameNode is capable of performing some diagnostics that include checking if internal services are running as expected. This command will return 0 if the NameNode is healthy else it will return a non-zero code.

  Note:

  This subcommand is in implementation phase and currently always returns success unless the given NameNode is down.

Configuring ResourceManager High Availability

You can configure ResourceManager High Availability to avoid windows of cluster downtime.

This guide provides instructions on setting up the ResourceManager (RM) High Availability (HA) feature in a HDFS cluster. The Active and Standby ResourceManagers embed the ZooKeeper-based ActiveStandbyElector to determine which ResourceManager should be active.

Note:

This guide assumes that an existing HDP cluster has been manually installed and deployed. It provides instructions on how to manually enable ResourceManager HA on top of the existing cluster.

The ResourceManager is a single point of failure (SPOF) in an HDFS cluster. Each cluster has a single ResourceManager, and if that machine or process become unavailable, the entire cluster will be unavailable until the ResourceManager is either restarted or started on a separate machine. This situation impacts the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster will be unavailable until an operator restarts the ResourceManager.
- Planned maintenance events such as software or hardware upgrades on the ResourceManager machine result in windows of cluster downtime.

The ResourceManager HA feature addresses these problems. This feature enables you to run redundant ResourceManagers in the same cluster in an Active/Passive configuration with a hot standby. This mechanism thus facilitates either a fast failover to the standby ResourceManager during machine crash, or a graceful administrator-initiated failover during planned maintenance.

Preparing the Hardware Resources

Make sure that you prepare the hardware resources before configuring the Resource Manager for High Availability.

- ResourceManager machines: The machines where you run Active and Standby ResourceManagers should have exactly the same hardware. For recommended hardware for ResourceManagers, see Hardware for Master Nodes in the Cluster Planning Guide.
- ZooKeeper machines: For automated failover functionality, there must be an existing ZooKeeper cluster available. The ZooKeeper service nodes can be co-located with other Hadoop daemons.

Deploying ResourceManager HA Cluster

You must first configure manual or automatic ResourceManager failover and then deploy the ResourceManager HA cluster.

HA configuration is backward-compatible and works with your existing single ResourceManager configuration.

Configuring Manual or Automatic ResourceManager Failover

Set common ResourceManager HA properties and other properties in yarn-site.xml file.

Complete the following prerequisites:

- Make sure that you have a working ZooKeeper service. If you have an Ambari-deployed HDP cluster with ZooKeeper, you can use that ZooKeeper service.

  Note:

  In a typical deployment, ZooKeeper daemons are configured to run on three or five nodes. It is, however, acceptable to co-locate the ZooKeeper nodes on the same hardware as the HDFS NameNode and Standby Node. Many operators choose to deploy the third ZooKeeper process on the same node as the YARN ResourceManager. To achieve performance and improve isolation, Hortonworks recommends configuring the ZooKeeper nodes such that the ZooKeeper data and HDFS metadata is stored on separate disk drives.
- Shut down the cluster.

Set Common ResourceManager HA Properties

The following properties are required for both manual and automatic ResourceManager HA. Add these properties to the etc/hadoop/conf/yarn-site.xml file:

Property Name	Recommended Value	Description
yarn.resourcemanager. ha.enabled	true	Enable RM HA
yarn.resourcemanager. ha.rm-ids	Cluster-specific, e.g., rm1,rm2	A comma-separated list of ResourceManager IDs in the cluster.
yarn.resourcemanager. hostname.<rm-id>	Cluster-specific	The host name of the ResourceManager. Must be set for all RMs.
yarn.resourcemanager. recovery.enabled	true	Enable job recovery on RM restart or failover.
yarn.resourcemanager. store.class	org.apache.hadoop.yarn. server.resourcemanager. recovery.ZKRMStateStore	The RMStateStore implementation to use to store the ResourceManager's internal state. The ZooKeeper- based store supports fencing implicitly, i.e., allows a single ResourceManager to make multiple changes at a time, and hence is recommended.
yarn.resourcemanager .zk-address	Cluster-specific	The ZooKeeper quorum to use to store the ResourceManager's internal state. For multiple ZK servers, use commas to separate multiple ZK servers.
yarn.client.failover-proxy-provider	org.apache.hadoop.yarn. client. ConfiguredRMFailover ProxyProvider	When HA is enabled, the class to be used by Clients, AMs and NMs to failover to the Active RM. It should extend org.apache.hadoop.yarn. client.RMFailoverProxyProvider This is an optional configuration. The default value is “org.apache.hadoop.yarn.client. ConfiguredRMFailoverProxyProvider”

Note:

You can also set values for each of the following properties in yarn-site.xml:

```
yarn.resourcemanager.address.<rm#id>
yarn.resourcemanager.scheduler.address.<rm#id>
yarn.resourcemanager.admin.address.<rm#id>
yarn.resourcemanager.resource#tracker.address.<rm#id>
```

```
yarn.resourcemanager.webapp.address.<rm#id>
```

If these addresses are not explicitly set, each of these properties will use

```
                                yarn.resourcemanager.hostname.<rm-
id>:default_port
```

such as DEFAULT_RM_PORT, DEFAULT_RM_SCHEDULER_PORT, etc.

The following is a sample yarn-site.xml file with these common ResourceManager HA properties configured:

```
<!-- RM HA Configurations-->

<property>
 <name>yarn.resourcemanager.ha.enabled</name>
 <value>true</value>
</property>

<property>
 <name>yarn.resourcemanager.ha.rm-ids</name>
 <value>rm1,rm2</value>
</property>

<property>
 <name>yarn.resourcemanager.hostname.rm1</name>
 <value>${rm1 address}</value>
</property>

<property>
 <name>yarn.resourcemanager.hostname.rm2</name>
 <value>${rm2 address}</value>
</property>

<property>
 <name>yarn.resourcemanager.webapp.address.rm1</name>
 <value>rm1_web_address:port_num</value>
 <description>We can set rm1_web_address separately.
   If not, it will use
   ${yarn.resourcemanager.hostname.rm1}:DEFAULT_RM_WEBAPP_PORT
   </description>
</property>

<property>
 <name>yarn.resourcemanager.webapp.address.rm2</name>
 <value>rm2_web_address:port_num</value>
</property>

<property>
 <name>yarn.resourcemanager.recovery.enabled</name>
 <value>true</value>
</property>

<property>
 <name>yarn.resourcemanager.store.class</name>
 <value>org.apache.hadoop.yarn.server.resourcemanager.recovery.
    ZKRMStateStore</value>
</property>

<property>
 <name>yarn.resourcemanager.zk-address</name>
 <value>${zk1.address,zk2.address}</value>
</property>
```

```
<property>
 <name>yarn.client.failover-proxy-provider</name>
 <value>org.apache.hadoop.yarn.client.
    ConfiguredRMFailoverProxyProvider</value>
</property>
```

Configure Manual ResourceManager Failover

Automatic ResourceManager failover is enabled by default, so it must be disabled for manual failover.

To configure manual failover for ResourceManager HA, add the yarn.resourcemanager.ha.automatic-failover.enabled configuration property to the etc/hadoop/conf/yarn-site.xml file, and set its value to "false":

```
<property>
 <name>yarn.resourcemanager.ha.automatic-failover.enabled</name>
 <value>false</value>
</property>
```

Configure Automatic ResourceManager Failover

The preceding section described how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby ResourceManager, even if the active node has failed. This section describes how to configure automatic failover.

1. Add the following configuration options to the yarn-site.xml file:

Property Name	Recommended Value	Description
yarn.resourcemanager.ha. automatic-failover.zk-base-path	/yarn-leader-election	The base znode path to use for storing leader information, when using ZooKeeper-based leader election. This is an optional configuration. The default value is /yarn-leader-election
yarn.resourcemanager. cluster-id	yarn-cluster	The name of the cluster. In a HA setting, this is used to ensure the RM participates in leader election for this cluster, and ensures that it does not affect other clusters.

   Example:

```
<property>
 <name>yarn.resourcemanager.ha.automatic-failover.zk-base-path</name>
 <value>/yarn-leader-election</value>
 <description>Optional setting. The default value is
   /yarn-leader-election</description>
</property>

<property>
 <name>yarn.resourcemanager.cluster-id</name>
 <value>yarn-cluster</value>
</property>
```

2. Automatic ResourceManager failover is enabled by default.

   If you previously configured manual ResourceManager failover by setting the value of yarn.resourcemanager.ha.automatic-failover.enabled to "false", you must delete this property to return automatic failover to its default enabled state.

Deploying the ResourceManager HA Cluster

Update the yarn-site.xml file and configuration files and start Zookeeper, HDFS, and YARN in that order.

Procedure

1. Copy the etc/hadoop/conf/yarn-site.xml file from the primary ResourceManager host to the standby ResourceManager host.
2. Make sure that the clientPort value set in etc/zookeeper/conf/zoo.cfg matches the port set in the following yarn-site.xml property:

```
<property>
 <name>yarn.resourcemanager.zk-state-store.address</name>
 <value>localhost:2181</value>
</property>
```

3. Start ZooKeeper. Execute this command on the ZooKeeper host machines:

```
su - zookeeper -c "export ZOOCFGDIR=/usr/hdp/current/zookeeper-server/
conf ; export ZOOCFG=zoo.cfg; source /usr/hdp/current/zookeeper-server/
conf/zookeeper-env.sh ; /usr/hdp/current/zookeeper-server/bin/zkServer.sh
 start"
```

4. Start HDFS.
5. Start YARN.
6. Set the active ResourceManager:

   MANUAL FAILOVER ONLY: If you configured manual ResourceManager failover, you must transition one of the ResourceManagers to Active mode. Execute the following CLI command to transition ResourceManager "rm1" to Active:

```
yarn rmadmin -transitionToActive rm1
```

   You can use the following CLI command to transition ResourceManager "rm1" to Standby mode:

```
yarn rmadmin -transitionToStandby rm1
```

   AUTOMATIC FAILOVER: If you configured automatic ResourceManager failover, no action is required -- the Active ResourceManager will be chosen automatically.
7. Start all remaining unstarted cluster services.

Minimum Settings for Automatic ResourceManager HA Configuration

Consider the minimum yarn-site.xml configuration settings for ResourceManager HA with automatic failover.

The minimum yarn-site.xml configuration settings for ResourceManager HA with automatic failover are as follows:

```
<property>
 <name>yarn.resourcemanager.ha.enabled</name>
 <value>true</value>
</property>

<property>
 <name>yarn.resourcemanager.ha.rm-ids</name>
 <value>rm1,rm2</value>
</property>

<property>
 <name>yarn.resourcemanager.hostname.rm1</name>
 <value>192.168.1.9</value>
</property>

<property>
 <name>yarn.resourcemanager.hostname.rm2</name>
 <value>192.168.1.10</value>
</property>
```

```
<property>
 <name>yarn.resourcemanager.recovery.enabled</name>
 <value>true</value>
</property>

<property>
 <name>yarn.resourcemanager.store.class</name>
 <value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</
value>
</property>

<property>
 <name>yarn.resourcemanager.zk-address</name>
 <value>192.168.1.9:2181,192.168.1.10:2181</value>
 <description>For multiple zk services, separate them with comma</
description>
</property>

<property>
 <name>yarn.resourcemanager.cluster-id</name>
 <value>yarn-cluster</value>
</property>
```

Testing ResourceManager HA on a Single Node

Enable ResourceManagers to launch and explicitly set values specific to each ResourceManager separately yarn-site.xml file.

Test ResourceManager High Availability on a single node

To test ResourceManager HA on a single node (launch more than one ResourceManager on a single node), you need to add the following settings in yarn-site.xml.

To enable ResourceManager "rm1" to launch:

```
<property>
 <name>yarn.resourcemanager.ha.id</name>
 <value>rm1</value>
 <description>If we want to launch more than one RM in single node, we need
 this configuration</description>
</property>
```

To enable ResourceManager rm2 to launch:

```
<property>
 <name>yarn.resourcemanager.ha.id</name>
 <value>rm2</value>
 <description>If we want to launch more than one RM in single node, we need
 this configuration</description>
</property>
```

You should also explicitly set values specific to each ResourceManager for the following properties separately in yarn-site.xml:

- yarn.resourcemanager.address.<rm-id>
- yarn.resourcemanager.scheduler.address.<rm-id>
- yarn.resourcemanager.admin.address.<rm-id>
- yarn.resourcemanager.resource#tracker.address.<rm-id>
- yarn.resourcemanager.webapp.address.<rm-id>

For example:

```
<!-- RM1 Configs -->

<property>
 <name>yarn.resourcemanager.address.rm1</name>
 <value>localhost:23140</value>
</property>

<property>
 <name>yarn.resourcemanager.scheduler.address.rm1</name>
 <value>localhost:23130</value>
</property>

<property>
 <name>yarn.resourcemanager.webapp.address.rm1</name>
 <value>localhost:23188</value>
</property>

<property>
 <name>yarn.resourcemanager.resource-tracker.address.rm1</name>
 <value>localhost:23125</value>
</property>

<property>
 <name>yarn.resourcemanager.admin.address.rm1</name>
 <value>localhost:23141</value>
</property>

<!-- RM2 configs -->
<property>
 <name>yarn.resourcemanager.address.rm2</name>
 <value>localhost:33140</value>
</property>

<property>
 <name>yarn.resourcemanager.scheduler.address.rm2</name>
 <value>localhost:33130</value>
</property>

<property>
 <name>yarn.resourcemanager.webapp.address.rm2</name>
 <value>localhost:33188</value>
</property>

<property>
 <name>yarn.resourcemanager.resource-tracker.address.rm2</name>
 <value>localhost:33125</value>
</property>

<property>
 <name>yarn.resourcemanager.admin.address.rm2</name>
 <value>localhost:33141</value>
</property>
```

Deploying Hue with a ResourceManager HA Cluster

Add the HA configuration sub-section at the end of the yarn_clusters section of the hue_ini configuration file.

Add the HA sub-section

If it does not exist, you can add a [[[ha]]] configuration sub-section to end of the [[yarn_clusters]] section of the hue.ini configuration file.

To do this, specify the following:

- resourcemanager_api_url

  Host on which you are running the failover Resource Manager.
- logical_name

  Logical name of the Resource Manager.
- submit_to

  Specify if Hue should use the cluster. Specify true to use the cluster, or false to not use the cluster.

If you ensure that the cluster section names are unique, you can add more than one Resource Manager failover configuration, for example, [[[ha2]]]. By doing so, if the standby Resource Manager also fails over, the next available RM instance is tried.

Example:

```
[[yarn_clusters]]

    [[[default]]]
      # Whether to submit jobs to this cluster
      submit_to=true

      ## security_enabled=false

      # Resource Manager logical name (required for HA)
      logical_name=rm1

      # URL of the ResourceManager webapp address
 (yarn.resourcemanager.webapp.address)
      resourcemanager_api_url=http://cl2-node02.local:8088

      # URL of Yarn RPC adress (yarn.resourcemanager.address)
      resourcemanager_rpc_url=http://cl2-node02.local:8050

      # URL of the ProxyServer API
      proxy_api_url=http://cl2-node02.local:8088

      # URL of the HistoryServer API
      history_server_api_url=http://cl2-node02.local:19888

      # URL of the NodeManager API
      node_manager_api_url=http://localhost:8042

      # HA support by specifying multiple clusters
      # e.g.

      [[[ha]]]
        # Enter the host on which you are running the failover Resource
 Manager
        resourcemanager_api_url=http://cl2-node01.local:8088
        history_server_api_url=http://cl2-node02.local:19888
        proxy_api_url=http://cl2-node01.local:8088
        resourcemanager_rpc_url=http://cl2-node01.local:8050
        logical_name=rm2
        submit_to=True
Note only one ha section is necessary, since Hue still uses the default
 section.
```

Configuring Apache Ranger High Availability

You must consider the specified prerequisites and then configure High Availability (HA) for Apache Ranger.

Configuring Ranger Admin HA

You can configure Ranger Admin HA with or without SSL on an Ambari-managed cluster.

The configuration settings used in this section are sample values. You should adjust these settings to reflect your environment (folder locations, passwords, file names, and so on).

Prerequisites for Configuring Apache Ranger for High Availability

Consider truststore and security prerequisites before configuring Apache Ranger for HA.

- Copy the keystore/truststore files into a different location (e.g. /etc/security/serverKeys) than the /etc/<component>/conf folder.
- Make sure that the JKS file names are unique.
- Make sure that the correct permissions are applied.
- Make sure that passwords are secured.

Configuring Ranger Admin HA Without SSL

Set up the load-balancer and enable Ranger Admin on an Ambari-managed cluster.

Procedure

1. Use SSH to connect to the cluster node where you will set up the load-balancer. In this procedure, we use the IP address 172.22.71.37.
2. Use the following command to switch to the /usr/local directory:

```
cd /usr/local
```

3. Download the httpd file and its dependencies (apr and apr-util):

```
wget https://archive.apache.org/dist/httpd/httpd-2.4.16.tar.gz
wget https://archive.apache.org/dist/apr/apr-1.5.2.tar.gz
wget https://archive.apache.org/dist/apr/apr-util-1.5.4.tar.gz
```

4. Extract the contents of these files:

```
tar -xvf httpd-2.4.16.tar.gz
tar -xvf apr-1.5.2.tar.gz
tar -xvf apr-util-1.5.4.tar.gz
```

5. Run the following commands to move apr and apr-util to the srclib directory under httpd:

```
mv apr-1.5.2/ apr
mv apr httpd-2.4.16/srclib/
mv apr-util-1.5.4/ apr-util
mv apr-util httpd-2.4.16/srclib/
```

6. Install PCRE (Perl-Compatible Regular Expressions Library):

```
yum install pcre pcre-devel
```

Note:

Here we are using yum install, but you can also download the latest bits from http://www.pcre.org/

7. Install gcc (ANSI-C Compiler and Build System):

```
yum install gcc
```

8. Run the following commands to configure the source tree:

```
cd /usr/local/httpd-2.4.16
./configure
```

9. Run the following command to make the build:

```
make
```

10. Run the install:

```
make install
```

11. Run the following commands to confirm the preceding configuration steps:

```
cd /usr/local/apache2/bin
./apachectl start
curl localhost
```

This should return:

```
<html><body><h1>It works!</h1></body></html>
```

12. Run the following commands to create a backup conf file.

```
cd /usr/local/apache2/conf
cp httpd.conf ~/httpd.conf.backup
```

13. Edit the httpd.conf file:

```
vi /usr/local/apache2/conf/httpd.conf
```

Make the following updates:

- If you are not running the load-balancer on the default port 80, change the default listening port in line Listen 80 to match the port setting.
- Un-comment the following module entries (remove the # symbol at the beginning of each line):

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule lbmethod_bytraffic_module modules/mod_lbmethod_bytraffic.so
LoadModule lbmethod_bybusyness_module modules/mod_lbmethod_bybusyness.so
```

- Update the ServerAdmin email address, or comment out that line.

```
#ServerAdmin you@example.com
```

- At the end of the httpd.conf file, add the following line to read the custom configuration file:

```
Include conf/ranger-cluster.conf
```

14. Create a custom conf file:

```
vi ranger-cluster.conf
```

Make the following updates:

- Add the following lines, then change the <VirtualHost *:88> port to match the default port you set in the httpd.conf file in the previous step.

```
#
# This is the Apache server configuration file providing SSL support.
# It contains the configuration directives to instruct the server how to
# serve pages over an https connection. For detailing information about
 these
# directives see <URL:http://httpd.apache.org/docs/2.2/mod/mod_ssl.html>
#
# Do NOT simply read the instructions in here without understanding
# what they do.  They're here only as hints or reminders.  If you are
 unsure
# consult the online docs. You have been warned.

#Listen 80
<VirtualHost *:88>
        ProxyRequests off
        ProxyPreserveHost on

        Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e;
 path=/" env=BALANCER_ROUTE_CHANGED

        <Proxy balancer://rangercluster>
                BalancerMember http://172.22.71.38:6080 loadfactor=1
 route=1
                BalancerMember http://172.22.71.39:6080 loadfactor=1
 route=2

                Order Deny,Allow
                Deny from none
                Allow from all

                ProxySet lbmethod=byrequests scolonpathdelim=On
 stickysession=ROUTEID maxattempts=1 failonstatus=500,501,502,503
 nofailover=Off
        </Proxy>

        # balancer-manager
        # This tool is built into the mod_proxy_balancer
        # module and will allow you to do some simple
        # modifications to the balanced group via a gui
        # web interface.
        <Location /balancer-manager>
                SetHandler balancer-manager
                Order deny,allow
                Allow from all
        </Location>


       ProxyPass /balancer-manager !
       ProxyPass / balancer://rangercluster/
       ProxyPassReverse / balancer://rangercluster/

</VirtualHost>
```

Note:

The URLs listed in the BalancerMember entries are the IP addresses of the Ranger Admin hosts. In this example, the Ranger Admin host addresses are:

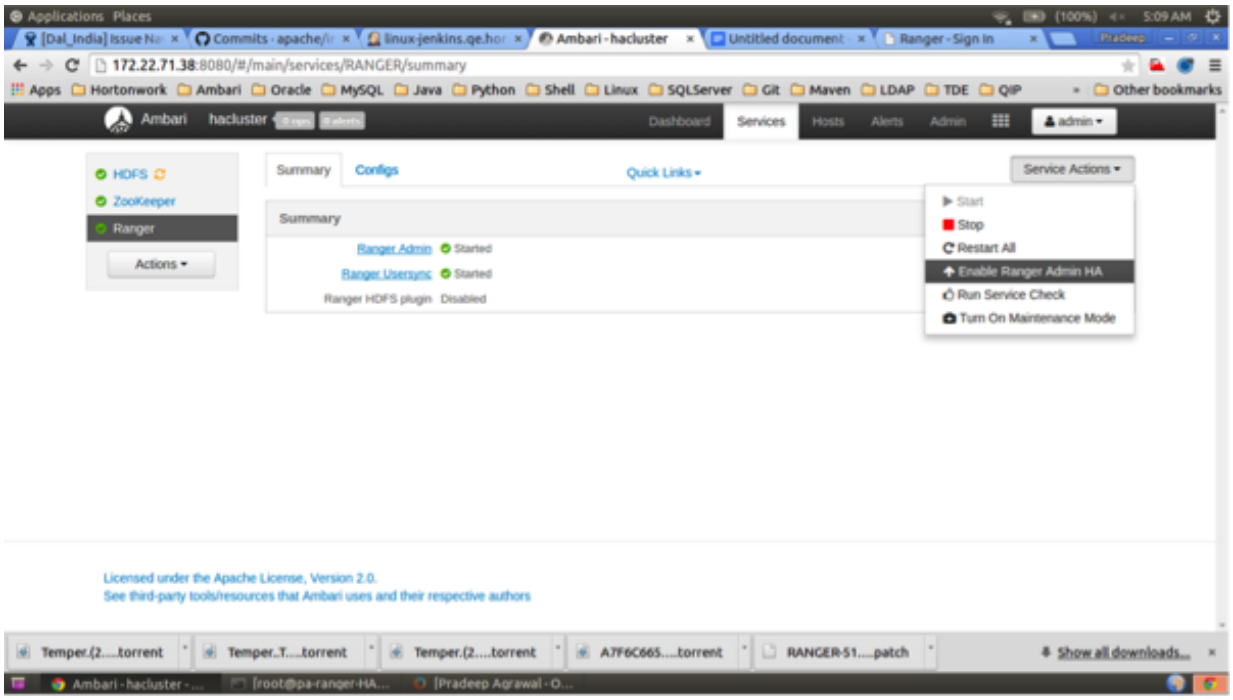
```
http://172.22.71.38:6080
http://172.22.71.39:6080
```

15. Run the following commands to restart the httpd server:

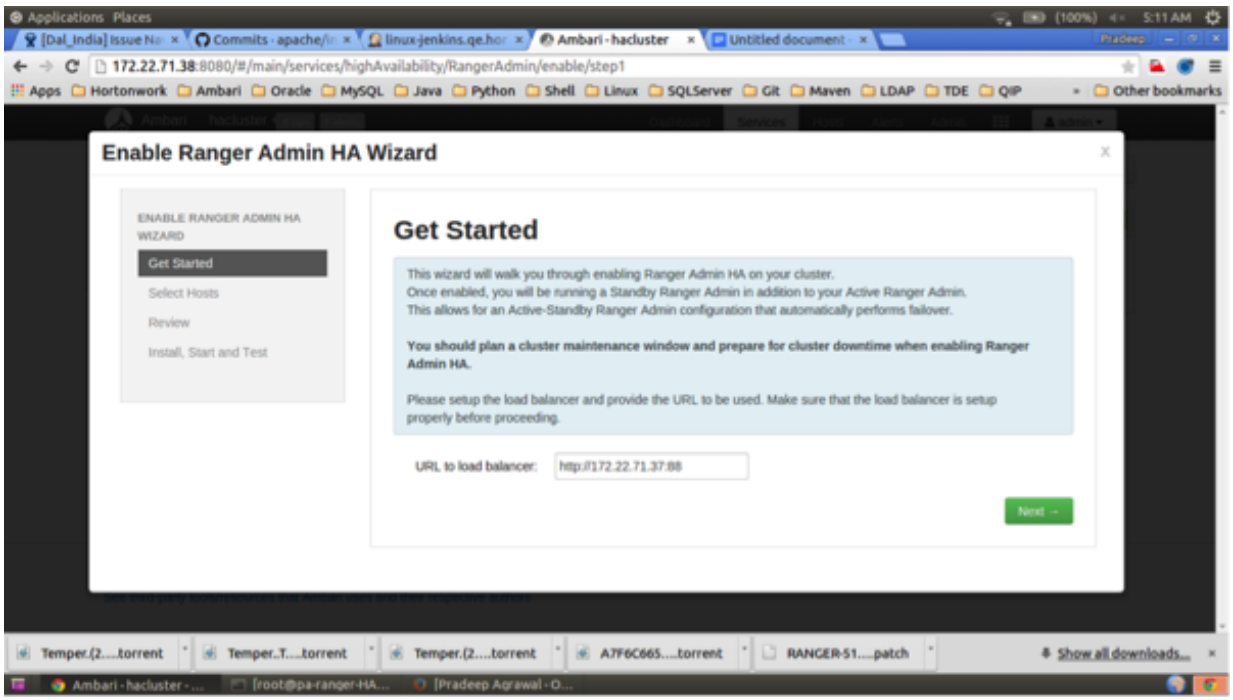
```
cd /usr/local/apache2/bin
./apachectl restart
```

If you use a browser to check the load-balancer host (with port) as specified in the BalanceMember entries in the ranger-cluster.conf file, you should see the Ranger Admin page.

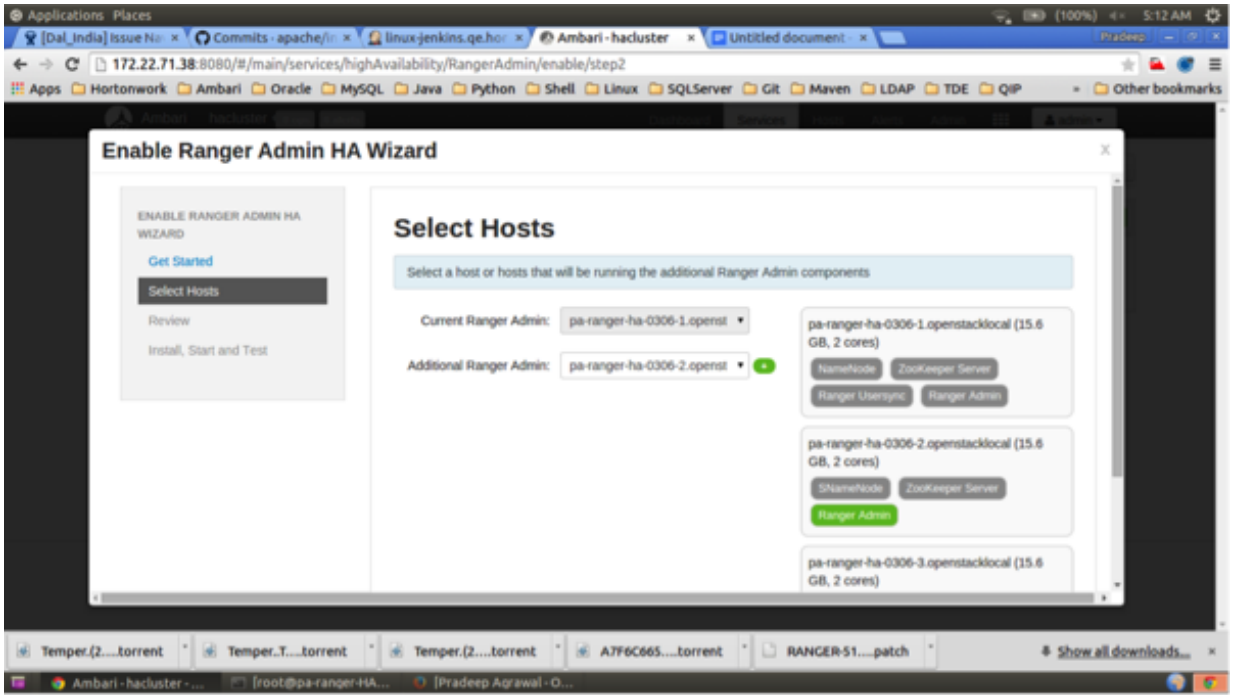
16. Enable Ranger Admin HA using Ambari. On the Ambari dashboard of the first Ranger host, select Services > Ranger, then select Service Actions > Enable Ranger Admin HA to launch the Enable Ranger Admin HA Wizard.



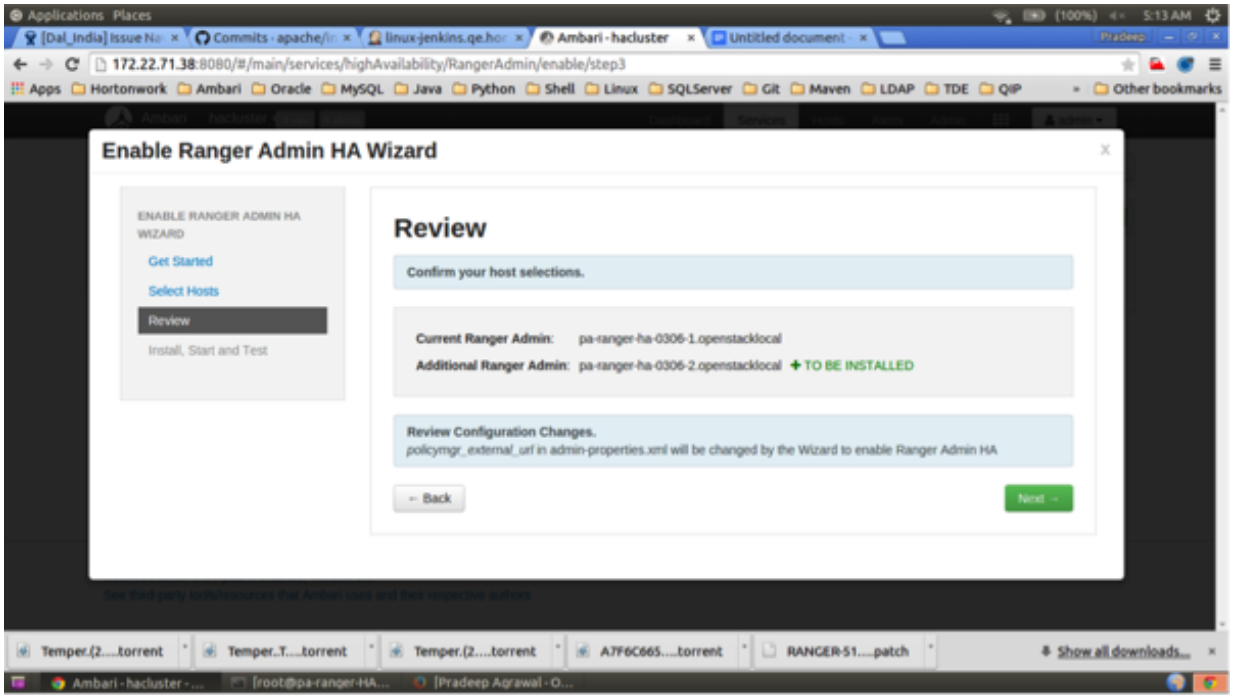
17. On the Get Started page, enter the load-balancer URL and port number (in this example, 172.22.71.37:88), then click Next.



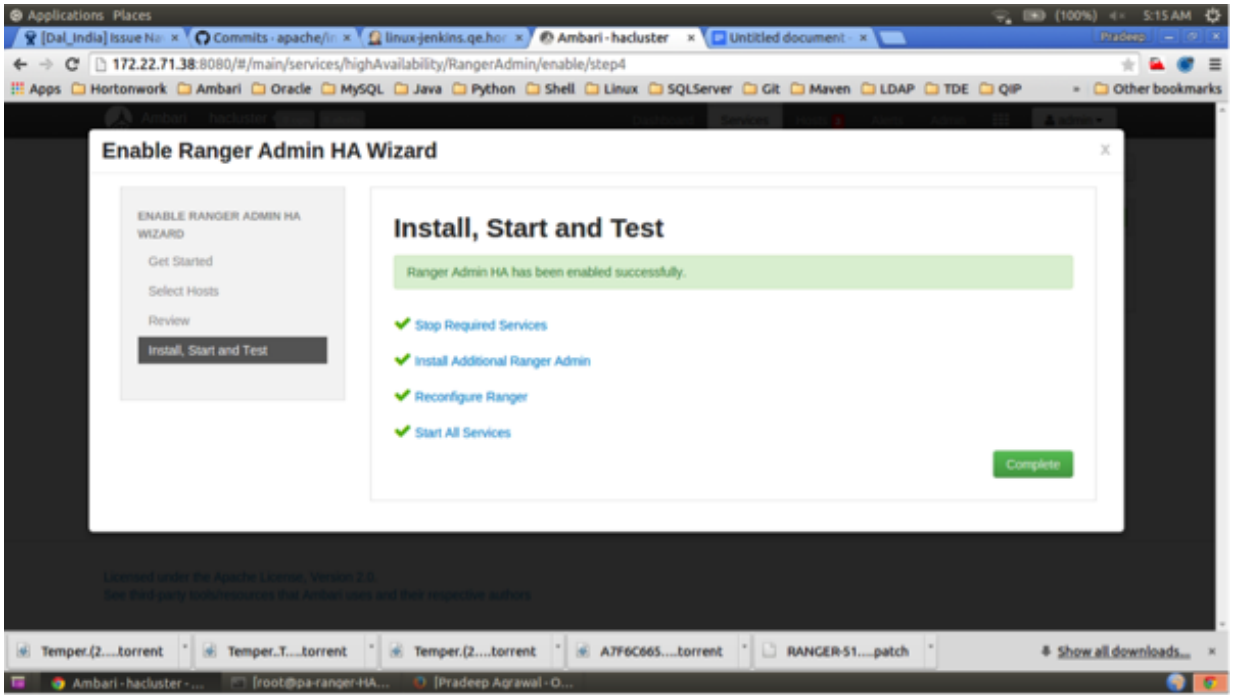
18. On the Select Hosts page, confirm the host assignments, then click Next.



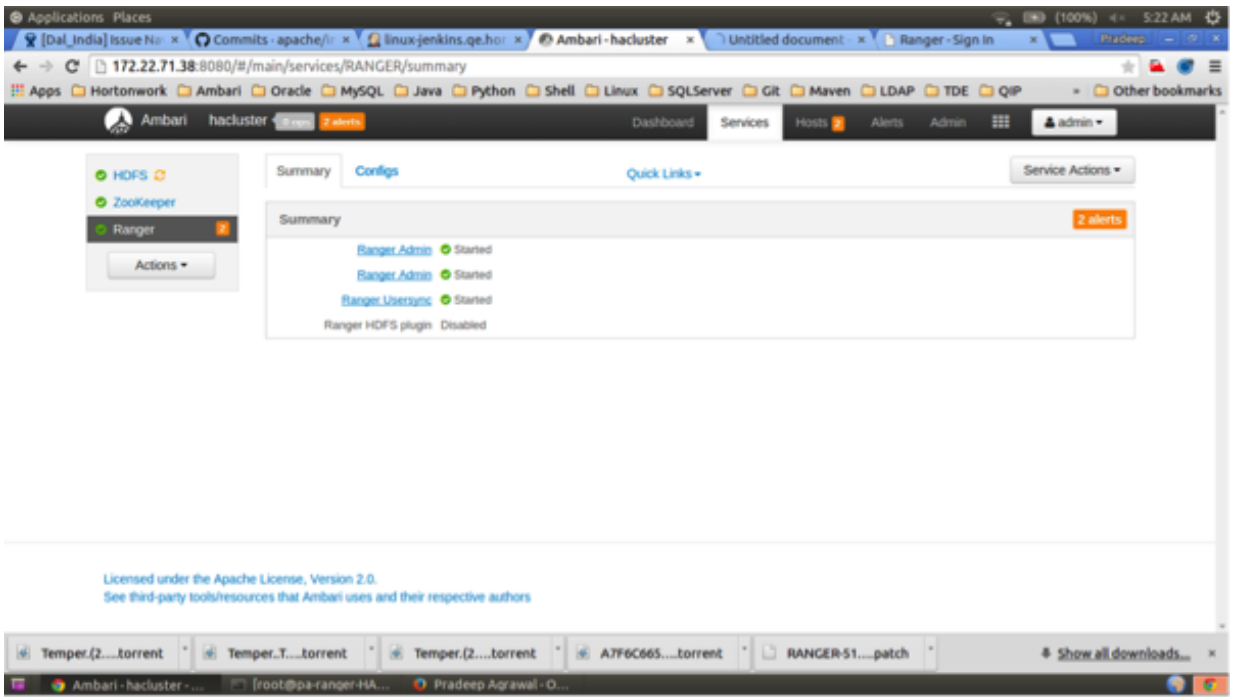
19. Check the settings on the Review page, then click Next.



20. Click Complete on the Install, Start, and Test page to complete the installation.



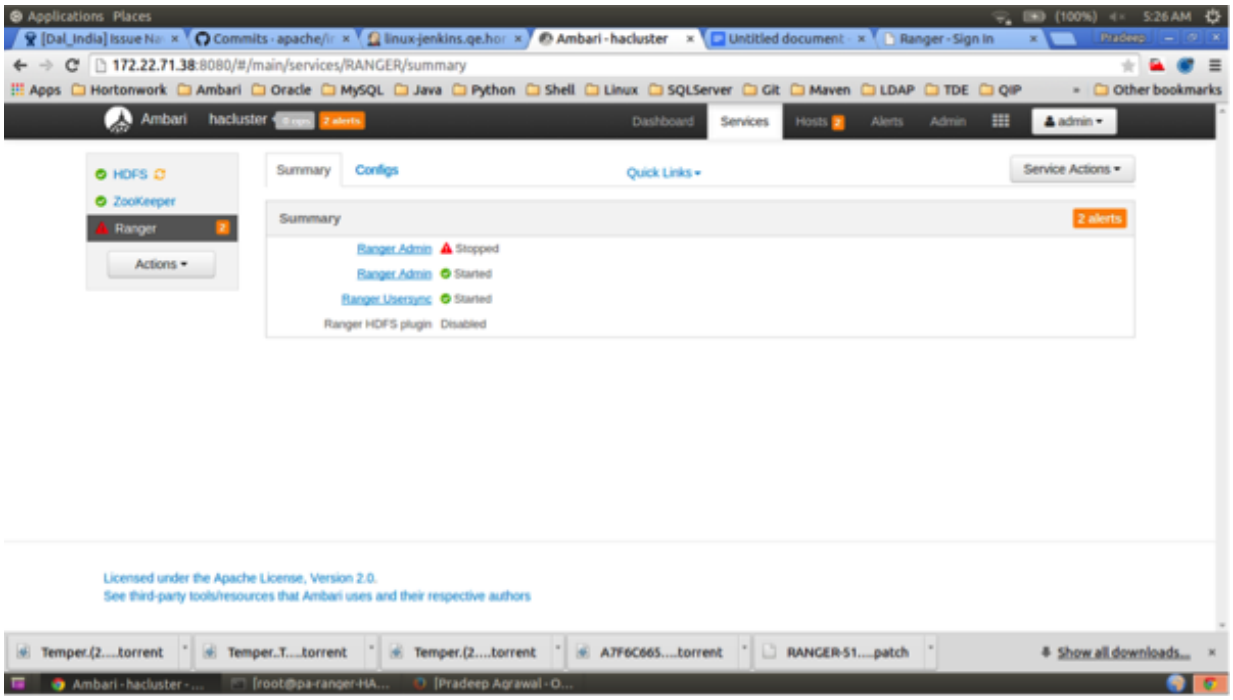
21. When the installation is complete, the Ranger Admin instances are listed on the Ranger Summary page. Select Actions > Restart All Required to restart all services that require a restart.



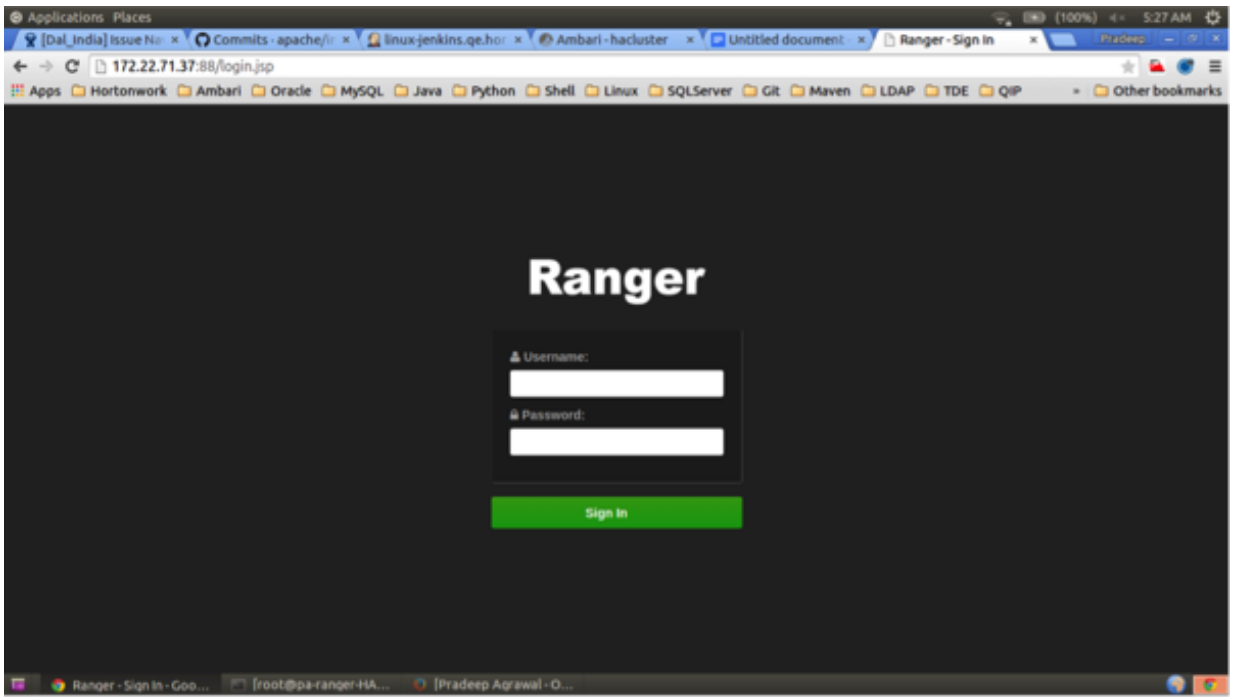
Note:

For Oracle, clear the Setup DB and DB user check box under "Advanced ranger-env" to avoid DB setup.

22. To test the load-balancer and Ranger HA configuration, select Ranger > Service Actions > Stop on one of the Ranger hosts.



23. Use a browser to check the load-balancer host URL (with port). You should see the Ranger Admin page.



Configuring Ranger Admin HA With SSL

Set up a load-balancer with SSL and enable Ranger Admin HA on an Ambari-managed cluster.

Procedure

1. Use SSH to connect to the cluster node where you will set up the load-balancer. In this procedure, we use the IP address 172.22.71.37.
2. Use the following command to switch to the /usr/local directory:

```
cd /usr/local
```

3. Download the httpd file and its dependencies (apr and apr-util):

```
wget https://archive.apache.org/dist/httpd/httpd-2.4.16.tar.gz
wget https://archive.apache.org/dist/apr/apr-1.5.2.tar.gz
wget https://archive.apache.org/dist/apr/apr-util-1.5.4.tar.gz
```

4. Extract the contents of these files:

```
tar -xvf httpd-2.4.16.tar.gz
tar -xvf apr-1.5.2.tar.gz
tar -xvf apr-util-1.5.4.tar.gz
```

5. Run the following commands to move apr and apr-util to the srclib directory under httpd:

```
mv apr-1.5.2/ apr
mv apr httpd-2.4.16/srclib/
mv apr-util-1.5.4/ apr-util
mv apr-util httpd-2.4.16/srclib/
```

6. Install the required packages:

```
yum groupinstall "Development Tools"
yum install openssl-devel
yum install pcre-devel
```

7. Run the following commands to configure the source tree:

```
cd /usr/local/httpd-2.4.16
./configure --enable-so --enable-ssl --with-mpm=prefork --with-included-
apr
```

8. Run the following command to make the build:

```
make
```

9. Run the install:

```
make install
```

10. Run the following commands to confirm the preceding configuration steps:

```
cd /usr/local/apache2/bin
./apachectl start
curl localhost
```

This should return:

```
<html><body><h1>It works!</h1></body></html>
```

11. Run the following commands to create a backup conf file.

```
cd /usr/local/apache2/conf
cp httpd.conf ~/httpd.conf.backup
```

12. Edit the httpd.conf file:

```
vi /usr/local/apache2/conf/httpd.conf
```

Make the following updates:

- If you are not running the load-balancer on the default port 80, change the default listening port in line Listen 80 to match the port setting.
- Un-comment the following module entries (remove the # symbol at the beginning of each line):

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule lbmethod_bytraffic_module modules/mod_lbmethod_bytraffic.so
LoadModule lbmethod_bybusyness_module modules/mod_lbmethod_bybusyness.so
LoadModule ssl_module modules/mod_ssl.so
```

Note:

If LoadModule ssl_module modules/mod_ssl.so is not available in the httpd.conf file, check to make sure that you performed all of the previous installation steps. The load balancer will not work properly without the SSL module.

- Update the ServerAdmin email address, or comment out that line.

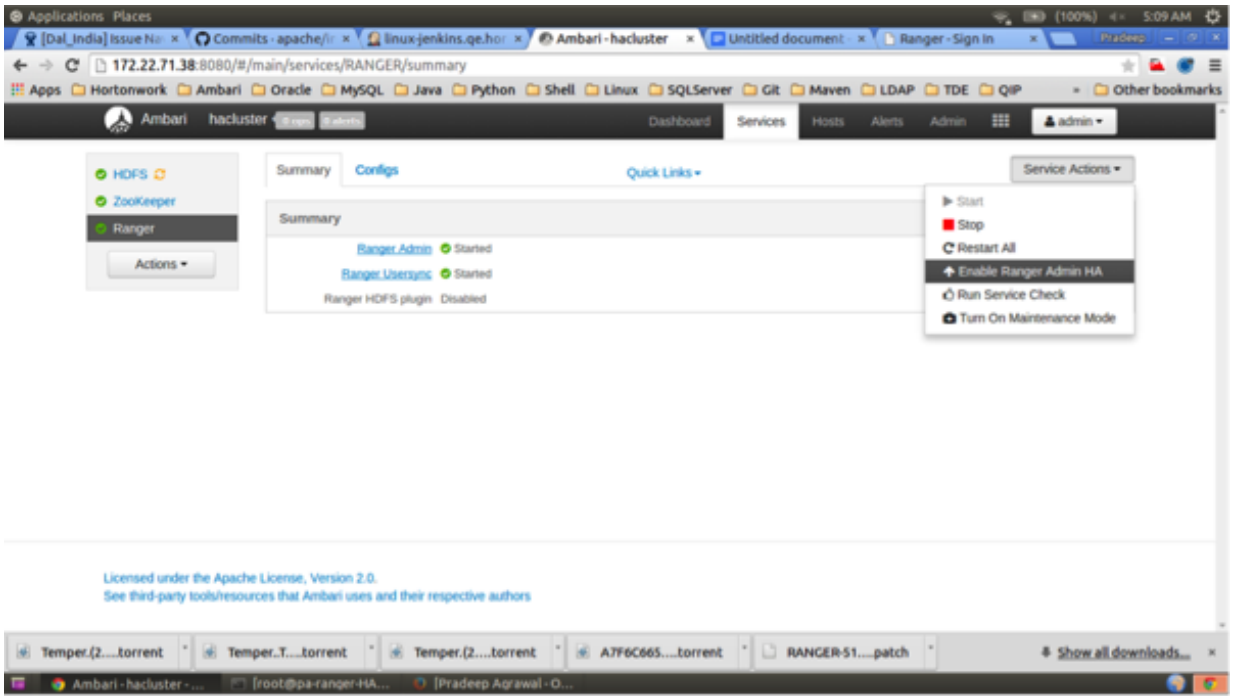
```
#ServerAdmin you@example.com
```

13. Run the following command to restart the httpd server:

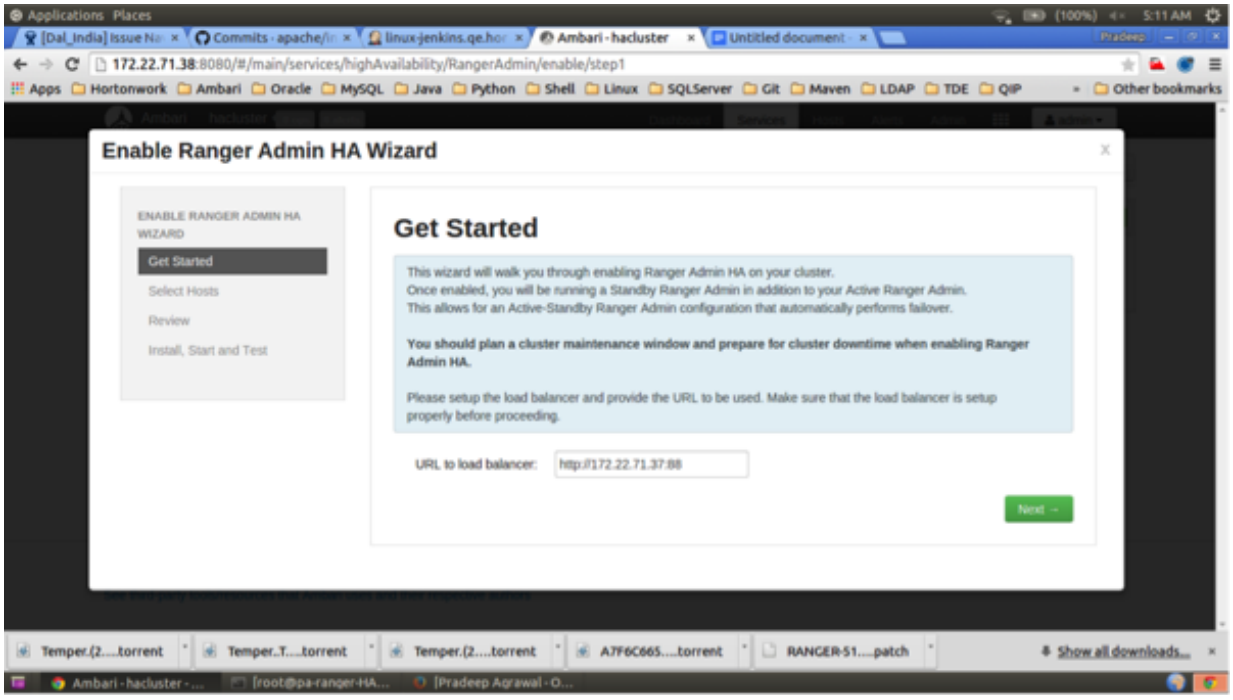
```
/usr/local/apache2/bin/apachectl restart
```

You should now be able to use Curl or a browser to access the load-balancer server IP address (with the port configured in the httpd.conf file) using the HTTPS protocol.

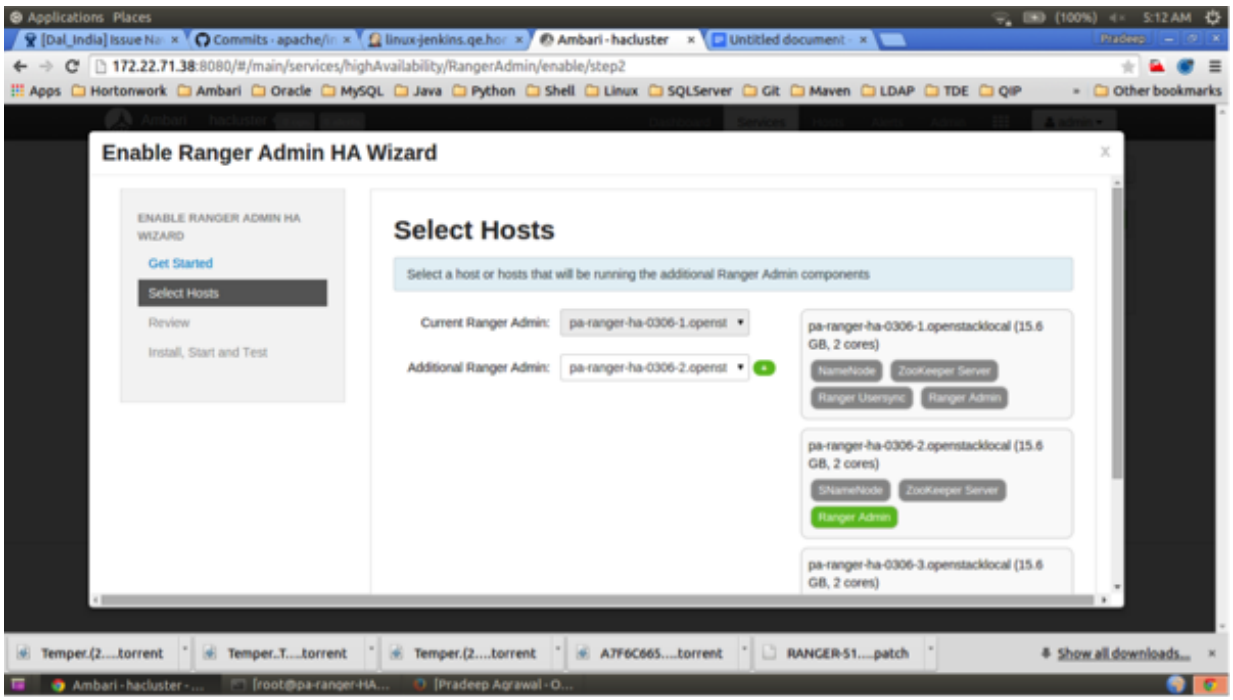
14. Enable Ranger Admin HA using Ambari. On the Ambari dashboard of the first Ranger host, select Services > Ranger, then select Service Actions > Enable Ranger Admin HA to launch the Enable Ranger Admin HA Wizard.



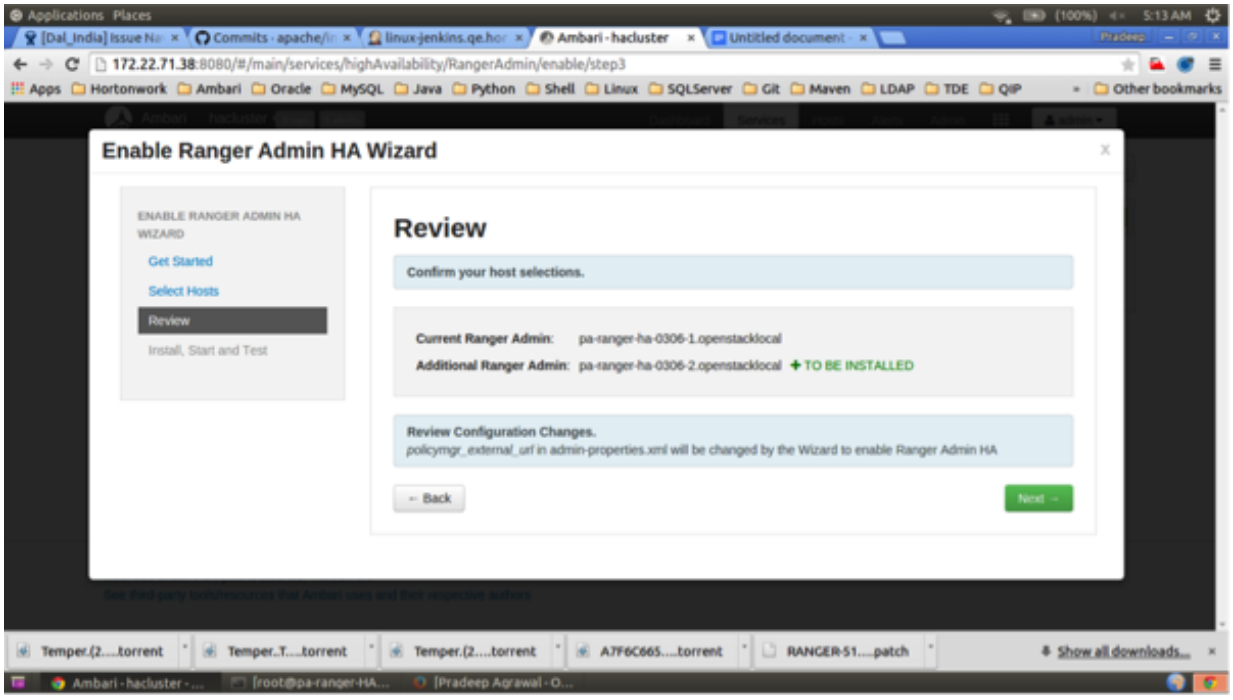
15. On the Get Started page, enter the load-balancer URL and port number (in this example, 172.22.71.37:88), then click Next.



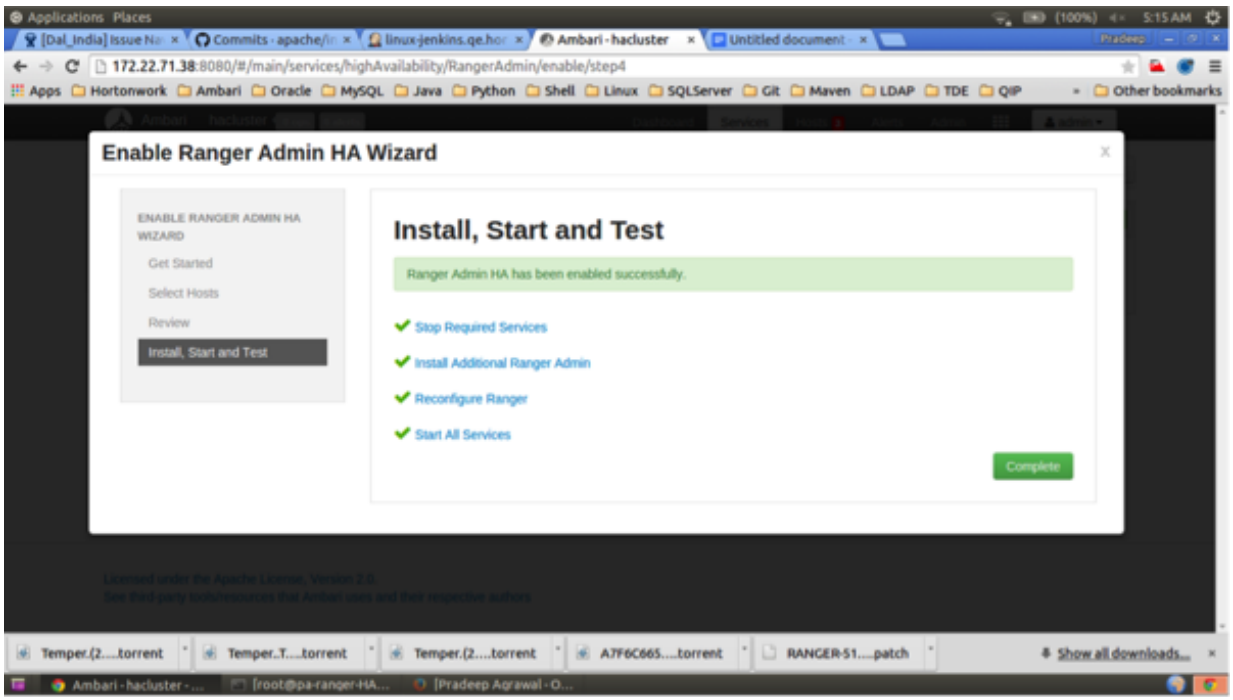
16. On the Select Hosts page, confirm the host assignments, then click Next.



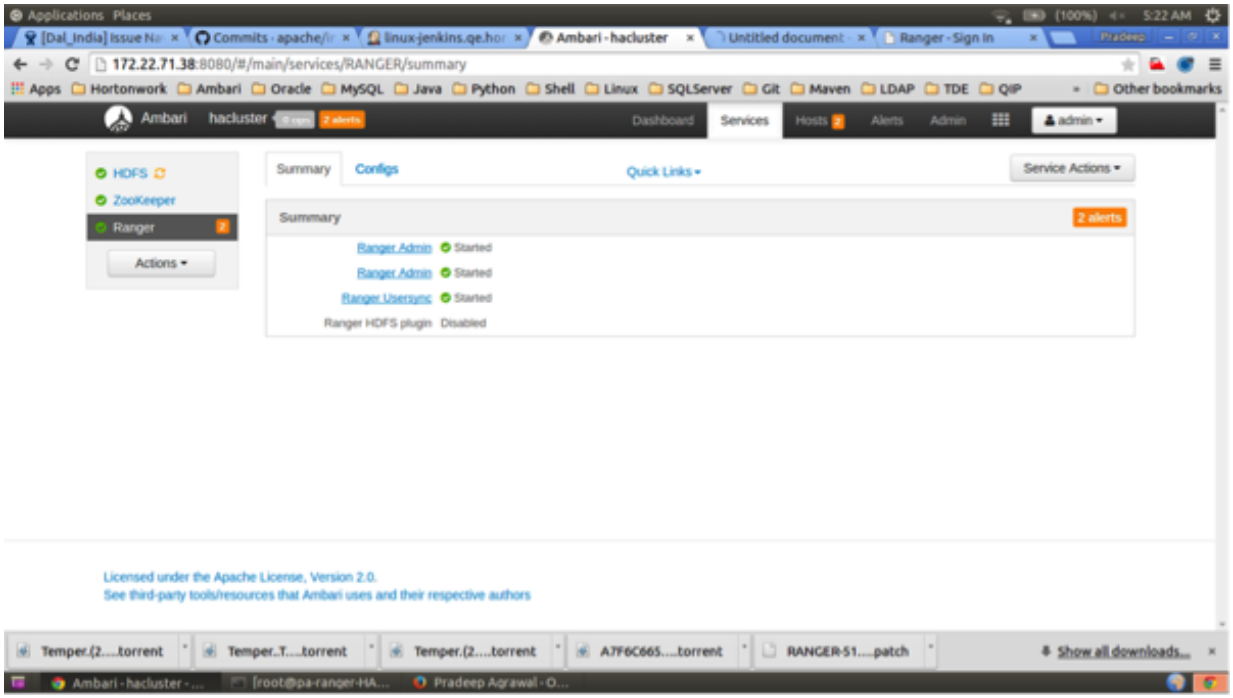
17. Check the settings on the Review page, then click Next.



18. Click Complete on the Install, Start, and Test page to complete the installation.



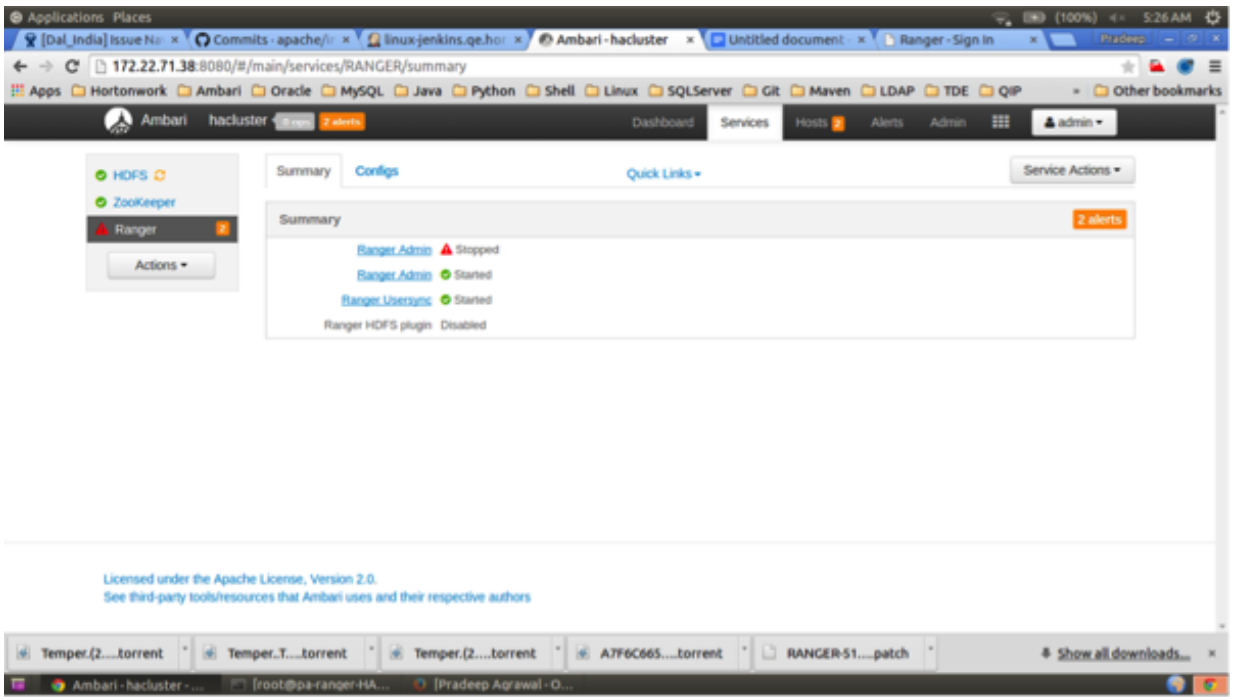
19. When the installation is complete, the Ranger Admin instances are listed on the Ranger Summary page. Select Actions > Restart All Required to restart all services that require a restart.



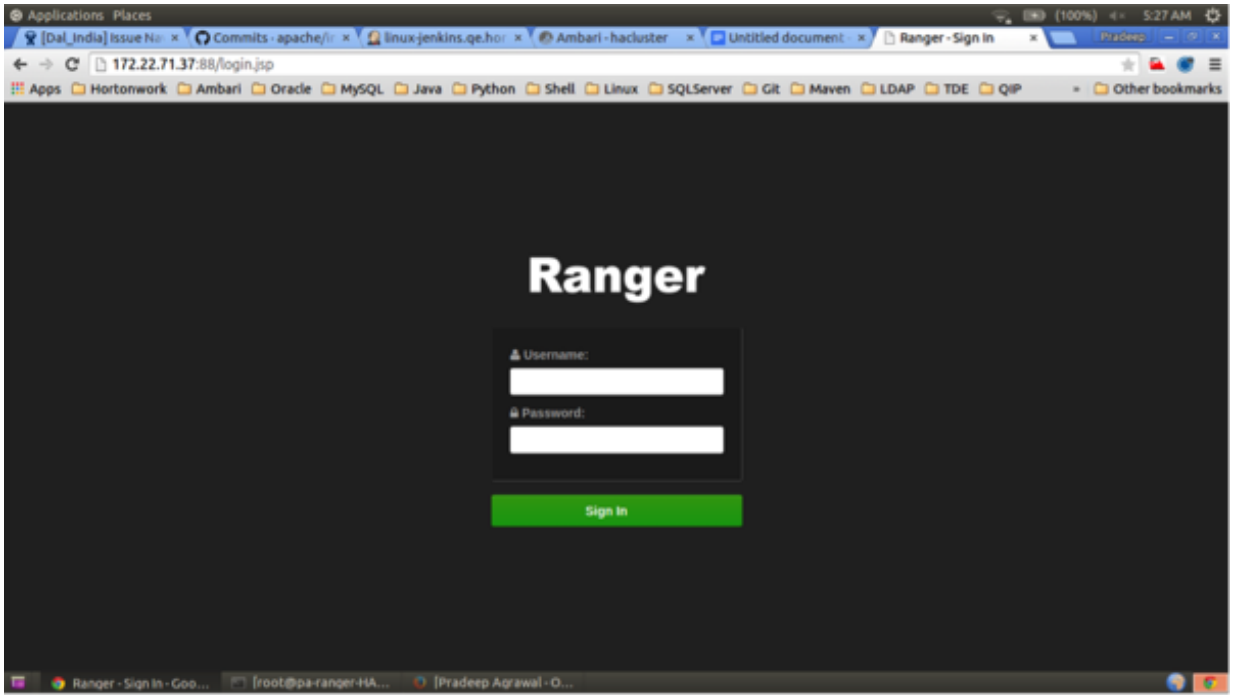
Note:

For Oracle, clear the Setup DB and DB user check box under "Advanced ranger-env" to avoid DB setup.

20. To test the load-balancer and Ranger HA configuration, select Ranger > Service Actions > Stop on one of the Ranger hosts.



21. Use a browser to check the load-balancer host URL (with port). You should see the Ranger Admin page.



22. Use the following steps to generate the self-signed certificate:

a) Switch to the directory that will contain the self-signed certificate:

```
cd /tmp
```

b) Generate the private key:

```
openssl genrsa -out server.key 2048
```

c) Generate the CSR:

```
openssl req -new -key server.key -out server.csr
```

d) Generate the self-signed key:

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out
 server.crt
```

e) Generate the keystore in PEM format:

```
openssl pkcs12 -export -passout pass:ranger -in server.crt -inkey
 server.key -out lbkeystore.p12 -name httpd.lb.server.alias
```

f) Use the keytool to convert the PEM format keystore to JKS format:

```
keytool -importkeystore -deststorepass ranger -destkeypass ranger
 -destkeystore httpd_lb_keystore.jks -srckeystore lbkeystore.p12 -
srcstoretype PKCS12 -srcstorepass ranger -alias httpd.lb.server.alias
```

g) Create a truststore of the load-balancer self-signed keystore:

```
keytool -export -keystore httpd_lb_keystore.jks -alias
 httpd.lb.server.alias -file httpd-lb-trust.cer
```

23. Copy the generated key and certificate into the /usr/local/apache2/conf/ directory.

```
cp server.crt /usr/local/apache2/conf/
cp server.key /usr/local/apache2/conf/
```

24. Add the following entry at the end of the /usr/local/apache2/conf/httpd.conf file to read the custom configuration file:

```
Include /usr/local/apache2/conf/ranger-lb-ssl.conf
```

25. Create a custom conf file for the load-balancer SSL configuration:

```
vi /usr/local/apache2/conf/ranger-lb-ssl.conf
```

Make the following updates:

Add the following lines, then change the <VirtualHost *:8443> port to match the default port you set previously in the httpd.conf file.

```
<VirtualHost *:8443>

        SSLEngine On
        SSLProxyEngine On
        SSLCertificateFile /usr/local/apache2/conf/server.crt
        SSLCertificateKeyFile /usr/local/apache2/conf/server.key

        #SSLCACertificateFile /usr/local/apache2/conf/ranger_lb_crt.pem
        #SSLProxyCACertificateFile /usr/local/apache2/conf/
ranger_lb_crt.pem
        SSLVerifyClient optional
        SSLOptions +ExportCertData
        SSLProxyVerify none
        SSLProxyCheckPeerCN off
        SSLProxyCheckPeerName off
        SSLProxyCheckPeerExpire off
        ProxyRequests off
        ProxyPreserveHost off
```

```
        Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
 env=BALANCER_ROUTE_CHANGED

        <Proxy balancer://rangercluster>
               BalancerMember http://172.22.71.39:6080 loadfactor=1
 route=1
               BalancerMember http://172.22.71.38:6080 loadfactor=1
 route=2


                Order Deny,Allow
                Deny from none
                Allow from all

                ProxySet lbmethod=byrequests scolonpathdelim=On
 stickysession=ROUTEID maxattempts=1 failonstatus=500,501,502,503
 nofailover=Off
        </Proxy>

        # balancer-manager
        # This tool is built into the mod_proxy_balancer
        # module and will allow you to do some simple
        # modifications to the balanced group via a gui
        # web interface.
        <Location /balancer-manager>
                SetHandler balancer-manager
                Order deny,allow
                Allow from all
        </Location>


       ProxyPass /balancer-manager !
       ProxyPass / balancer://rangercluster/
       ProxyPassReverse / balancer://rangercluster/

</VirtualHost>
```

Note:

The URLs listed in the BalancerMember entries are the IP addresses of the Ranger Admin hosts. In this example, the Ranger Admin host adresses are:

```
http://172.22.71.38:6080
http://172.22.71.39:6080
```

26. Run the following command to restart the httpd server:

```
/usr/local/apache2/bin/apachectl restart
```

If you use a browser to check the load-balancer host (with port), you should see the Ranger Admin page.

27. Run the following command to enable Usersync to communicate with Ranger via the load-balancer. This command copies the previously generated truststore file from the /tmp directory imports the certificate into the Usersync truststore.

```
keytool -import -file /tmp/httpd-lb-trust.cer -alias httpd.lb.server.alias
 -keystore /etc/ranger/usersync/conf/mytruststore.jks -storepass changeit
```

28. Restart Ranger Usersync.

29. Run the following command to enable the HDFS plug-in to communicate with Ranger via the load-balancer. This command copies the previously generated truststore file from the /tmp directory imports the certificate into the HDFS truststore.

```
keytool -import -file /tmp/httpd-lb-trust.cer -alias httpd.lb.server.alias
 -keystore /etc/hadoop/conf/ranger-plugin-truststore.jks -storepass
 changeit
```

30. Restart HDFS.
31. In the Ranger Admin UI, select Audit > Plugins. You should see an entry for your repo name with HTTP Response Code 200.
32. Use SSH to connect to the KDC server host. Use the kadmin.local command to access the Kerberos CLI, then check the list of principals for each domain where Ranger Admin and the load-balancer are installed.

```
kadmin.local
kadmin.local: list_principals
```

For example, if Ranger Admin is installed on <host1> and <host2>, and the load-balancer is installed on <host3>, the list returned should include the following entries:

```
HTTP/ <host3>@EXAMPLE.COM
HTTP/ <host2>@EXAMPLE.COM
HTTP/ <host1>@EXAMPLE.COM
```

If the HTTP principal for any of these hosts is not listed, use the following command to add the principal:

```
kadmin.local: addprinc -randkey HTTP/<host3>@EXAMPLE.COM
```

Note:

This step will need to be performed each time the Spnego keytab is regenerated.

33. Use the following kadmin.local commands to add the HTTP Principal of each of the Ranger Admin and load-balancer nodes to the Spnego keytab file:

```
kadmin.local: ktadd -norandkey -kt /etc/security/keytabs/
spnego.service.keytab HTTP/ <host3>@EXAMPLE.COM
kadmin.local: ktadd -norandkey -kt /etc/security/keytabs/
spnego.service.keytab HTTP/ <host2>@EXAMPLE.COM
kadmin.local: ktadd -norandkey -kt /etc/security/keytabs/
spnego.service.keytab HTTP/ <host1>@EXAMPLE.COM
```

Use the exit command to exit ktadmin.local.

34. Run the following command to check the Spnego keytab file:

```
klist -kt /etc/security/keytabs/spnego.service.keytab
```

The output should include the principals of all of the nodes on which Ranger Admin and the load-balancer are installed. For example:

```
Keytab name: FILE:/etc/security/keytabs/spnego.service.keytab
KVNO Timestamp      Principal
---- -----------------
 -------------------------------------------------------
   1 07/22/16 06:27:31 HTTP/ <host3>@EXAMPLE.COM
   1 07/22/16 06:27:31 HTTP/ <host3>@EXAMPLE.COM
   1 07/22/16 06:27:31 HTTP/ <host3>@EXAMPLE.COM
   1 07/22/16 06:27:31 HTTP/ <host3>@EXAMPLE.COM
   1 07/22/16 06:27:31 HTTP/ <host3>@EXAMPLE.COM
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
```

```
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
   1 07/22/16 08:37:23 HTTP/ <host2>@EXAMPLE.COM
   1 07/22/16 08:37:35 HTTP/ <host1>@EXAMPLE.COM
   1 07/22/16 08:37:36 HTTP/ <host1>@EXAMPLE.COM
   1 07/22/16 08:37:36 HTTP/ <host1>@EXAMPLE.COM
   1 07/22/16 08:37:36 HTTP/ <host1>@EXAMPLE.COM
   1 07/22/16 08:37:36 HTTP/ <host1>@EXAMPLE.COM
   1 07/22/16 08:37:36 HTTP/ <host1>@EXAMPLE.COM
```

35. Use scp to copy the Spnego keytab file to every node in the cluster on which Ranger Admin and the load-balancer are installed. Verify that the /etc/security/keytabs/spnego.service.keytab file is present on all Ranger Admin and load-balancer hosts.
36. Configure the Ranger HA Keytab file.

    When setting up Ranger in HA where Kerberos is enabled, the Ambari-managed SPNEGO keytab file is altered. Since Ambari expects the file to contain certain data, the file is now overwritten with the Ambari-cached data during some Kerberos operations. This breaks Ranger in HA.

    In HDP 3.0+, you must duplicate the SPNEGO keytab file for Ranger HA, make various changes, and add the custom property "ranger.ha.spnego.kerberos.keytab". This enables Ambari to use the relevant SPNEGO properties differently depending on whether the Ambari server or the Ambari agent is acting.

    a) Login to the load balancer node.
    b) Enter: cp /etc/security/keytabs/spnego.service.keytab /etc/security/keytabs/ranger.ha.keytab.
    c) Run kadmin.local.
    d) Add the SPNEGO principal entry of the node where the first ranger_admin is installed: ktadd -norandkey -kt /etc/security/keytabs/ranger.ha.keytab HTTP/as-amb-21-1.openstacklocal@EXAMPLE.COM.
    e) Add the SPNEGO principal entry of the node where the second ranger_admin is installed: ktadd -norandkey -kt /etc/security/keytabs/ranger.ha.keytab HTTP/as-amb-21-1.openstacklocal@EXAMPLE.COM.
    f) Verify /etc/security/keytabs/ranger.ha.keytab contains an entry of all the required SPNEGO principals: klist -kt /etc/security/keytabs/ranger.ha.keytab.
    g) Copy the Ranger keytab file to other nodes where ranger_admin is installed: scp /etc/security/keytabs/ranger.ha.keytab.
    h) Update permission: chmod 440 /etc/security/keytabs/ranger.ha.keytab.
    i) Update ownership: chown root:hadoop /etc/security/keytabs/ranger.ha.keytab.
    j) Add config from **Ambari** > **Ranger** > **Configs** > **Advanced** > **Custom ranger-admin-site**: ranger.ha.spnego.kerberos.keytab=/etc/security/keytabs/ranger.ha.keytab.
    k) Restart the Ranger service.

Data Protection

You can ensure data protection by preventing accidental deletion of files and backing up HDFS metadata.

Preventing Accidental Deletion of Files

You can prevent accidental deletion of files by enabling the Trash feature for HDFS.

For additional information regarding HDFS trash configuration, see HDFS Architecture.

You might still cause irrecoverable data loss if the -skipTrash and -R options are accidentally used on directories with a large number of files. You can obtain an additional layer of protection by using the -safely option to the fs shell -rm command. The fs shell -rm command checks the hadoop.shell.safely.delete.limit.num.files property from core-site.xml file, even if you specify -skipTrash. By specifying the -safely option, the -rm command requires that you

confirm if the number of files to be deleted is greater than the limit specified by the assigned value. The default limit for value is 100, referring to 100 files.

This confirmation warning is disabled if value is set at 0 or the -safely is not specified to the -rm command.

To enable the hadoop.shell.safely.delete.limit.num.files property, add the following lines to core-site.xml:

```
<property>
<name>hadoop.shell.safely.delete.limit.num.files</name>
<value>100</value>
<description>Used by -safely option of hadoop fs shell -rm command to avoid
accidental deletion of large directories.</description>
</property>
```

In the following example, the hadoop.shell.safely.delete.limit.num.files property with an associated value of 10 has been added to core-site.xml with -skipTrash . In this example, fs shell -r prompts deletion of a directory with only 10 files. It does not prompt if trash is enabled and -skipTrash is not.

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test1
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 20:57 /tmp/test1/
capacity-scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 20:57 /tmp/test1/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 20:57 /tmp/test1/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 20:57 /tmp/test1/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 20:57 /tmp/test1/kms-
acls.xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 20:57 /tmp/test1/kms-
site.xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 20:57 /tmp/test1/mapred-
site.xml
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 20:57 /tmp/test1/ssl-
client.xml
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 20:57 /tmp/test1/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 20:57 /tmp/test1/yarn-
site.xml
[ambari-qa@c6405 current]$ hdfs dfs -rm -R /tmp/test1
16/10/20 20:58:37 INFO fs.TrashPolicyDefault: Moved: 'hdfs://
c6403.ambari.apache.org:8020/tmp/test1' to trash at: hdfs://
c6403.ambari.apache.org:8020/user/ambari-qa/.Trash/Current/tmp/test1
```

The following example deletes files without prompting or moving to the trash:

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test2
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 20:59 /tmp/test2/
capacity-scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 20:59 /tmp/test2/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 20:59 /tmp/test2/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 20:59 /tmp/test2/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 20:59 /tmp/test2/kms-
acls.xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 20:59 /tmp/test2/kms-
site.xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 20:59 /tmp/test2/mapred-
site.xml
```

```
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 20:59 /tmp/test2/ssl-
client.xml
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 20:59 /tmp/test2/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 20:59 /tmp/test2/yarn-
site.xml
[ambari-qa@c6405 current]$ hdfs dfs -rm -R -skipTrash /tmp/test2
Deleted /tmp/test2
```

The following example prompts for you to confirm file deletion if the number of files to be deleted is greater than the value specified to hadoop.shell.safely.delete.limit.num.files:

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test3
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 21:00 /tmp/test3/
capacity-scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 21:00 /tmp/test3/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 21:00 /tmp/test3/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 21:00 /tmp/test3/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 21:00 /tmp/test3/kms-
acls.xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 21:00 /tmp/test3/kms-
site.xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 21:00 /tmp/test3/mapred-
site.xml
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 21:00 /tmp/test3/ssl-
client.xml
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 21:00 /tmp/test3/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 21:00 /tmp/test3/yarn-
site.xml
[ambari-qa@c6405 current]$ hdfs dfs -rm -R -skipTrash -safely /tmp/test3
Proceed deleting 10 files? (Y or N) N
Delete aborted at user request.
```

Attention:

Using the -skipTrash option without the -safely option is not recommended, as files will be deleted immediately and without warning.

Related Information

HDFS Architecture

Backing Up HDFS Metadata

HDFS metadata represents the structure and attributes of HDFS directories and files in a tree. You can back up the metadata without affecting NameNode availability.

Introduction to HDFS Metadata Files and Directories

HDFS metadata represents the structure of HDFS directories and files in a tree. It also includes the various attributes of directories and files, such as ownership, permissions, quotas, and replication factor.

Files and Directories

Persistence of HDFS metadata is implemented using fsimage file and edits files.

Attention:

Do not attempt to modify metadata directories or files. Unexpected modifications can cause HDFS downtime, or even permanent data loss. This information is provided for educational purposes only.

Persistence of HDFS metadata broadly consist of two categories of files:

fsimage — Contains the complete state of the file system at a point in time. Every file system modification is assigned a unique, monotonically increasing transaction ID. An fsimage file represents the file system state after all modifications up to a specific transaction ID.

edits file — Contains a log that lists each file system change (file creation, deletion or modification) that was made after the most recent fsimage.

Checkpointing is the process of merging the content of the most recent fsimage, with all edits applied after that fsimage is merged, to create a new fsimage. Checkpointing is triggered automatically by configuration policies or manually by HDFS administration commands.

NameNodes

Understand the HDFS metadata directory details taken from a NameNode.

The following example shows an HDFS metadata directory taken from a NameNode. This shows the output of running the tree command on the metadata directory, which is configured by setting dfs.namenode.name.dir in hdfs-site.xml.

```
data/dfs/name
### current#
### VERSION#
### edits_0000000000000000001-0000000000000000007
# ### edits_0000000000000000008-0000000000000000015
# ### edits_0000000000000000016-0000000000000000022
# ### edits_0000000000000000023-0000000000000000029
# ### edits_0000000000000000030-0000000000000000030
# ### edits_0000000000000000031-0000000000000000031
# ### edits_inprogress_0000000000000000032
# ### fsimage_0000000000000000030
# ### fsimage_0000000000000000030.md5
# ### fsimage_0000000000000000031
# ### fsimage_0000000000000000031.md5
# ### seen_txid
### in_use.lock
```

In this example, the same directory has been used for both fsimage and edits. Alternative configuration options are available that allow separating fsimage and edits into different directories. Each file within this directory serves a specific purpose in the overall scheme of metadata persistence:

VERSION — Text file that contains the following elements:

- **layoutVersion** — Version of the HDFS metadata format. When you add new features that require a change to the metadata format, you change this number. An HDFS upgrade is required when the current HDFS software uses a layout version that is newer than the current one.

	namespaceID/clusterID/ blockpoolID	Unique identifiers of an HDFS cluster. These identifiers are used to prevent DataNodes from registering accidentally with an incorrect NameNode that is part of a different cluster. These identifiers also are particularly important in a federated deployment. Within a federated deployment, there are multiple NameNodes working independently. Each NameNode serves a unique portion of the namespace (namespaceID) and manages a unique set of blocks (blockpoolID). The clusterID ties the whole cluster together as a single logical unit. This structure is the same across all nodes in the cluster.
	storageType	Always NAME_NODE for the NameNode, and never JOURNAL_NODE.
	cTime	Creation time of file system state. This field is updated during HDFS upgrades.
edits_start transaction ID-end transaction ID		Finalized and unmodifiable edit log segments. Each of these files contains all of the edit log transactions in the range defined by the file name. In an High Availability deployment, the standby can only read up through the finalized log segments. The standby NameNode is not up-to-date with the current edit log in progress. When an HA failover happens, the failover finalizes the current log segment so that it is completely caught up before switching to active.
fsimage_end transaction ID		Contains the complete metadata image up through . Each fsimage file also has a corresponding .md5 file containing a MD5 checksum, which HDFS uses to guard against disk corruption.
seen_txid		Contains the last transaction ID of the last checkpoint (merge of edits into an fsimage) or edit log roll (finalization of current edits_inprogress and creation of a new one). This is not the last transaction ID accepted

	by the NameNode. The file is not updated on every transaction, only on a checkpoint or an edit log roll. The purpose of this file is to try to identify if edits are missing during startup. It is possible to configure the NameNode to use separate directories for fsimage and edits files. If the edits directory accidentally gets deleted, then all transactions since the last checkpoint would go away, and the NameNode starts up using just fsimage at an old state. To guard against this, NameNode startup also checks seen_txid to verify that it can load transactions at least up through that number. It aborts startup if it cannot verify the load transactions.
in_use.lock	Lock file held by the NameNode process, used to prevent multiple NameNode processes from starting up and concurrently modifying the directory.

JournalNodes

Understand the components of the JournalNode metadata directory.

In an HA deployment, edits are logged to a separate set of daemons called JournalNodes. A JournalNode's metadata directory is configured by setting dfs.journalnode.edits.dir. The JournalNode contains a VERSION file, multiple edits__ files and an edits_inprogress_, just like the NameNode. The JournalNode does not have fsimage files or seen_txid. In addition, it contains several other files relevant to the HA implementation. These files help prevent a split-brain scenario, in which multiple NameNodes could think they are active and all try to write edits.

committed-txid	Tracks last transaction ID committed by a NameNode.
last-promised-epoch	Contains the "epoch," which is a monotonically increasing number. When a new NameNode, starts as active, it increments the epoch and presents it in calls to the JournalNode. This scheme is the NameNode's way of claiming that it is active and requests from another NameNode, presenting a lower epoch, must be ignored.
last-writer-epoch	Contains the epoch number associated with the writer who last actually wrote a transaction.
paxos	Specifies the directory that temporary files used in the implementation of the Paxos distributed consensus protocol. This directory often appears as empty.

DataNodes

Although DataNodes do not contain metadata about the directories and files stored in an HDFS cluster, they do contain a small amount of metadata about the DataNode itself and its relationship to a cluster.

This shows the output of running the tree command on the DataNode's directory, configured by setting dfs.datanode.data.dir in hdfs-site.xml.

```
data/dfs/data/
### current
# ### BP-1079595417-192.168.2.45-1412613236271
# # ### current
# # # ### VERSION
# # # ### finalized
# # # # ### subdir0# # # # ### subdir1
# # # # ### blk_1073741825
# # # # ### blk_1073741825_1001.meta
```

```
# # # ### lazyPersist
# # # ### rbw
# # ### dncp_block_verification.log.curr
# # ### dncp_block_verification.log.prev
# # ### tmp
# ### VERSION
### in_use.lock
```

The purpose of these files are as follows:

BP-random integer-NameNode-IP address-creation time

Top level directory for datanodes. The naming convention for this directory is significant and constitutes a form of cluster metadata. The name is a block pool ID. “BP” stands for “block pool,” the abstraction that collects a set of blocks belonging to a single namespace. In the case of a federated deployment, there are multiple “BP” sub-directories, one for each block pool. The remaining components form a unique ID: a random integer, followed by the IP address of the NameNode that created the block pool, followed by creation time.

VERSION

Text file containing multiple properties, such as layoutVersion, clusterId and cTime, which is much like the NameNode and JournalNode. There is a VERSION file tracked for the entire DataNode as well as a separate VERSION file in each block pool sub-directory.

In addition to the properties already discussed earlier, the DataNode’s VERSION files also contain:

storageType

storageType field is set to DATA_NODE.

blockpoolID

Repeats the block pool ID information encoded into the sub-directory name.

finalized/rbw

Both finalized and rbw contain a directory structure for block storage. This holds numerous block files, which contain HDFS file data and the corresponding .meta files, which contain checksum information. rbw stands for “replica being written”. This area contains blocks that are still being written to by an HDFS client. The finalized sub-directory contains blocks that are not being written to by a client and have been completed.

lazyPersist

HDFS is incorporating a new feature to support writing transient data to memory, followed by lazy persistence to disk in the background. If this feature is in use, then a lazyPersist sub-directory is present and used for lazy persistence of in-memory blocks to disk. We’ll cover this exciting new feature in greater detail in a future blog post.

scanner.cursor

File to which the "cursor state" is saved.

	The DataNode runs a block scanner which periodically does checksum verification of each block file on disk. This scanner maintains a "cursor," representing the last block to be scanned in each block pool slice on the volume, and called the "cursor state."
in_use.lock	Lock file held by the DataNode process, used to prevent multiple DataNode processes from starting up and concurrently modifying the directory.

HDFS Commands

You can use HDFS commands to manipulate metadata files and directories.

hdfs namenode	Automatically saves a new checkpoint at NameNode startup. As stated earlier, checkpointing is the process of merging any outstanding edit logs with the latest fsimage, saving the full state to a new fsimage file, and rolling edits. Rolling edits means finalizing the current edits_inprogress and starting a new one.
hdfs dfsadmin -safemode enter **hdfs dfsadmin -saveNamespace**	Saves a new checkpoint (similar to restarting NameNode) while the NameNode process remains running. The NameNode must be in safe mode, and all attempted write activity fails while this command runs.
hdfs dfsadmin -rollEdits	Manually rolls edits. Safe mode is not required. This can be useful if a standby NameNode is lagging behind the active NameNode and you want it to get caught up more quickly. The standby NameNode can only read finalized edit log segments, not the current in progress edits file.
hdfs dfsadmin -fetchImage	Downloads the latest fsimage from the NameNode. This can be helpful for a remote backup type of scenario.

Configuration Properties

Use the NameNode and data node properties to configure the NameNode and data nodes.

dfs.namenode.name.dir	Specifies where on the local filesystem the DFS name node stores the name table (fsimage). If this is a comma-delimited list of directories then the name table is replicated in all of the directories, for redundancy.
dfs.namenode.edits.dir	Specifies where on the local filesystem the DFS name node stores the transaction (edits) file. If this is a comma-delimited list of directories, the transaction file is replicated in all of the directories, for redundancy. The default value is set to the same value as dfs.namenode.name.dir.
dfs.namenode.checkpoint.period	Specifies the number of seconds between two periodic checkpoints.

dfs.namenode.checkpoint.txns	The standby creates a checkpoint of the namespace every dfs.namenode.checkpoint.txns transactions, regardless of whether dfs.namenode.checkpoint.period has expired.
dfs.namenode.checkpoint.check.period	Specifies how frequently to query for the number of un-checkpointed transactions.
dfs.namenode.num.checkpoints.retained	Specifies the number of image checkpoint files to be retained in storage directories. All edit logs necessary to recover an up-to-date namespace from the oldest retained checkpoint are also retained.
dfs.namenode.num.extra.edits.retained	Specifies the number of extra transactions which are retained beyond what is minimally necessary for a NN restart. This can be useful for audit purposes or for an HA setup where a remote Standby Node might have been offline and need to have a longer backlog of retained edits to start again.
dfs.namenode.edit.log.autoroll.multiplier.threshold	Specifies when an active namenode rolls its own edit log. The actual threshold (in number of edits) is determined by multiplying this value by dfs.namenode.checkpoint.txns. This prevents extremely large edit files from accumulating on the active namenode, which can cause timeouts during namenode start-up and pose an administrative hassle. This behavior is intended as a fail-safe for when the standby fails to roll the edit log by the normal checkpoint threshold.
dfs.namenode.edit.log.autoroll.check.interval.ms	Specifies the time in milliseconds that an active namenode checks if it needs to roll its edit log.
dfs.datanode.data.dir	Determines where on the local filesystem an DFS data node should store its blocks. If this is a comma-delimited list of directories, then data is stored in all named directories, typically on different devices. Directories that do not exist are ignored. Heterogeneous storage allows specifying that each directory resides on a different type of storage: DISK, SSD, ARCHIVE or RAM_DISK.

Back Up HDFS Metadata

You can back up HDFS metadata without taking down either HDFS or the NameNodes.

Prepare to Back Up the HDFS Metadata

Regardless of the solution, a full, up-to-date continuous backup of the namespace is not possible. Some of the most recent data is always lost. HDFS is not an Online Transaction Processing (OLTP) system. Most data can be easily recreated if you re-run Extract, Transform, Load (ETL) or processing jobs.

- Normal NameNode failures are handled by the Standby NameNode. Doing so creates a safety-net for the very unlikely case where both master NameNodes fail.
- In the case of both NameNode failures, you can start the NameNode service with the most recent image of the namespace.
- Name Nodes maintain the namespace as follows:

- Standby NameNodes keep a namespace image in memory based on edits available in a storage ensemble in Journal Nodes.
- Standby NameNodes make a namespace checkpoint and saves an fsimage_* to disk.
- Standby NameNodes transfer the fsimage to the primary NameNodes using HTTP.

Both NameNodes write fsimages to disk in the following sequence:

- NameNodes write the namespace to a file fsimage.ckpt_* on disk.
- NameNodes creates an fsimage_*.md5 file.
- NameNodes moves the file fsimage.ckpt_* to fsimage_.*.

The process by which both NameNodes write fsimages to disk ensures that:

- The most recent namespace image on disk in an fsimage_* file is on the standby NameNode.
- Any fsimage_* file on disk is finalized and does not receive updates.

Perform a Backup of the HDFS Metadata

You can back up HDFS metadata without affecting the availability of NameNode.

Procedure

1. Make sure the Standby NameNode checkpoints the namespace to fsimage_ once per hour.
2. Deploy monitoring on both NameNodes to confirm that checkpoints are triggering regularly.

   This helps reduce the amount of missing transactions in the event that you need to restore from a backup containing only fsimage files without subsequent edit logs. It is good practice to monitor this because edit logs that are large in size and without checkpoints can cause long delays after a NameNode restart while it replays those transactions.
3. Back up the most recent “fsimage_*” and “fsimage_*.md5” from the standby NameNode periodically.

   Try to keep the latest version of the file on another machine in the cluster.
4. Back up the VERSION file from the standby NameNode.

Using HDFS snapshots for data protection

HDFS snapshots enable you to capture point-in-time copies of the file system and protect your important data against user or application errors. You can take snapshots of the entire file system or specified subtrees on the file system.

Using snapshots to protect data is efficient because of the following reasons:

- Snapshot creation is instantaneous regardless of the size and depth of the directory subtree.
- Snapshots capture the block list and file size for a specified subtree. Snapshots do not create extra copies of blocks on the file system.

Related Information

Protecting your enterprise data with HDFS snapshots

Considerations for working with HDFS snapshots

You can create snapshots only for directories that allow the creation of snapshots. If a directory already contains snapshots, you cannot delete or rename the directory unless you remove all the snapshots.

You must consider the following when working with HDFS snapshots:

- You must enable snapshot creation on a particular directory before creating snapshots on that directory. However, you cannot create snapshots on a directory if its corresponding child or parent directory is already enabled for snapshot creation.
- There is no limit on the number of directories on which you can enable snapshot creation. However, you can create a maximum of 65,536 snapshots for a directory.

- You cannot delete or rename a directory that contains snapshots. You must first remove all the snapshots before attempting the delete or rename operation.
- For a directory that has snapshot creation enabled, its path component .snapshot can be used to access the snapshots.

  For example, consider the directory /foo that is enabled for snapshot creation. For the directory /foo with a snapshot snap1, the path /foo/.snapshot/snap1 refers to the snapshot of /foo.
- You can enable or disable snapshot creation on a particular directory only if you have the superuser privilege.
- On a directory that has snapshot creation enabled; you can create, delete, or rename snapshots. These operations require either the superuser privilege or the owner access to the directory. In addition, you can list directories that have snapshot creation enabled or view differences between contents of snapshots.

Enable snapshot creation on a directory

You must enable snapshot creation on a directory before creating snapshots on that directory. If the snapshot creation is enabled, the directory becomes *snapshottable*.

About this task

- You can perform this task only if you have the superuser privilege.
- You cannot enable snapshot creation on any directory if its parent or child directory is already enabled for snapshot creation.

Procedure

Run the hdfs dfsadmin command with the -allowSnapshot option and specify the directory on which you want to enable snapshot creation.
The following example shows how you can enable snapshot creation for the directory /data/dir1:

```
hdfs dfsadmin -allowSnapshot /data/dir1
```

If snapshot creation is successfully enabled on the specified directory, a confirmation message appears.

```
Allowing snapshot on /data/dir1 succeeded
```

Related Information

HDFS Snapshots - Administrator Operations
HDFS Snapshots - User Operations

Create snapshots on a directory

You can create snapshots on a specified directory and protect your important data.

Before you begin

You must have enabled snapshot creation.

About this task

Only a user with either of the following privileges can perform this task:

- The owner privilege to the directory on which to create the snapshots
- The superuser privilege

Procedure

Run the hdfs dfs command with the -createSnapshot option and specify the path to the directory on which you want to create snapshots.

The following example shows how you can create a snapshot snap1 on the directory /data/dir1:

```
hdfs dfs -createSnapshot /data/dir1 snap1
```

If snapshot creation is successfully enabled on the specified directory, a confirmation message appears.

```
Created snapshot /data/dir1/.snapshot/snap2
```

Note: You can also run the command without mentioning the snapshot name. In such a situation, the new snapshot has the time stamp of creation as its name. See the following example:

```
Created snapshot /data/dir1/.snapshot/s20180412-065533.159
```

Related Information

HDFS Snapshots - User Operations

Recover data from a snapshot

If data is erroneously removed from a directory for which snapshots are available, you can recover the lost data using snapshots. The snapshot ensures that the file blocks corresponding to the deleted files or directories are not removed from the file system. Only the metadata is modified to reflect the deletion.

About this task

You must have read access to the files or directories that you want to restore.

Procedure

Run the hdfs dfs command with the cp option to copy the deleted data from the snapshot to the destination directory. The following example shows how you can recover a file imp_details.xls from a snapshot of the directory (/data/dir1) that contained the file:

```
hdfs dfs -cp /data/dir1/.snapshot/s20180412-065533.159/imp_details.xls /
data/dir1/
```

Options to determine differences between contents of snapshots

Run the hdfs snapshotDiff command for a report that lists the difference between the contents of two snapshots. Run the distcp diff command to determine the difference between contents of specified source and target snapshots, and use the command with the -update option to move the difference to a specified target directory.

Generating a report listing the difference between contents of two snapshots

Using the hdfs snapshotDiff between two snapshots on a specified directory path provides the list of changes to the directory. Consider the following example:

```
hdfs snapshotDiff /data/dir1 snap1 snap2
M .
- ./file1.csv
R ./file2.txt -> ./fileold.txt
+ ./filenew.txt
```

This example shows the following changes to the directory /data/dir1 after the creation of snap1 and before the creation of snap2:

Statement	Explanation
M .	The directory /data/dir1 is modified.
- ./file1.csv	The file file1.csv is deleted.

Statement	Explanation
R ./file2.txt -> ./fileold.txt	The file file2.txt is renamed to fileold.txt.
+ ./filenew.txt	The file filenew.txt is added to the directory /data/dir1.

Moving the differences between the contents of two snapshots to a specified directory

Using the distcp diff command with the -update option on snapshots enables you to determine the difference between the contents of two snapshots and move the difference to a specified target directory. Consider the following example:

hadoop distcp -diff snap_old snap_new -update /data/source_dir /data/target_dir

The command in this example determines the changes between the snapshots snap_old and snap_new present in the source_dir directory, and updates the target_dirdirectory with the changes.

The following conditions must be satisfied for the content changes to be moved to /data/target_dir:

- Both /data/source_dir and /data/target_dir are distributed file system paths.
- The snapshots snap_old and snap_new are created for /data/source_dir such that snap_old is older than snap_new.
- The /data/target_dir path also contains snap_old. In addition, no changes are made to /data/target_dir after the creation of snap_old.

Related Information

HDFS Snapshots - User Operations

Managing Hadoop DR with 'distcp' and 'snapshots'

Snapshot operations

As an administrator, you can enable or disable snapshot creation on a directory. These operations require the superuser privilege. As a user; you can create, delete, or rename snapshots on a directory that has snapshot creation enabled. These operations require either the superuser privilege or the owner privilege on the directory.

Administrator Operations

The following table lists the snapshot-related administrator operations that you can on specified directories:

Operation	Command
Enable snapshot creation on a directory	hdfs dfsadmin -allowSnapshot <path>
Disable snapshot creation on a directory	hdfs dfsadmin -disallowSnapshot <path>

For more information about these commands, see https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html#Administrator_Operations.

User Operations

The following table lists the user operations that you can perform on snapshots:

Operation	Command
Create snapshots	hdfs dfs -createSnapshot <path> [<snapshotName>]
Delete snapshots	hdfs dfs -deleteSnapshot <path> <snapshotName>
Rename snapshots	hdfs dfs -renameSnapshot <path> <oldName> <newName>
List directories on which snapshot creation is enabled (*snapshottable* directories)	hdfs lsSnapshottableDir

Operation	Command
List differences between contents of snapshots	hdfs snapshotDiff <path> <fromSnapshot> <toSnapshot>

For more information about these commands, see https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html#User_Operations.