Cloudera Runtime 7.1.1

Introduction to HBase MCC

Date published: 2022-08-10
Date modified: 2022-08-10

https://docs.cloudera.com/

Legal Notice

Contents

Introduction to HBase Multi-cluster Client.....4
HBase MCC Usage with Kerberos.....5
HBase MCC Usage in Spark with Scala.....7
HBase MCC Usage in Spark with Java.....13
Zookeeper Configurations.....14
HBase MCC Configurations.....14
HBase MCC Restrictions.....15

Introduction to HBase Multi-cluster Client

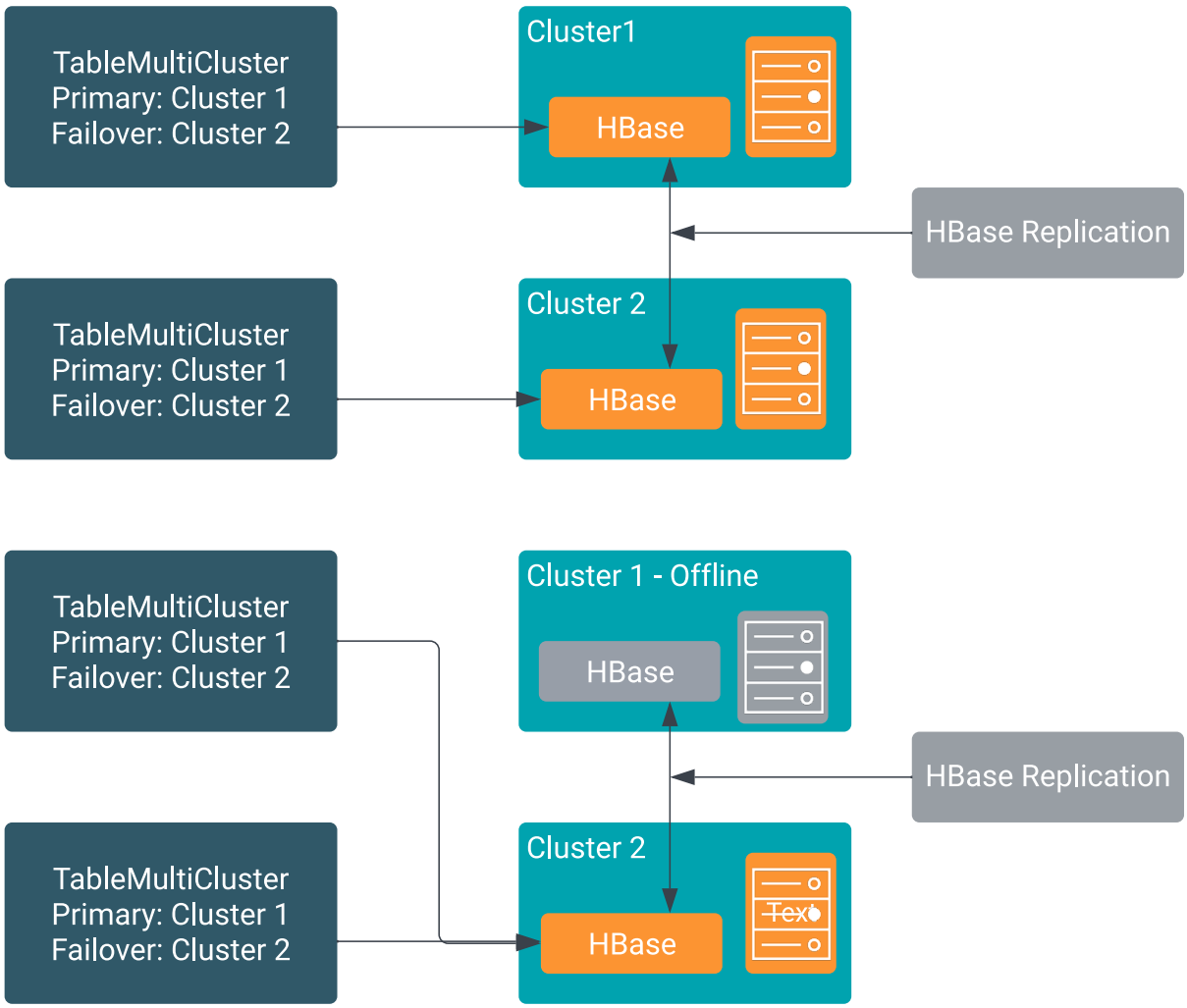
Learn about the HBase Multi-cluster Client (MCC) and ways to switch between the single HBase clusters and Multi-HBase clients. Explore HBase MCC usage with Kerberos with examples. Implement your HBase clients in Spark using Scala or Java.

Cloudera Operational Database supports HBase MCC that ensures high availability and better performance. HBase MCC relies on the existing HBase replication functionality to provide an eventually consistent solution in cases of primary cluster downtime.

HBase MCC offers the following benefits:

- Switch between single HBase clusters and Multi-HBase Client with limited code changes.
- Support multiple number of linked HBase clusters.
- Continue to keep applications running without disrupting a cluster during an upgrade or outage.

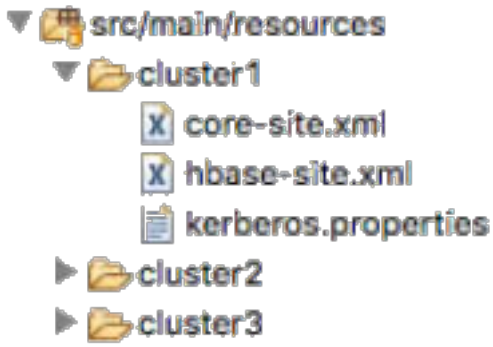
Figure 1: HBase MCC architecture



HBase MCC Usage with Kerberos

You can configure HBase MCC with Kerberos authentication for enhanced security and better performance.

When you build a client application and want to add the configurations as resources, you must configure the client resources as follows:



- Each cluster folder inside the client application (src/main/resources) must be named with a lowercase "cluster" followed by a sequence starting with 1 and through the “n” number of clusters.
- The first cluster directory is configured as primary, each additional as a failover.
- There must be at least one folder available in the classpath.

Each folder must contain the following three files:

1. hbase-site.xml
2. core-site.xml
3. kerberos.properties

The kerberos.properties file must be configured as a key value property using the following two configurations:

1. principal
2. keytab

For example,

```
principal=youruser/yourhost@EXAMPLE.COM
keytab=/path/to/youruser.keytab
```

The following Java example of a multi-cluster client MultiClusterConf multiClusterConf = new MultiClusterConf (true) shows that the default constructor is passed as true that creates an instance of an HBase configuration for each cluster assuming default configurations and no override.

```
package org.your.domain.testpackage;

import org.apache.commons.lang.StringUtils;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

public class AutoConfigClassPath {
 static final Log log = LogFactory.getLog(AutoConfigClassPath.class);
 public static void main(String[] args) throws Exception{
  MultiClusterConf multiClusterConf = new MultiClusterConf(true);  //autoCo
nfigure = true
  Connection connection = ConnectionFactoryMultiClusterWrapper.createConn
ectionMultiUgi(multiClusterConf);
```

```
  String tableName = "test_table";
  String familyName = "cf1";
  int numberOfPuts = 10;
  int milliSecondsOfWait = 200;
  Table table = connection.getTable(TableName.valueOf(tableName));
  for (int i = 1; i <= numberOfPuts; i++) {
   log.info("PUT");
   Put put = new Put(Bytes.toBytes(i % 10 + ".key." + StringUtils.leftPad(
String.valueOf(i), 12)));
   put.addColumn(Bytes.toBytes(familyName), Bytes.toBytes("C"), Bytes.toBy
tes("Value:" + i));
   table.put(put);
   Thread.sleep(milliSecondsOfWait);
  }
  log.info("Closing Connection");
  connection.close();
  log.info(" - Connection Closed");
  System.exit(0);
 }
}

java -cp "hbase-mcc-0.2.0-SNAPSHOT.jar:/path/to/clusters/:/opt/cloudera/parc
els/CDH/jars/*" org.your.domain.testpackage.AutoConfigClassPath
```

Note: You can also build and execute with src/main/resources configured with cluster directories.

You can use autoConfigure to override any parameter in the configuration file, and if the parameter impacts MCC, you must configure with "cluster1".

You can use the multiClusterConf method to change the parameter in the Java program directly. The following example shows setting the parameter and referencing the index of "cluster1" to impact MCC.

```
MultiClusterConf multiClusterConf = new MultiClusterConf(true, false);  //au
toConfigure = true, addResourceOnStart = false
multiClusterConf.set("hbase.mcc.failover.mode", "false", 0);
```

However, it is recommended not to use the previous method; instead update the Java configuration files. Refer to the following code snippet.

```
package com.cloudera.hbase.mcc.examples;

import com.cloudera.hbase.mcc.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.util.Bytes;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class BuilderPatternExample {
    private static final Logger LOG = LoggerFactory.getLogger(BuilderPattern
Example.class);
    public static void main(String[] args) throws Exception {
        MultiClusterConf multiClusterConf = new MultiClusterConf.MultiClu
sterConfBuilder()
                .addClusterConfig(new ClusterConfig.ClusterConfigBuilder()
                        .coreSite("/Users/admin/Documents/hbase/cluster1/co
re-site.xml")
                        .hbaseSite("/Users/admin/Documents/hbase/cluster1/
hbase-site.xml")
                        .principal("cluster1tls/server1.example.site@TLS1.C
OM")
```

```
                        .keytab("/Users/admin/Documents/hbase/cluster1/clus
ter1tls_host.keytab")
                        .build()
                        .set("hbase.client.retries.number", "1")
                        .set("hbase.client.pause", "1"))
                        .set("zookeeper.recovery.retry", "0"))
                .addClusterConfig(new ClusterConfig.ClusterConfigBuilder()
                        .coreSite("/Users/admin/Documents/hbase/cluster2/
core-site.xml")
                        .hbaseSite("/Users/admin/Documents/hbase/cluster2/hb
ase-site.xml")
                        .principal("cluster2tls/server2.example.site@TLS2
.COM")
                        .keytab("/Users/admin/Documents/hbase/cluster2/cl
uster2tls_host.keytab")
                        .build()
                        .set("hbase.client.retries.number", "1")
                        .set("hbase.client.pause", "1"))
                .build();

        Connection connection = ConnectionFactoryMultiClusterWrapper.creat
eConnectionMultiUgi(multiClusterConf);
        String tableName = "test_table";
        String familyName = "cf1";
        int numberOfPuts = 3000;
        int milliSecondsOfWait = 200;
        Table table = connection.getTable(TableName.valueOf(tableName));
        for (int i = 1; i <= numberOfPuts; i++) {
            LOG.info("PUT");
            Put put = new Put(Bytes.toBytes("key" + i));
            put.addColumn(Bytes.toBytes(familyName), Bytes.toBytes("C"),
Bytes.toBytes("Value:" + i));
            table.put(put);
            Thread.sleep(milliSecondsOfWait);
        }
        LOG.info("Closing Connection");
        connection.close();
        LOG.info(" - Connection Closed");
        System.exit(0);
    }

}
```

HBase MCC Usage in Spark with Scala

The Spark implementation does not support multiple keytabs. You must configure and enable the cross-realm kerberos for Spark to operate properly.

You do not need to distribute the keytabs or configuration files on the cluster, instead make them available on the edge node from which you start the Spark application. The principal and keytab are referenced twice. First, the credentials must be passed to start the application on the cluster and second, they must be made available to the code responsible for creating credentials for each cluster.

Spark driver writes the credentials to HDFS and later executors obtain and apply them before any code is executed. A process for both the driver and executor is started to ensure the credentials are refreshed in a timely manner.

The following example uses a constructor method for the HBase MCC configuration. It assumes that you are passing hbase-site.xml and core-site.xml into your executable main method. While using this utility, you must modify the configurations using the configuration files only. This utility does not support Spark streaming.

To start the Spark application when the primary cluster's HBase is down, use the following parameter:

```
--conf "spark.security.credentials.hbase.enabled=false"
```

You must override the HBase `UserProvider` class with the `UserProviderMultiCluster` class. If you are using the constructor method, this is set automatically.

```
val HBASE_CLIENT_USER_PROVIDER_CLASS = "hbase.client.userprovider.class"
mccConf.set(HBASE_CLIENT_USER_PROVIDER_CLASS, ConfigConst.HBASE_MCC_USER_PR
OVIDER)
```

Example on writing to HBase with Spark DataFrame API

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.SparkToHbase \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.COM"
 \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/con
figs/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml
core-site1.xml hbase-site2.xml core-site2.xml
```

Example code: you can view the example code on this GitHub repository.

Breakdown of the code execution:

- The first part of the code sets up HBase to override the connection implementation in hbase-spark along with setting up variables containing configuration information for the two clusters. A recent addition is `HBASE_CLIENT_USER_PROVIDER_CLASS` used in MCC to allow Spark to start jobs where an HBase cluster may be down for maintenance.

  ```
  val HBASE_CLIENT_CONNECTION_IMPL = "hbase.client.connection.impl"
      val HBASE_CLIENT_USER_PROVIDER_CLASS = "hbase.client.userprovider.clas
  s"
      val CONNECTION_IMPL = "com.cloudera.hbase.mcc.ConnectionMultiCluster"
      val primaryHBaseSite = args(0)
      val primaryCoreSite = args(1)
      val failoverHBaseSite = args(2)
      val failoverCoreSite = args(3)
  ```

- HBase MCC provides a base configuration that stores the configurations for the MCC applications. The first step is to create an instance of the configuration and set the override from the variables above. For spark, it is also required to set the `hbase.mcc.userName` and `hbase.mcc.applicationId`. These are used to write the delegation tokens into HDFS; in addition, it is required to set `fs.defaultFS` from the current Spark execution.

  ```
  val mccConf = new MultiClusterConf
      mccConf.set(HBASE_CLIENT_CONNECTION_IMPL, CONNECTION_IMPL)
      //Sets the default FS to that of the cluster submitting the spark job
      mccConf.set(CommonConfigurationKeysPublic.FS_DEFAULT_NAME_KEY, sc.ha
  doopConfiguration.get(CommonConfigurationKeysPublic.FS_DEFAULT_NAME_KEY))
      mccConf.set("hbase.mmc.userName", sc.sparkUser)
      mccConf.set("hbase.mcc.applicationId", sc.applicationId)
  ```

- The HBase configurations are stored independently and prefixed when loaded into the application to maintain isolation between each cluster configuration and the base MCC configuration. The following code segment shows HBase execution related parameters, which you plan to override.

```
val primary = HBaseConfiguration.create()
    primary.addResource(new Path(primaryHBaseSite))
    primary.addResource(new Path(primaryCoreSite))
    primary.set("hbase.client.retries.number", "1"); //Override Default
 Parameters
    primary.set("hbase.client.pause", "1"); //Override Default Parameters
    primary.set("zookeeper.recovery.retry", "0"); //Override Default Param
eters

    val failover = HBaseConfiguration.create()
    failover.addResource(new Path(failoverHBaseSite))
    failover.addResource(new Path(failoverCoreSite))
    failover.set("hbase.client.retries.number", "1"); //Override Default
 Parameters
    failover.set("hbase.client.pause", "1"); //Override Default Parameters
```

- Once the configurations for each HBase cluster have been created, it is time to call the code that manages the delegation tokens for the clusters. This process returns the token name to the configuration and ensures that the executor is reading the proper tokens and applying the credentials.

```
val credentialsManager = CredentialsManager.getInstance
    primary.set(ConfigConst.HBASE_MCC_TOKEN_FILE_NAME, credentialsManage
r.confTokenForCluster(primaryHBaseSite, primaryCoreSite, sc))
    failover.set(ConfigConst.HBASE_MCC_TOKEN_FILE_NAME, credentialsManager
.confTokenForCluster(failoverHBaseSite, failoverCoreSite, sc))
```

- Each of the final cluster configurations must be added to the MCC configuration and the following function prefixes all of the parameters for each cluster. As the hbase-spark implementation only takes a single configuration, everything is merged into an Uber configuration.

```
mccConf.addClusterConfig(primary)
    mccConf.addClusterConfig(failover)
```

- The `HBaseContext` from hbase-spark is created and the dataframe API is used as in any other case for Spark.

```
import spark.implicits._

    new HBaseContext(sc, mccConf.getConfiguration())

    val rdd = sc.parallelize(Seq(("rowkey","SparkToHbase","0","","1")))
    val df_withcol = rdd.toDF("rowKey", "application", "batchId", "timeS
tamp", "loaded_events")
    logger.info("Data frame to Hbase : " + df_withcol.show())
    val hbaseTableName = "test_table"
    val hbaseTableSchema ="""rowKey STRING :key, application STRING cf1:
APP, batchId STRING cf1:BID, timeStamp STRING cf1:TS, loaded_events STRING
 cf1:PR"""
    logger.info("Started writing to Hbase")
    df_withcol.write.format("org.apache.hadoop.hbase.spark")
      .options(Map("hbase.columns.mapping" -> hbaseTableSchema, "hbase.tab
le" -> hbaseTableName))
      .save()
    logger.info("Completed writing to hbase")
```

Example on writing to HBase with Spark Streaming

The following example shows an application to an HDFS directory for new files. When a file is added, it counts each of the words, keeping record of the count for any previously read file and then updates HBase with the new count.

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.SparkStreamingExample
 \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.COM"
 \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/config
s/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml
 core-site1.xml hbase-site2.xml core-site2.xml
```

Example code: you can view the example code on this GitHub repository.

In this example the configuration must be executed inside the `putHBase` function. That is why the variables for the token file names must be set outside of the function. If you try to use the Spark Streaming Context inside that function, it throws an error as MCC is not serializable. This does not happen in structured streaming which is shown in the following examples.

Example on reading from HDFS and writing to HBase with Spark structured streaming

In this example an application is created that will listen to an HDFS directory for new files. When a file is added, it will read the file into a micro-batch using structured streaming and push the results into HBase.

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.StructuredStreamingHD
FSToHBase \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-memory 2g \
--executor-memory 2g \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.C
OM" \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/cluster1/hbase-site1.xml,/tmp/cluster1/core-site1.xml,/tmp/clu
ster2/hbase-site2.xml,/tmp/cluster2/core-site2.xml \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn \
--deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml core-site1.xml
 hbase-site2.xml core-site2.xml \
/user/exampleuser/StructuredStreamingExampleSource/ /user/exampleuser/Spa
rkStreamingExampleCheckpoint/
```

Example code: you can view the example code on this GitHub repository.

Example on reading from Kafka and writing to HBase with Spark structured streaming

The following example shows how to create an application that pulls messages from Kafka. When you push a set of messages to Kafka, this process creates a micro-batch using the structured streaming and push the results into HBase.

For this use case, a properties file is set up to pass into the application for the required Kafka configurations. Example properties file:

```
spark.kafkatohbase.target.hbase.table test_table
spark.kafkatohbase.target.hbase.cf cf1
spark.kafkatohbase.source.kafka.topic spark_2
spark.kafkatohbase.checkpoint.dir /user/exampleuser/SparkStreamingExampleC
heckpoint/
spark.kafkatohbase.kafka.ssl.truststore.location ./cm-auto-global_truststt
ore.jks
spark.kafkatohbase.kafka.bootstrap.servers yourzookeeper1:9093,yourzookeeper
2:9093,yourzookeeper3:9093
spark.kafkatohbase.kafka.security.protocol SASL_SSL
spark.kafkatohbase.kafka.ssl.truststore.password trustorepasswordhere
spark.kafkatohbase.startingOffsets earliest
spark.kafkatohbase.keytab kafka.keytab
spark.kafkatohbase.principal kafka/example.server.com@EXAMPLE.COM
```

A jaas.conf file also need to be set.

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    debug =true
    useKeyTab=true
    storeKey=true
    keyTab="/tmp/kafka.keytab"
    useTicketCache=false
    serviceName="kafka"
    principal="kafka/example.server.com@EXAMPLE.COM";
};
```

Example command:

```
hdfs dfs -rm -r /user/cluster1tls/SparkStreamingExampleCheckpoint/
hdfs dfs -mkdir /user/cluster1tls/SparkStreamingExampleCheckpoint/
spark-submit --class com.cloudera.hbase.mcc.scala.test.StructuredStreaming
ExampleKafkaToHBase \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-memory 2g \
--executor-memory 2g \
--conf spark.streaming.receiver.maxRate=20 \
--conf spark.dynamicAllocation.minExecutors=5 \
--conf spark.dynamicAllocation.initialExecutors=5 \
--conf spark.dynamicAllocation.maxExecutors=10 \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.server.com@EXAMPLE.COM" \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--conf "spark.security.credentials.hive.enabled=false" \
--conf "spark.security.credentials.hdfs.enabled=false" \
--driver-java-options "-Djava.security.auth.login.config=./producer_jaas2
.conf -Djava.io.tmpdir=/tmp"  \
--conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./
producer_jaas2.conf -Djava.io.tmpdir=/tmp" \
--files /tmp/cluster1/hbase-site1.xml,/tmp/cluster1/core-site1.xml,/tmp/clu
ster2/hbase-site2.xml,/tmp/cluster2/core-site2.xml,/tmp/producer_jaas.conf#p
roducer_jaas.conf,/var/lib/cloudera-scm-agent/agent-cert/cm-auto-global_trus
tstore.jks,/tmp/kafka.keytab \
--principal exampleuser/example.server.com@EXAMPLE.COM" \
--keytab /tmp/cluster1/exampleuser.keytab \
```

```
--master yarn \
--deploy-mode cluster \
--properties-file  /tmp/properties.conf \
/tmp/ScalaHbaseTest.jar hbase-site1.xml core-site1.xml hbase-site2.xml core-
site2.xml
```

Example code: you can view the example code on this GitHub repository.

Example on writing to HBase bulk put and hbase-spark

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.HBaseMCCBulkPutExampl
e \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.C
OM" \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/confi
gs/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml
 core-site1.xml hbase-site2.xml core-site2.xml
```

Note: This example is using the MCC configurations constructor to eliminate the boilerplate configurations.

Example code: you can view the example code on this GitHub repository.

Example on writing to HBase bulk get and hbase-spark

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.HBaseMCCBulkGetExampl
e \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.C
OM" \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/confi
gs/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml
 core-site1.xml hbase-site2.xml core-site2.xml
```

Note: This example is using the MCC configurations constructor to eliminate the boilerplate configurations.

Example code: you can view the example code on this GitHub repository.

Example on writing to HBase bulk delete and hbase-spark

Example command:

```
spark-submit --class com.cloudera.hbase.mcc.scala.test.HBaseMCCBulkDeleteExa
mple \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.COM"
 \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/config
s/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/ScalaHbaseTest.jar hbase-site1.xml
 core-site1.xml hbase-site2.xml core-site2.xml
```

Example code: you can view the example code on this GitHub repository.

HBase MCC Usage in Spark with Java

Example on writing to HBase bulk put and hbase-spark

Example command:

```
spark-submit --class com.cloudeara.hbase.mcc.java.HBaseMCCBulkPutExample \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.COM"
 \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/config
s/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/JavaMCCExamples-0.0.1-SNAPSHOT.jar
 hbase-site1.xml core-site1.xml hbase-site2.xml core-site2.xml
```

Example code: you can view the example code on this GitHub repository.

Example on writing to HBase bulk get and hbase-spark

Example command:

```
spark-submit --class com.cloudeara.hbase.mcc.java.HBaseMCCBulkGetExample \
--jars "/tmp/hbase-mcc-0.2.0-SNAPSHOT.jar" \
--driver-class-path "hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.executor.extraClassPath=hbase-mcc-0.2.0-SNAPSHOT.jar" \
--conf "spark.hbase.connector.security.credentials.enabled=true" \
--conf "spark.yarn.principal=exampleuser/example.kdc.server.com@EXAMPLE.COM"
 \
--conf "spark.yarn.keytab=/tmp/configs/exampleuser.keytab" \
--conf "spark.security.credentials.hbase.enabled=false" \
```

```
--files /tmp/configs/hbase-site1.xml,/tmp/configs/core-site1.xml,/tmp/config
s/hbase-site2.xml,/tmp/configs/core-site2.xml  \
--principal exampleuser/example.kdc.server.com@EXAMPLE.COM \
--keytab /tmp/configs/exampleuser.keytab \
--master yarn --deploy-mode cluster /tmp/JavaMCCExamples-0.0.1-SNAPSHOT.jar
 hbase-site1.xml core-site1.xml hbase-site2.xml core-site2.xml
```

Example code: you can view the example code on this GitHub repository.

Zookeeper Configurations

HBase MCC uses the zookeeper.recovery.retry parameter to monitor the number of retries when Zookeeper is down.

zookeeper.recovery.retry

When a cluster is online and an application is started, the HBase client communicates to the Zookeeper quorum. When you perform a read or write operation on region servers and a failover happens, MCC runs a background thread to check when HBase becomes available again. If an application is started and the cluster is down, MCC uses the zookeeper.recovery.retry parameter to check Zookeeper multiple times. This might not be desired, especially for Spark streaming which operates with a microbatch. Each batch needs to iterate over the default before it finally fails over and hit the second cluster.

You can set this parameter to either 1 or 0. You can set the parameters differently for each cluster; set it as default for the second cluster and 0 for the primary so that it may fail over quickly. In addition, when you set zookeeper.recovery.retry to 0 with HBase, the number of client connections to Zookeeper increases and you might want to monitor `maxclientcnxns`. This is applicable for a failure scenario, when an application is starting and Zookeeper is down.

HBase MCC Configurations

HBase Multi-cluster Client (MCC) provides various configuration parameters to set up clusters, mode, and different connection related properties.

hbase.mcc.failover.clusters

This is a comma separated list of cluster names that represent the clusters available when running MCC. The first cluster in the list is the primary cluster. If this cluster fails to respond, the requests are routed to the next cluster in line. This configuration is automatically generated based on N+ number of clusters in the submission.

The following example shows an uber configuration.

```
hbase.failover.clusters=hbase.mcc.cluster0,hbase.mcc.cluster1
hbase.mcc.cluster0.hbase.zookeeper.quorum=1.0.0.2
hbase.mcc.cluster1.hbase.zookeeper.quorum=1.0.0.3
```

Note: Auto-configuration does not support building uber configuration.

hbase.mcc.failover.mode

The default value is set to true, this means that if the primary cluster fails, the code automatically attempts to execute on failover clusters until all failover clusters are exhausted. If set to false, the only attempt is on the primary or first cluster. If you are executing a put or delete when set to true and culter failover happens, HBase MCC executes the put or delete on the failover cluster.

This configuration is not applicable for the read or scan methods.

Note: This configuration does not apply to reads or scans. This is applied to puts or deletes.

hbase.mcc.connection.pool.size

The default value is set as 20. Part of `java.util.concurrent.Executors` creates a thread pool that reuses a fixed number of threads operating from a shared unbounded queue.

hbase.mcc.allow.check.and.mutate

The default value is false that disables the `checkAndMutate` function on the Table class.

hbase.mcc.wait.time.before.trying.primary.after.failure

The default value is 30000 milliseconds (30 seconds). This is the default amount of time in milliseconds that the application waits before re-attempting the primary cluster after a failure and executing against a failover cluster. The cluster tracker uses this thread to check and ensure that a cluster is available before calls are routed to that cluster.

hbase.mcc.wait.time.before.enable.cluster

The default value is 10000 milliseconds (10 seconds). This represents the wait time to read the metadata for a cluster before the region server is actually available to read. If this number is too low, the system detects that the cluster is available from the metastore, tries to read or write from the cluster because the region server is decoupled from the meta and is not yet available. This additional buffer allows some time before the metadata is available for the region to become fully online.

hbase.mcc.buffered.mutator.behavior

The supported values are failover, replicate, and replay. The default value is failover. When `BufferedMutator` is used, there might be a data loss. When set to failover, HBase MCC writes to a single cluster connection. When a flush fails for that connection, it starts writing new mutations to the new connection. Any mutations that are part of the failure flush are lost. Data loss in `BufferedMutator` is caused when HBase puts are stored in client memory up until the point that a flush takes place. If that flush fails when a region server is down, data is not be persisted.

The second option is to configure `BufferedMutator` with replicate, which replicates all calls for mutations to all connections. When a connection fails, data is not lost because we are writing to another connection automatically.

The third option is to configure replay, this operation stores data in a cache until flush is called. When flush is called, HBase MCC attempts to write all data from the cache into the first available and active connection's `BufferedMutator` followed by an immediate flush. If the flush is a success, the cache is cleared. If the flush fails, HBase MCC pushes the data to the next connection's `BufferedMutator`. This also helps to prevent data loss with `BufferedMutator` while not pushing to each cluster by default as an alternative to the replicate option.

In all cases with `BufferedMutator`, you could still have data loss on the cluster when services go down or if no cluster is available to write the data. You need to ensure that your application is ready to handle the failures as they happen.

HBase MCC Restrictions

HBase MCC contains a few restrictions when you are executing it in Spark compared to executing it in a standard Java application.

Only a single keytab can be used with Spark. In the Java implementation multiple UGIs can be called as long as a proper krb5.conf can connect easily to different clusters in different realms with multiple keytabs.