Hortonworks Data Platform

Hadoop High Availability

(March 2, 2016)

docs.cloudera.com

Hortonworks Data Platform: Hadoop High Availability

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Table of Contents

List of Figures

List of Tables

1. High Availability for Hive Metastore

This document is intended for system administrators who need to configure the Hive Metastore service for High Availability.

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.

1.1. Use Cases and Failover Scenarios

This section provides information on the use cases and failover scenarios for high availability (HA) in the Hive metastore.

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 boxes concurrently. Each Hive metastore client reads the configuration property `hive.metastore.uris` to get a list of metastore servers with which it can 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

A Hive metastore client always uses the first URI to connect with the metastore server. If the metastore server becomes unreachable, the client randomly picks up a URI from the list and attempts to connect with that.

1.2. Software Configuration

Complete the following tasks to configure Hive HA solution:

1. Install HDP [2]
2. Update the Hive Metastore [2]
3. Validate configuration [3]

1.2.1. Install HDP

Use the following instructions to install HDP on your cluster hardware. Ensure that you specify the virtual machine (configured in the previous section) as your NameNode.

1. Download the Apache Ambari repository using the instructions provided in " Download the Ambari Repository" in the *Automated Install with Ambari* guide.

   Note

   Do not start the Ambari server until you have configured the relevant templates as outlined in the following steps.

2. 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 HDP installation.
     - Continue the Ambari installation process using the instructions provided in " Installing, Configuring, and Deploying an HDP Cluster" in the *Automated Install with Ambari* guide.
     - Complete the Ambari installation. Ensure that the installation was successful.

1.2.2. Update the Hive Metastore

HDP components configured for HA must use a NameService rather than a NameNode. Use the following instructions to update the Hive Metastore to reference the NameService rather than a Name Node.

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.

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/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>
```

1.2.3. Validate configuration

Test various failover scenarios to validate your configuration.

2. Highly Available Reads with 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.

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 Region Server.
Region server	A Region server serves data requests for one or more regions. A single region is serviced by only one Region Server, but a Region Server may serve multiple regions. When region replication is enabled, a Region Server 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 Region Server. Region Servers write files to HDFS after the MemStore reaches a configurable maximum value specified with the

HBase Concept	Description
	`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 Region Servers 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.

For information about configuring regions, see "HBase Cluster Capacity and Region Sizing" in the System Administration Guide.

2.1. Introduction to HBase High Availability

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

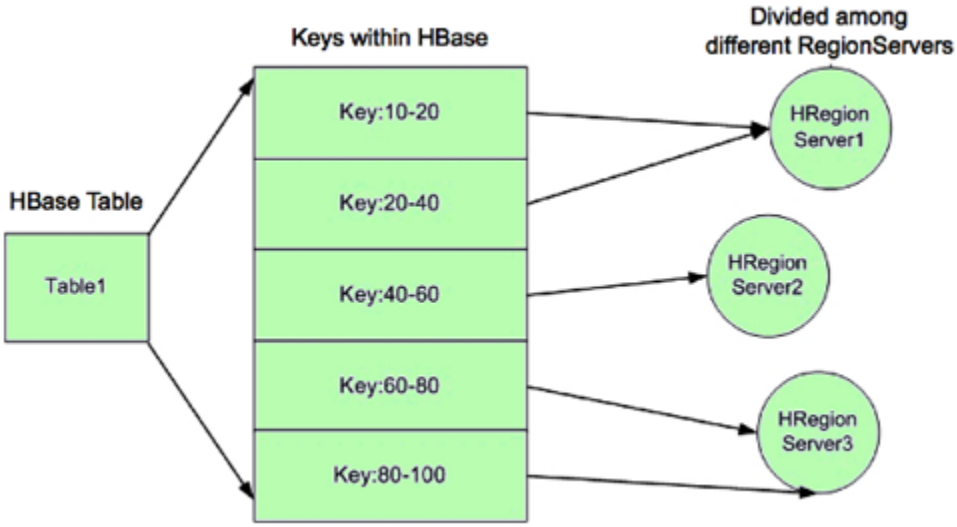
However, because of this "single homing" of reads to a single location, if the server becomes unavailable, the regions of the table that are hosted in the Region Server become unavailable for some time until they are recovered. There are three phases in the region recovery process: detection, assignment,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

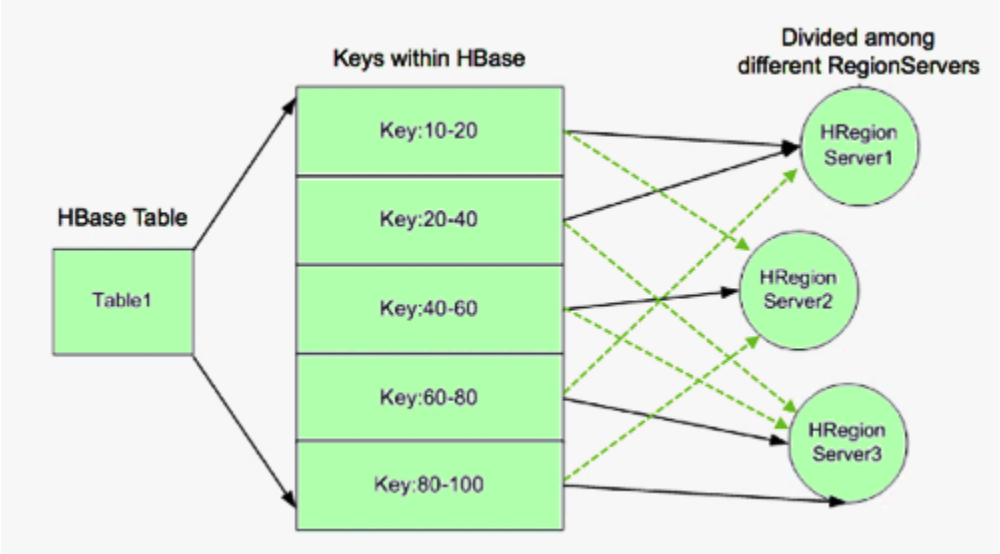
For achieving high availability for reads, HBase provides a feature called “region replication”. In this model, for each region of a table, there can be multiple replicas that are opened in different Region Servers. By default, the region replication is set to 1, so only a single region replica is deployed 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, Region Server 1 is responsible for responding to queries and scans for keys 10 through 40. If Region Server 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 Region Server 1 is not available, by hosting replicas of the region and assigning the region replicas to other Region Servers as backups. In the following image, Region Server 2 hosts secondary region replicas for keys 10-20, and Region Server 3 hosts the secondary region replica for keys 20-40. Region Server 2 also hosts the secondary region replica for keys 80-100. There are no separate Region Server processes for secondary replicas. Rather, Region Servers can serve regions in primary or secondary mode. When Region Server 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 Region Server 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 Region Servers 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 Region Server 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 Region Server failure, because only the primary Region Server 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 Region Server.

2.2. Propagating Writes to Region Replicas

As discussed in the introduction, 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 more information, see Creating Highly-Available HBase Tables.

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 Region Server, 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. For more information about these properties, see Configuring HA Reads for HBase.

Async WAL Replication

The second mechanism for propagating writes to secondaries is done via the Async WAL Replication feature. This feature is only available in HA Phase 2 (starting with HDP 2.2).

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. For more information about these properties, see Creating Highly-Available HBase Tables.

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`.

2.3. Timeline Consistency

With timeline consistency, HBase introduces a Consistency definition that can be provided per read operation (get or scan):

```
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 Region Server 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 Region Server 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.

2.4. Configuring HA Reads for HBase

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

The following table describes server-side properties. Set these properties for all servers in your HBase cluster that use region replicas.

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.
`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.`

Property	Example value	Description
		`StochasticLoadBalancer`, which is the only load balancer that supports reading data from Region Servers 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 Region Server.
`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.

The following table lists client-side properties. Set these properties for all clients, applications, and servers in your HBase cluster that use region replicas.

Property	Example value	Description
`hbase.ipc.client.specificThreadForWriting`	true	Specifies whether to enable interruption of RPC threads at

Property	Example value	Description
		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.

2.5. Creating Highly-Available HBase Tables

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

Creating HA Tables with the HBase Java API

HBase application developers 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.

Creating HA Tables with the HBase Shell

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

2.6. Querying Secondary Regions

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

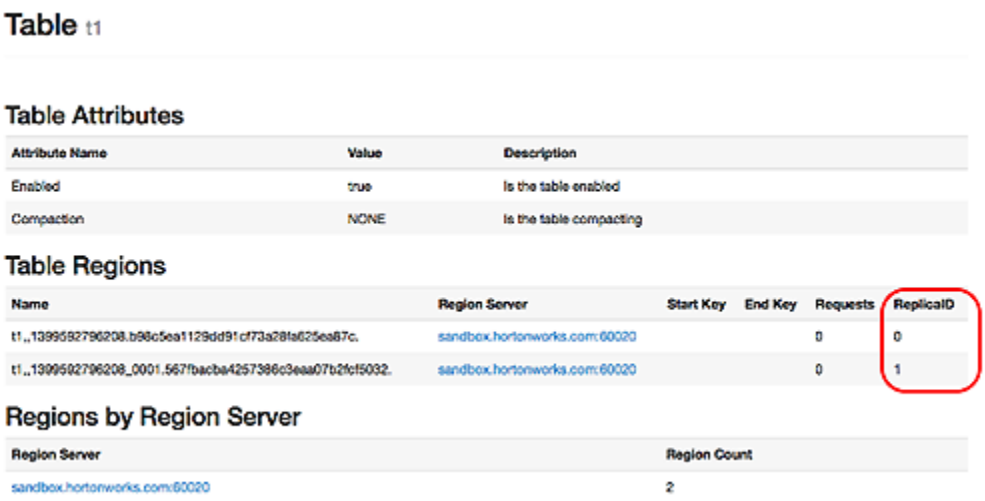
This release of HBase does not provide a mechanism to determine if the results from an interactive query arrived from the primary or a secondary replica.

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'}
```

2.7. Monitoring 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.

2.8. 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. Some use cases for cluster replication include:

- 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.

2.8.1. HBase Cluster Replication Overview

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 Region Server 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 (HBASE-7709), all clusters that have already consumed the data are also tracked. This prevents replication loops.

The WALs for each Region Server must be kept in HDFS as long as they are needed to replicate data to a slave cluster. Each Region Server 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.

Note

Terms such as *master-master*, *master-slave*, and *cyclical* were used to describe replication relationships in HBase. These terms added confusion, and have been abandoned in favor of discussions about cluster topologies appropriate for different scenarios.

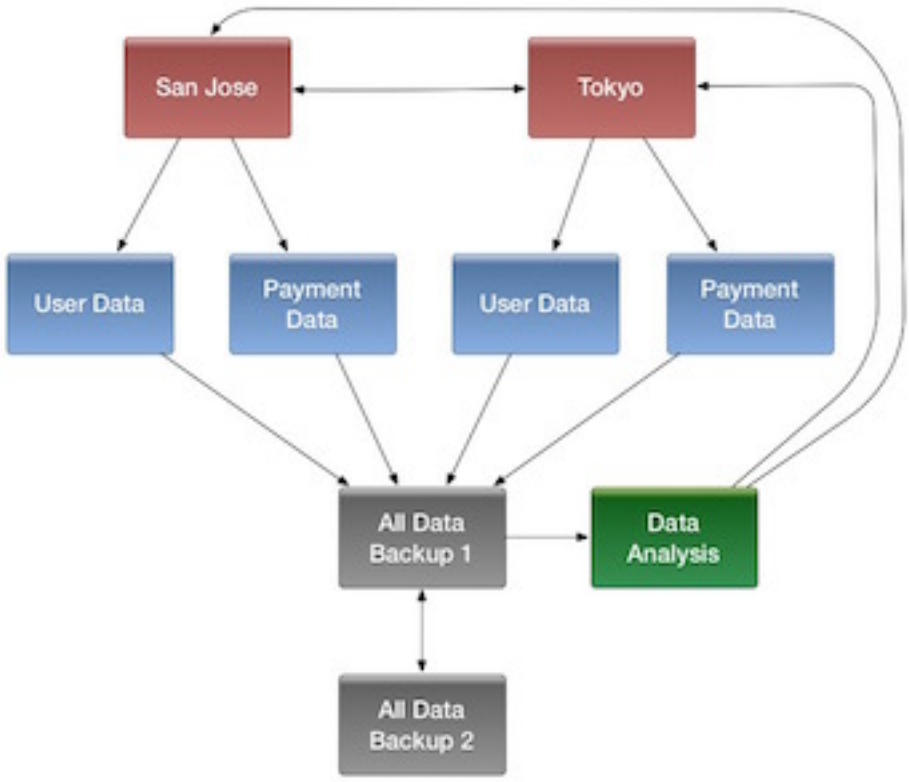
2.8.1.1. HBase Cluster Topologies

- 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. The arrows indicate the data paths.

Figure 2.1. 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.

2.8.2. Managing and Configuring HBase Cluster Replication

Process Overview

1. Configure and start the source and destination clusters. Create tables with the same names and column families on both the source and destination clusters, so that the destination cluster knows where to store the data that it receives.
2. All hosts in the source and destination clusters should be reachable to each other.

3. 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. Check to be sure that replication has not been disabled. The `hbase.replication` setting defaults to `true`.
5. On the source cluster, in HBase shell, add the destination cluster as a peer, using the `add_peer` command.
6. On the source cluster, in HBase shell, enable the table replication, using the `enable_table_replication` command.
7. Check the logs to see if replication is taking place. If so, you see messages like the following, coming from the Replication Source:

```
LOG.info("Replicating "+ClusterId + " -> " + peerClusterId);
```

Table 2.1. HBase Cluster Management Commands

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.

2.8.3. Verifying Replicated HBase Data

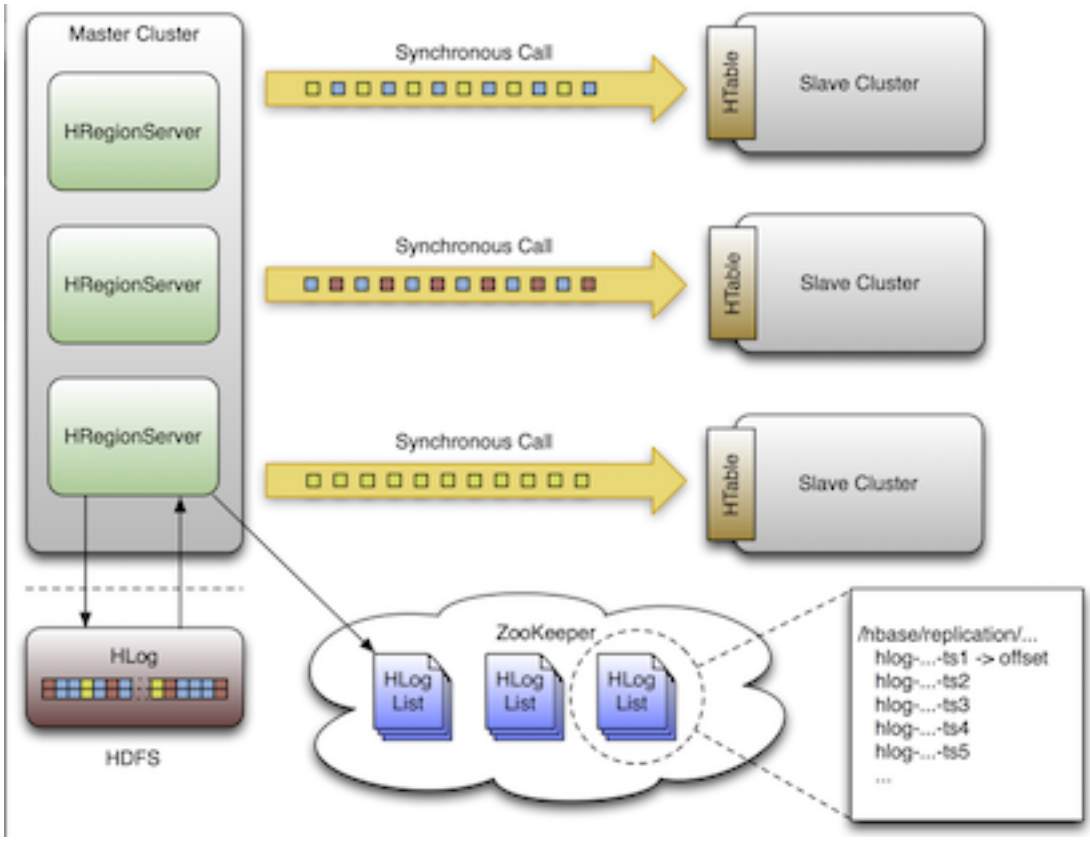
The `VerifyReplication` MapReduce job, which is included in HBase, performs a systematic comparison of replicated data between two different clusters. Run the VerifyReplication job on the master cluster, supplying it with the peer ID and table name to use for validation. You can limit the verification 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.

2.8.4. HBase Cluster Replication Details

Figure 2.2. HBase Replication Architecture Overview



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 Region Server 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 Region Server on the slave cluster.
6. The Region Server 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 Region Server 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 Region Server reads, filters, and edits the log edits as described in Step 4. The slave Region Server does not answer the RPC call.
10. The master Region Server sleeps and tries again. The number of attempts can be configured.
11. If the slave Region Server is still not available, the master selects a new subset of Region Servers 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 Region Server (by moving them from the Region Server 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 Region Server then replicates the backlog of logs that accumulated during the outage.

2.8.4.1. Spreading Queue Failover Load

When replication is active, a subset of Region Servers in the source cluster is responsible for shipping edits to the sink. This responsibility must be failed over like all other Region Server 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.

2.8.4.2. Preserving 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 Region Servers which are involved in the replication. This option was introduced in HBASE-10322.

2.8.4.3. HBase Replication Internals

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 Region Server is shipping the logs to. The `rs` znode has one child znode for each Region Server in the cluster. The child znode name is the Region Server hostname, client port, and start code. This list includes both live and dead Region Servers.

2.8.4.4. Choosing Region Servers to Replicate to

When a master cluster Region Server 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' (Region Servers 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 Region Server. Because this selection is performed by each master Region Server, the probability that all slave Region Servers 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 Region Servers 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 Region Servers. 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 Region Servers respond by selecting a new pool of slave Region Servers to which to replicate.

2.8.4.5. Keeping Track of Logs

Each master cluster Region Server 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 Region Server, 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. See Region Server Failover for an example.

When a source is instantiated, it contains the current WAL that the Region Server 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 Region Server 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 Region Server 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 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.

2.8.4.6. Reading, Filtering, and Sending Edits

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 KeyValues that are scoped GL0BAL 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 Region Server 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.

2.8.4.7. 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.

2.8.4.8. Region Server Failover

When no Region Servers are failing, keeping track of the logs in ZooKeeper adds no value. Unfortunately, Region Servers 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 Region Servers keeps a watcher on every other Region Server, 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 Region Server znode that contains its queues. The Region Server 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 Region Server 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 Region Server. When the reader hits the end of the last log, the queue znode is deleted and the master cluster Region Server closes that replication source.

For example, the following hierarchy represents what the znodes layout might be for a master cluster with 3 Region Servers that are replicating to a single slave with the ID of 2. The Region Server 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 Region Server 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)
```

2.8.5. HBase Replication Metrics

The following metrics are exposed at the global Region Server 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.

2.8.6. Replication Configuration 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`
`replication.sleep.before.failover`	Number of milliseconds a worker should sleep before attempting to replicate the WAL queues for a dead Region Server.	–
`replication.executor.workers`	Number of Region Servers a Region Server should attempt to failover simultaneously.	`1`

2.8.7. 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.

3. 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.

Formerly, if a cluster had a single NameNode, and that machine or process became unavailable, the entire cluster would be unavailable until the NameNode was either restarted or started 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 the new NameNode 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 User's Guide.

3.1. Architecture

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

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

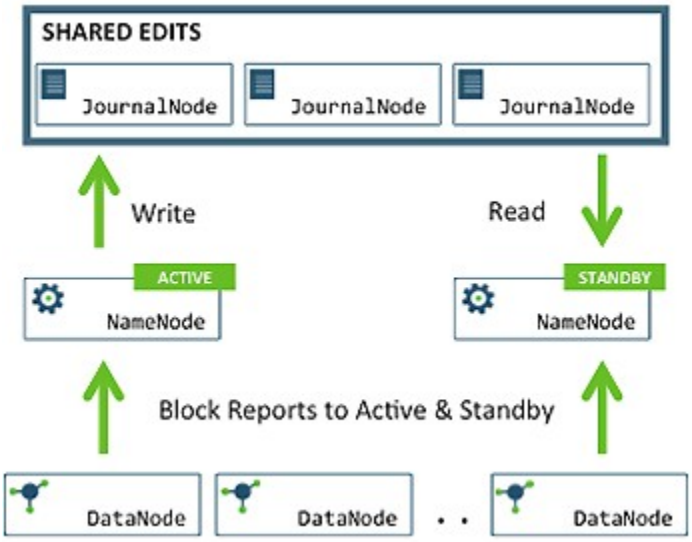
In order for the Standby node to keep its state synchronized with the Active node, both nodes communicate with a group of separate daemons called **JournalNodes** (JNs). When the Active node performs any namespace modification, the Active node durably logs a modification record to a majority of these 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 acts 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 both of the NameNodes, and send block location information and heartbeats to both 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 two NameNode machines thus causing potential data loss. (This situation is called 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 to 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.

3.2. Hardware Resources

Ensure that you prepare the following hardware resources:

- **NameNode machines**: The machines where you run Active and Standby NameNodes, should have exactly the same hardware. For recommended hardware for NameNodes, see "Hardware for Master Nodes" in the Cluster Planning Guide.
- **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, (i.e. 3, 5, 7, etc.).

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.

3.3. Deploy 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.

3.3.1. Configure 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 the properties mentioned below to those files.

After you have added the configurations below, ensure that the same set of HDFS configuration files are propogated 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:

- **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` and `nn2`, the value of this property is:

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

Note

Currently, a maximum of two NameNodes can be configured per nameservice.

- **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 above two NameNode IDs - `nn1` and `nn2`.

  Note that there will be two 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>
```

- **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 both NameNodes HTTP servers to listen on. For example:

```
<property>
```

```
 <name>dfs.namenode.http-address.mycluster.nn1</name>
 <value>machine1.example.com:50070</value>
</property>

<property>
 <name>dfs.namenode.http-address.mycluster.nn2</name>
 <value>machine2.example.com:50070</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

See "Creating Service Principals and Keytab files for HDP" in the "Setting Up Security for Manual Installs" chapter of the Installing HDP Manually for instructions on configuring Kerberos-based security for Highly Available clusters.

3.3.2. Deploy NameNode HA Cluster

In this section, 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.

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.

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/
hadoop-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.

   - At the host with the journal node (if it is separated from the primary host), execute the following command:

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

   - 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/hadoop-
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"
   ```

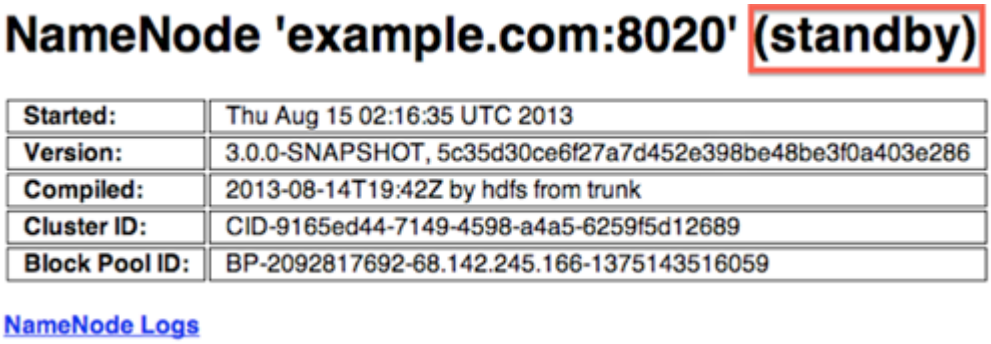
   Ensure that NN2 is running correctly.

7. 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"
   ```

8. 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.

   
   NameNode 'example.com:8020' (standby)

Started:	Thu Aug 15 02:16:35 UTC 2013
Version:	3.0.0-SNAPSHOT, 5c35d30ce6f27a7d452e398be48be3f0a403e286
Compiled:	2013-08-14T19:42Z by hdfs from trunk
Cluster ID:	CID-9165ed44-7149-4598-a4a5-6259f5d12689
Block Pool ID:	BP-2092817692-68.142.245.166-1375143516059

   NameNode Logs

   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
   ```

9. Transition one of the HA NameNode to Active state.

   Initially, both NN1 and NN2 are in Standby state. Therefore you must transition one of the NameNode to Active state. This transition can be performed using one of the following options:

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

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

  For more information on the haadmin command, see "Appendix: Administrative Commands."

- **Option II - Deploying Automatic Failover** You can configure and deploy automatic failover using the instructions provided in Configure and Deploy NameNode Automatic Failover.

3.3.3. Deploy Hue with an HA Cluster

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

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

   For RHEL/CentOS/Oracle Linux:

   ```
   yum install hadoop-httpfs
   ```

   For SLES:

   ```
   yum 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 for use by setting up the symlink in `/etc/init.d`:

   ```
   > ln -s /usr/hdp/{HDP2.3.x version number}/etc/rc.d/init.d/
   hadoop-httpfs /etc/init.d/hadoop-httpfs
   ```

   For example, `{HDP2.3.x version number}` could be '2.3.4.7-$BUILD'.

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
```

3.3.4. Deploy Oozie with an HA Cluster

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

- 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. Find more information about ZooKeeper here.

- 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. For information about how to set up your Oozie servers to handle failover, see Configuring Oozie Failover.

3.4. Operating a NameNode 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 `dfs admin` 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 `dfs admin` 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 `dfs admin` command options are sent to the DataNodes:

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

3.5. Configure and Deploy NameNode Automatic Failover

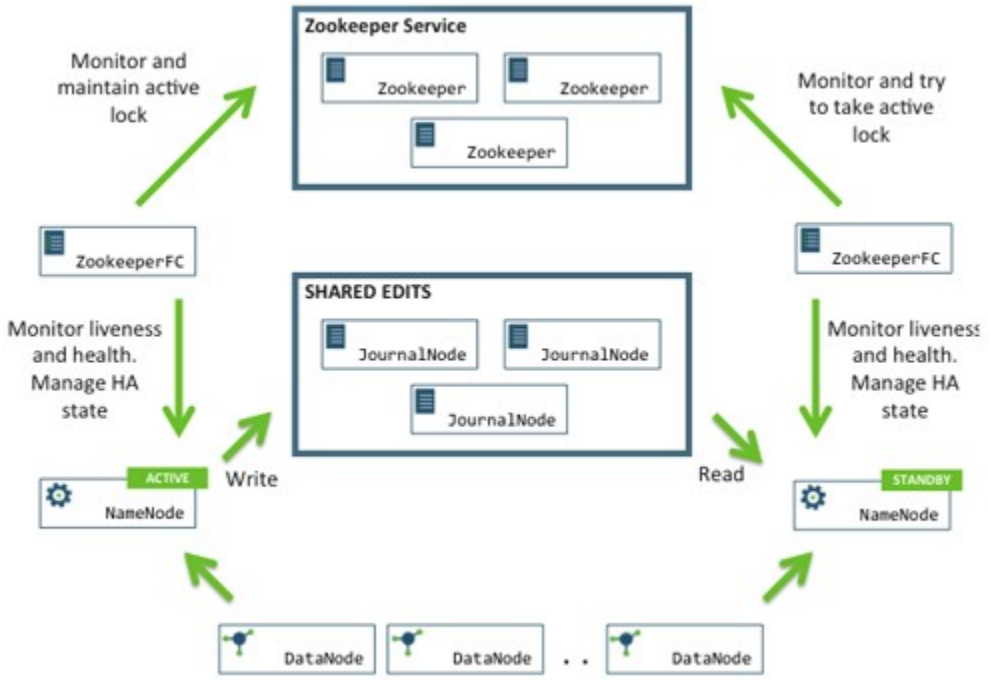
The preceding sections describe how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby NameNode, even if the active node has failed. This section describes how to configure and deploy automatic failover.

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.



3.5.1. Prerequisites

Complete the following prerequisites:

- Make sure that you have a working ZooKeeper service. If you had an Ambari deployed HDP cluser with ZooKeeper, you can use that. If not, deploy ZooKeeper using the instructions provided in the Non-Ambari Cluster Installation Guide.

  ### 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) using the instructions provided in " Controlling HDP Services Manually," in the *HDP Reference Guide*.

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

3.5.2. Instructions

Complete the following instructions:

1. Configure automatic failover.

- Set up your cluster for automatic failover. Add the following property to the the hdfs-site.xml file for both 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 the core-site.xml file for both 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 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"
```

4. Start the JournalNodes, NameNodes, and DataNodes using the instructions provided in "Controlling HDP Services Manually," in the *HDP Reference 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.

3.5.3. Configuring Oozie Failover

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.

3.6. Appendix: 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: DFSHAAdmin [-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.

4. Resource Manager High Availability

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 was a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single ResourceManager, and if that machine or process became unavailable, the entire cluster would be unavailable until the ResourceManager was either restarted or started 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 ResourceManager.
- Planned maintenance events such as software or hardware upgrades on the ResourceManager machine would 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.

4.1. Hardware Resources

Ensure that you prepare the following hardware resources:

- **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.

4.2. Deploy ResourceManager HA Cluster

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

As described in the following sections, first configure manual or automatic ResourceManager failover. Then deploy the ResourceManager HA cluster.

4.2.1. Configure Manual or Automatic ResourceManager Failover

Prerequisites

Complete the following prerequisites:

- Make sure that you have a working ZooKeeper service. If you had an Ambari deployed HDP cluster with ZooKeeper, you can use that ZooKeeper service. If not, deploy ZooKeeper using the instructions provided in the Non-Ambari Cluster Installation Guide.

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 using the instructions provided in "Controlling HDP Services Manually," in the HDP Reference Guide.

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.ConfiguredRMFailoverProxyProvider`	When HA is enabled, the class to be used by Clients, AMs and NMs to failover to the Active RM. It should extend

Property Name	Recommended Value	Description
		`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`

Property Name	Recommended Value	Description
`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.

4.2.2. Deploy the ResourceManager HA Cluster

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 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"
```

4. Start HDFS using the instructions provided in "Controlling HDP Services Manually," in the HDP Reference Guide.

5. Start YARN using the instructions provided in "Controlling HDP Services Manually," in the HDP Reference Guide.

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 using the instructions provided in "Controlling HDP Services Manually," in the HDP Reference Guide.

4.2.3. Minimum Settings for Automatic ResourceManager HA Configuration

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>
```

4.2.4. Testing ResourceManager HA on a Single Node

If you would like 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>
```

5. HiveServer2 High Availability via ZooKeeper

This chapter describes how to implement HiveServer2 High Availability through ZooKeeper, including:

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

5.1. 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.

- Rolling Upgrade

  It is possible to register HS2 instances based on their version, configuring HS2s from the new version as active and HS2s from the old version as passive. ZooKeeper passes connections only to active HS2s, so over time the old version participates less and less in existing sessions. Ultimately, when the old version is no longer participating at all, it can be removed.

  For further information, see Rolling Upgrade Guide.

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.

5.2. Dynamic Service Discovery 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.

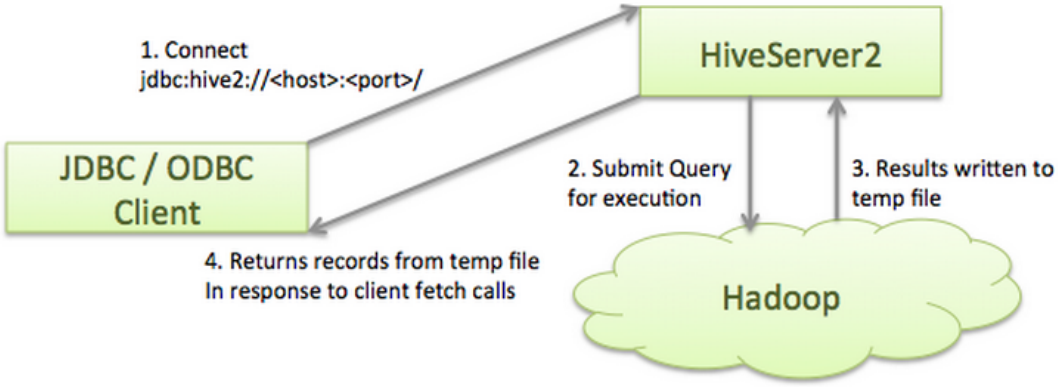
   This establishes a session where multiple queries can be executed.

   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

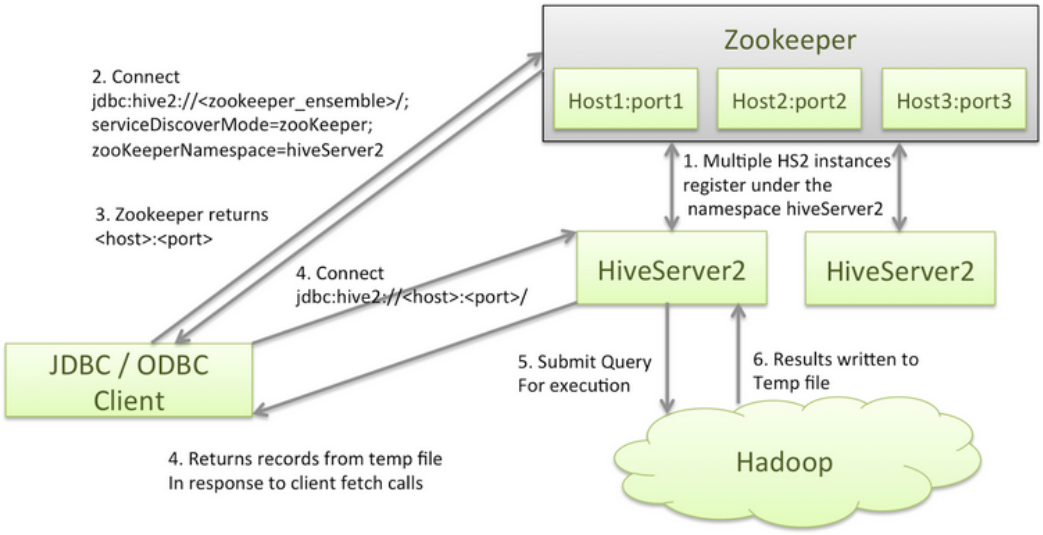
Query execution with ZooKeeper takes advantage of dynamic discovery. Thus, the client driver needs to know how to use this capability, which is available in HDP 2.2 and later with the JDBC driver and ODBC driver 2.0.0.

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).



5.3. Rolling Upgrade for HiveServer2 Through ZooKeeper

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

- Configuration Requirements

  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 adminstrator 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.

- Upgrade Steps

  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.

- Downgrade (Rollback) Steps

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.