Week-7 Hadoop
Assignment Report

Siddharth Kumar

October 3, 2025

Contents

Chapter 1

Enable High Availability on HDFS and yarn

```
root@stg-hdpsiddharth102:/home/sre[nb6][stg]# sudo -u yarn yarn rmadmin -getAllServiceState
stg-hdpsiddharth102.phonepe.nb6:8033                active
stg-hdpsiddharth104.phonepe.nb6:8033                standby
```

Figure 1.1: Active and standby nodes

Chapter 2

Understanding the Working and Internals of HDFS HA and YARN HA

2.1 Introduction

High Availability (HA) in the Hadoop ecosystem removes single points of failure for critical cluster services. This chapter explains the architecture, internals, state synchronization, failover mechanisms, and operational details of two core HA features:

- **HDFS High Availability (HDFS HA)** — makes the NameNode highly available.
- **YARN High Availability (YARN HA)** — makes the ResourceManager highly available.

2.2 HDFS High Availability (HA)

2.2.1 Problem statement and motivation

In a classic HDFS deployment the NameNode holds the namespace (file and directory metadata) and the mapping from files to blocks. Because the NameNode is responsible for the entire namespace, a single NameNode is a single point of failure (SPOF). If it crashes, clients cannot read or write metadata and the cluster becomes effectively unusable. HDFS HA eliminates this SPOF by running two (or more) NameNodes in an Active/Standby configuration with mechanisms that keep the Standby synchronized and allow fast automated failover.

2.2.2 Key components

Active NameNode Accepts client requests and performs metadata changes (creates, deletes, renames, block allocations).

Standby NameNode Maintains a synchronized copy of the namespace and stays ready to become Active without losing metadata updates.

JournalNodes (Quorum Journal Manager, QJM) A small ensemble (typically 3 or 5 nodes) that provides a replicated shared edit log. The Active writes edits to the JournalNodes; the Standby tails these edits to stay in sync.

ZooKeeper Used for leader election and for the ZooKeeper Failover Controller (ZKFC) coordination. ZooKeeper stores ephemeral znodes that indicate which NameNode is currently Active.

ZKFC (ZooKeeper Failover Controller) A process running on the NameNode hosts that participates in failover decision (health checks, lock acquisition, fencing).

2.2.3 Metadata: FSImage and Edit logs — roles and interaction

Two fundamental metadata artifacts exist for HDFS:

- **FSImage**: an on-disk checkpoint (snapshot) of the entire filesystem namespace at a particular point in time.
- **Edit log (edits)**: an append-only log of metadata operations that occurred after the last FSImage checkpoint. Every create, delete, replicate, block allocation is appended to the edit log.

How they work together:

1. Periodically a checkpoint is created: the NameNode applies edits to the FSImage to create a new FSImage and clears the edits (or rolls them).

2. With HA + QJM, the Active NameNode writes edits to the JournalNodes; those edits are the canonical stream Standby will read from.
3. The Standby replays the edits from the JournalNodes against its in-memory namespace so it mirrors the Active's namespace state.

2.2.4 Shared edits with Quorum Journal Manager (QJM)

QJM (JournalNodes) is the recommended shared edit storage for HA:

- The Active NameNode writes each edit to a quorum of JournalNodes (e.g. a majority of 3 or 5).
- JournalNodes persist the edits on local disk and acknowledge back to the NameNode.
- The Standby tails and applies the edits from the JournalNodes so it keeps its in-memory namespace current.

This architecture ensures that edits are replicated reliably and that the Standby is ready to become Active without losing operations.

2.2.5 HDFS write path and how Standby stays updated (step-by-step)

When a client writes a file in HA mode:

1. The client contacts the Active NameNode (logical nameservice resolves to the Active).
2. NameNode allocates one or more blocks and returns a pipeline (list of DataNodes) for each block.
3. The client streams block data to the first DataNode which forwards along the pipeline; data is written to DataNodes and acknowledgements flow back to the client.
4. For each metadata operation (e.g., file create, block allocation) the Active appends an entry to the edit log and writes that entry to the JournalNodes (QJM).
5. The Standby NameNode tails the edits from the JournalNodes and applies them to its in-memory namespace, keeping it consistent with the Active.

2.2.6 Failover: detection, election and fencing

Detection and election:

- ZKFC running on both NameNode hosts performs liveness checks and uses ZooKeeper for leader election. Each ZKFC will attempt to create / acquire an ephemeral znode representing the active role.
- If the ZKFC on the Active host fails to renew its ephemeral znode (because the process crashed or host went down), ZooKeeper removes the ephemeral znode and other ZKFCs detect the removal.
- A ZKFC on the Standby host will then try to acquire the znode and promote its NameNode to Active.

Fencing (preventing split-brain): Fencing is required to ensure the previously-active NameNode cannot continue making writes after a failover (split-brain). Common fencing techniques:

- **SSH-based fencing** — the new active runs a command via SSH to shut down or block the old active NameNode.

2.3 YARN High Availability (HA)

2.3.1 Problem statement and motivation

YARN's ResourceManager (RM) is the cluster's scheduler and central authority for resource allocation. A single RM is a SPOF because if it fails, currently running or pending applications cannot continue normal scheduling. YARN HA replicates the RM service using Active/Standby RMs so scheduling and cluster state survive failure of one RM.

2.3.2 Key components

Active ResourceManager Handles scheduling, maintains application states, assigns containers.

Standby ResourceManager Keeps state synchronized and can be promoted to Active if needed.

RM State Store A pluggable persistent store used to persist cluster and application state so Standby can reconstruct it. Implementations include:

- ZK-based RM state store (ZKRMStateStore) which uses ZooKeeper.
- Filesystem-based RM state store (e.g., on HDFS).

ZooKeeper Used for leader election and ephemeral znodes to indicate which RM is Active.

NodeManagers Worker daemons that register to the Active RM and report container statuses.

ApplicationMasters Per-application components that interact with the Active RM to request containers and report progress.

2.3.3 State synchronization and recovery

YARN RM HA relies on a persistent state store to reconstruct scheduling state after failover. The type of data persisted typically includes:

- Application submission records.
- Application attempts and AM (ApplicationMaster) metadata needed for recovery.
- Container allocations and reservation info required to recover running applications.

When the Active RM writes important state to the RMStateStore, the Standby reads or can reconstruct the same state from the store. On promotion to Active, the Standby has enough information to continue scheduling decisions and allow AMs to reconnect.

2.3.4 Failover process (YARN)

1. ZooKeeper-based leader election: each RM's failover controller registers an ephemeral node. The RM whose controller holds the znode is considered Active.
2. When Active RM fails, the ephemeral znode disappears and ZooKeeper notifies other controllers.
3. A Standby controller acquires the znode and promotes its RM instance to Active.

4. The newly Active RM reads persistent state from the RM-StateStore to rebuild in-memory structures and continues serving RM RPCs.
5. NodeManagers and ApplicationMasters reconnect to the new Active RM.

Chapter 3

Zookeeper znodes used by HDFS and YARN

3.1 How to list znodes

Use the ZooKeeper CLI to inspect znodes:

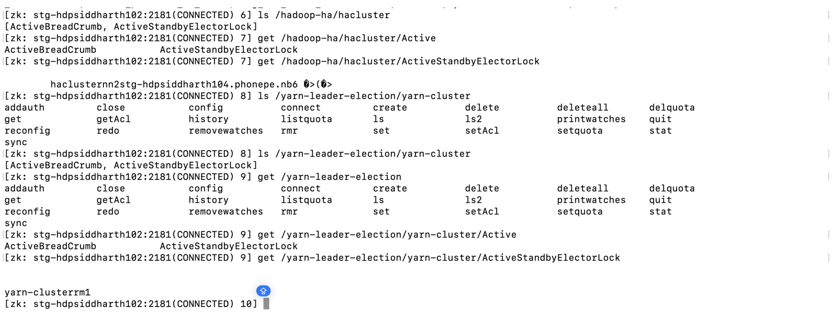
```

# connect to zk
./zkCli.sh -server stg-XXX:2181

# once in the shell:
ls /hadoop-ha/hacluster
ls /yarn-leader-election/yarn-cluster
```

3.1.1 znodes for HDFS HA

- `/hadoop-ha` — This is the root znode that acts as a parent directory to organize all of a Hadoop cluster's high-availability data in ZooKeeper.
- `/hadoop-ha/hacluster/ActiveStandbyElectorLock` — This ephemeral znode functions as a temporary lock that nodes compete to create, with the winner becoming the active leader.
- `/hadoop-ha/hacluster/ActiveBreadCrumb` — This persistent znode stores the address and information of the current active leader.

3.1.2 znodes for YARN HA

- `/yarn-leader-election` — This is the root znode used to organize all high-availability data for a specific YARN cluster in ZooKeeper.
- `/yarn-leader-election/yarn-cluster/ActiveStandbyElectorLock` — This ephemeral znode acts as a temporary lock that Standby ResourceManagers compete for to become the one active leader.
- `/yarn-leader-election/yarn-cluster/ActiveBreadCrumb` — This persistent znode stores the address of the current active ResourceManager, allowing all YARN clients to find it.



```
[zk: stg-hdpsiddharth102:2181(CONNECTED) 6] ls /hadoop-ha/hacluster
[ActiveBreadCrumb, ActiveStandbyElectorLock]
[zk: stg-hdpsiddharth102:2181(CONNECTED) 7] get /hadoop-ha/hacluster/Active
ActiveBreadCrumb           ActiveStandbyElectorLock
[zk: stg-hdpsiddharth102:2181(CONNECTED) 7] get /hadoop-ha/hacluster/ActiveStandbyElectorLock

        haclusternn2stg-hdpsiddharth104.phonepe.nb6 �>(�>
[zk: stg-hdpsiddharth102:2181(CONNECTED) 8] ls /yarn-leader-election/yarn-cluster
addauth         close           config          connect         create          delete          deleteall       delquota
get             getAcl          history         listquota       ls              ls2             printwatches    quit
reconfig        redo            removewatches   rmr             set             setAcl          setquota        stat
sync
[zk: stg-hdpsiddharth102:2181(CONNECTED) 8] ls /yarn-leader-election/yarn-cluster
[ActiveBreadCrumb, ActiveStandbyElectorLock]
[zk: stg-hdpsiddharth102:2181(CONNECTED) 9] get /yarn-leader-election
addauth         close           config          connect         create          delete          deleteall       delquota
get             getAcl          history         listquota       ls              ls2             printwatches    quit
reconfig        redo            removewatches   rmr             set             setAcl          setquota        stat
sync
[zk: stg-hdpsiddharth102:2181(CONNECTED) 9] get /yarn-leader-election/yarn-cluster/Active
ActiveBreadCrumb           ActiveStandbyElectorLock
[zk: stg-hdpsiddharth102:2181(CONNECTED) 9] get /yarn-leader-election/yarn-cluster/ActiveStandbyElectorLock


yarn-clusterrm1
[zk: stg-hdpsiddharth102:2181(CONNECTED) 10]
```

Figure 3.1: Znodes

Chapter 4

MapReduce: Word Count example

4.1 WordCount Java code (classic example)

Save this as `WordCount.java` in your workspace:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.
   FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.
   FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text,
          IntWritable>{

    private final static IntWritable one = new
       IntWritable(1);
    private Text word = new Text();
```

```

    @Override
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
      String[] tokens = value.toString().split("\\s+");
      for (String token : tokens) {
        if (!token.isEmpty()) {
          word.set(token);
          context.write(word, one);
        }
      }
    }
  }

public static class IntSumReducer
      extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
```

```
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(
       args[0]));
    FileOutputFormat.setOutputPath(job, new Path
       (args[1]));
    System.exit(job.waitForCompletion(true) ? 0
       : 1);
  }
}
```

4.2 Compile and run

```
# compile
javac -source 8 -target 8 -classpath "$(hadoop
   classpath)" -d wordcount_classes WordCount.
   java
jar -cvf wordcount.jar -C wordcount_classes/ .

# copy input to HDFS
sudo -u hdfs hdfs dfs -mkdir /input
sudo -u hdfs hdfs dfs -put input.txt /input/

# run the job
sudo -u hdfs yarn jar wordcount.jar WordCount /
   input /output_custom

# view results
sudo -u hdfs hdfs dfs -cat /output_custom/part-r
   -00000
```

4.3 Explanation of MapReduce flow for this job

- **Mapper:** Tokenizes each input line and emits (word, 1).

- **Combiner:** Performs local aggregation on mapper node to reduce network shuffle.
- **Reducer:** Aggregates counts for each unique word across all mappers.
- **Shuffle/Sort:** Keys are partitioned and transferred over the network to reducers; reduce input is sorted by key before reduce() runs.

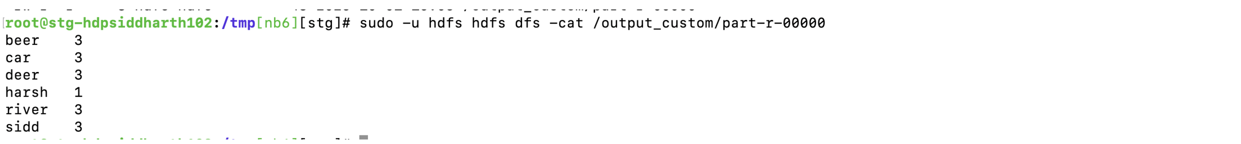


Figure 4.1: Output of map-reduce

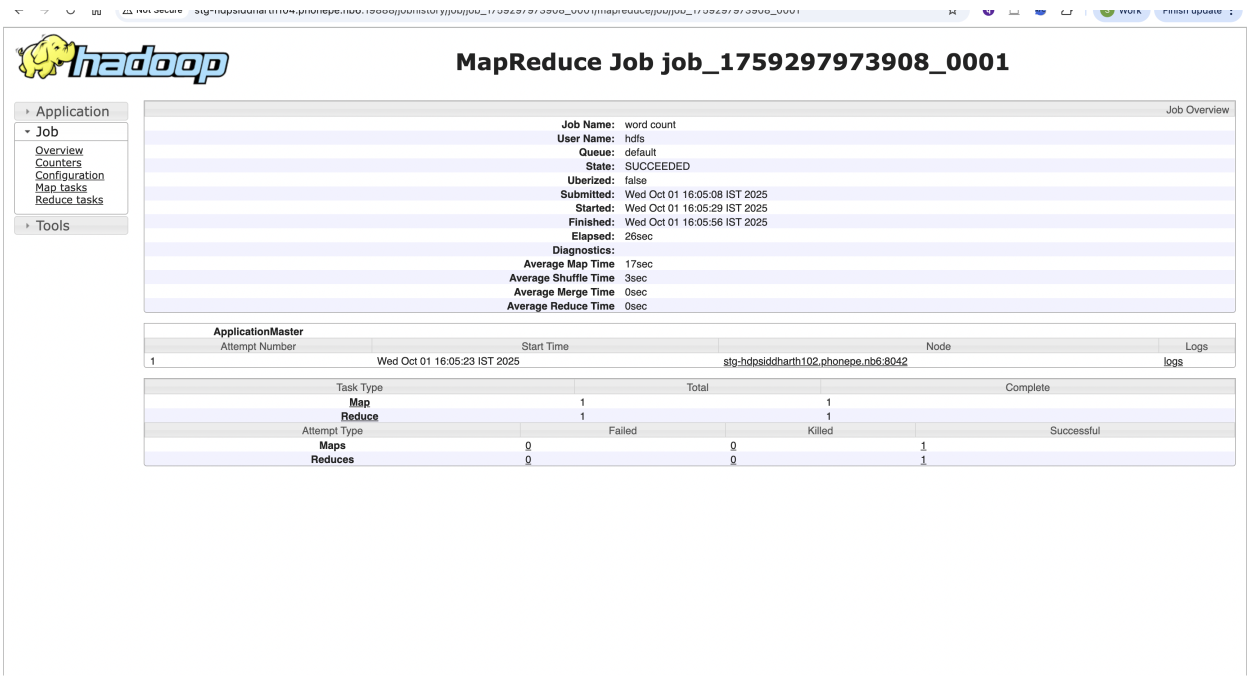


Figure 4.2: Map-reduce on it’s work

Chapter 5

HDFS internals: how a file is stored

5.1 Block-based storage

– HDFS stores files as sequences of blocks (default 128 MB).
– NameNode stores metadata: namespace, file-to-block mapping, block locations (which DataNodes hold each block).
– DataNodes store actual block data and send block reports to NameNode.
– Blocks are replicated (default 3) and placed according to rack-aware policy for fault tolerance and network topology.

5.2 HDFS write sequence

1. Client requests to write file to NameNode.
2. NameNode checks permissions and namespace, then returns block allocation and DataNode pipeline (list of DataNodes for the first block).
3. Client streams data to the first DataNode, which forwards to the second, and so on (pipeline).
4. Each DataNode writes block to disk and sends ack back through pipeline.
5. When block complete, client asks NameNode for next block until file is fully written.
6. NameNode updates metadata and block locations.

5.3 HDFS read sequence

1. Client asks NameNode for block locations for the file.
2. NameNode returns DataNode locations.
3. Client reads data directly from closest DataNode (client may prefer local or same-rack DataNode).
4. If a DataNode fails during read, client fetches from another replica.

Chapter 6

HBase tasks

6.1 Create an HBase table and insert 1000 records

6.1.1 Create table

```

# Start hbase shell

hbase shell

# In HBase shell:

create 'my_table', 'cf'

# exit shell with Ctrl+D or 'quit'
```

6.1.2 Insert 1000 rows

```
(1..1000).each { |i| put 'my_table', i.to_s, 'cf
    :name', "Name#{i}" }
```

6.2 Explain how the table is stored on RegionServers

- HBase table is split into **regions** (contiguous ranges of rowkeys). Each region is served by one RegionServer.
- RegionServer stores regions as a combination of **MemStore** (in-memory writes) and **HFiles** (immutable files stored on HDFS).
- Writes go to WAL (Write-Ahead Log) on HDFS first (for durability), then to MemStore. When MemStore flushes, it writes an HFile to HDFS.
- As HFiles accumulate, compactions merge HFiles to reduce small files and maintain read performance.

6.3 Delete the rows from 111–222

```
(111..222).each { |i| deleteall 'my_table', i.
   to_s }
```

6.4 Check the size of the HBase table on HDFS

```
sudo -u hdfs hdfs dfs -du -h /apps/hbase/data/
   data/default/my_table
```

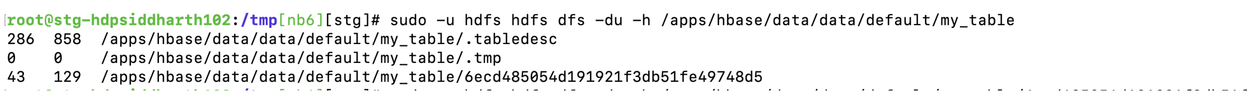


Figure 6.1: Hbase table size

6.5 Run major compaction and check logs

```
# Using HBase shell
major_compact 'my_table'
```

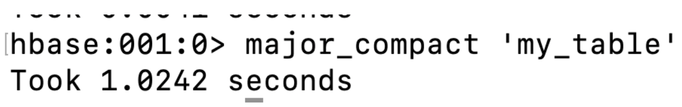


Figure 6.2: Major Compaction

6.6 Optimizations for HBase

- **Use proper rowkey design:** avoid hotspotting. Use salt or reverse key for sequential writes.
- **Tune block cache and MemStore sizes:** increase block cache for read-heavy workloads (reduces HDFS reads). Adjust MemStore to reduce flush frequency for write-heavy workloads.
- **Pre-split regions:** when loading large data, pre-split table into multiple regions so load spreads across RegionServers and avoids a single initial hot region.
- **Use BulkLoad for large ingests:** using HFiles and bulk load avoids WAL/region server CPU overhead and is faster.
- **Tune compaction settings:** configure major/minor compaction thresholds to avoid too many small HFiles (which harm read performance).
- **Monitor GC / RegionServer resources:** ensure RegionServers have adequate RAM tuning to avoid long GC pauses that cause region reassignment.
- **Use Bloom filters and compression:** Bloom filters reduce unnecessary HFile reads; compression reduces disk and IO.

6.7 Snapshot and restore

6.7.1 Snapshot

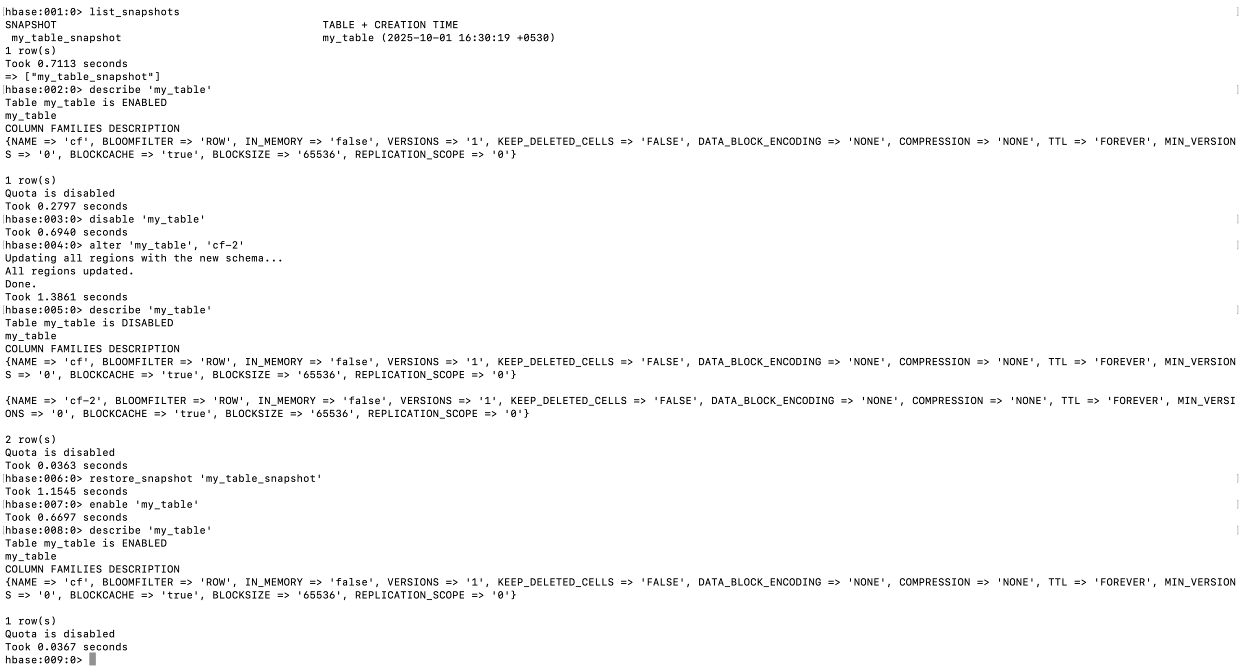
```
hbase shell <<EOF
snapshot 'my_table', 'my_table_snapshot'
EOF
```

6.7.2 Verify snapshot

```
hbase shell <<EOF
list_snapshots
EOF
```

6.7.3 Restore from snapshot (clone to a new table)

```

# create a new table from snapshot (safe
   approach)

hbase shell <<EOF
disable 'my_table'
restore_snapshot 'my_table_snapshot'
enable 'my_table'
EOF
```



```
hbase:001:0> list_snapshots
SNAPSHOT                                    TABLE + CREATION TIME
 my_table_snapshot                          my_table (2025-10-01 16:30:19 +0530)
1 row(s)
Took 0.7113 seconds
=> ["my_table_snapshot"]
hbase:002:0> describe 'my_table'
Table my_table is ENABLED
my_table
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', BLOOMFILTER => 'ROW', IN_MEMORY => 'false', VERSIONS => '1', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', COMPRESSION => 'NONE', TTL => 'FOREVER', MIN_VERSION
S => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}

1 row(s)
Quota is disabled
Took 0.2797 seconds
hbase:003:0> disable 'my_table'
Took 0.6940 seconds
hbase:004:0> alter 'my_table', 'cf-2'
Updating all regions with the new schema...
All regions updated.
Done.
Took 1.3861 seconds
hbase:005:0> describe 'my_table'
Table my_table is DISABLED
my_table
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', BLOOMFILTER => 'ROW', IN_MEMORY => 'false', VERSIONS => '1', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', COMPRESSION => 'NONE', TTL => 'FOREVER', MIN_VERSION
S => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}

{NAME => 'cf-2', BLOOMFILTER => 'ROW', IN_MEMORY => 'false', VERSIONS => '1', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', COMPRESSION => 'NONE', TTL => 'FOREVER', MIN_VERSI
ONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}

2 row(s)
Quota is disabled
Took 0.0363 seconds
hbase:006:0> restore_snapshot 'my_table_snapshot'
Took 1.1545 seconds
hbase:007:0> enable 'my_table'
Took 0.6697 seconds
hbase:008:0> describe 'my_table'
Table my_table is ENABLED
my_table
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', BLOOMFILTER => 'ROW', IN_MEMORY => 'false', VERSIONS => '1', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', COMPRESSION => 'NONE', TTL => 'FOREVER', MIN_VERSION
S => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}

1 row(s)
Quota is disabled
Took 0.0367 seconds
hbase:009:0>
```

Figure 6.3: Snapshot and restore