IBM Software
Information Management

Apache HBase

Hands-On Lab

Table of Contents

1 Introduction

If you're planning to use HBase for your data storage needs, learning the basics of HBase and the various options that BigInsights provides to manipulate data in HBase is important. If you already have data in relational tables, it is necessary to understand how HBase stores and reads data to be able to utilize the strengths of HBase. This hands-on lab takes you through the basics of HBase with information on how HBase servers can be started, managed and monitored, how tables can be created, loaded and backed up, how data can be queried, joined. You'll also see a comparison of various design choices that can help you make decisions that suit your data and resources.

This lab expects a basic knowledge of Jaql, a query language provided with InfoSphere BigInsights. It explores how you can use Jaql to query data from a relational database management system (DBMS) and transform it into a format that can be used both independently by HBase tools or consumed by Jaql to create and load HBase tables. Instead of going over the APIs that HBase provides to query data, it goes over a built-in HBase module available in Jaql.

2 About this Lab

After completing this hands-on lab, you'll be able to:

- Start, stop, monitor HBase server.
- Create HBase tables and load data from a relational DBMS.
- Query data in HBase tables both directly and using the MapReduce framework.
- Backup/Restore HBase data.
- Choose an HBase schema that fits your requirements.

3 Working with IBM InfoSphere BigInsights and Apache HBase

3.1 Background

BigInsights Enterprise Edition provides several options to access HBase such as:

1. HBase shell, which is available stand-alone and also available inside Eclipse environment.
2. Jaql HBase module, bundled with Jaql, which is a query language with a data model based on JavaScript Object Notation (JSON).
3. HBase application, accessible through the BigInsights Web Console. (Before you can launch this application, an administrator must deploy it on your BigInsights cluster and authorize you to access the application.)
4. BigSQL client and JDBC driver which provide SQL access to data residing on HDFS including HBase.

In this lab, you'll use options 1-3 to store and query data that has been offloaded from a sample warehouse into HBase. GSDB database, a rich and realistic database that contains sample data for the Great Outdoors company, which is a fictional outdoor equipment retailer is used for this purpose. For simplicity, we will use only a few tables from this database.

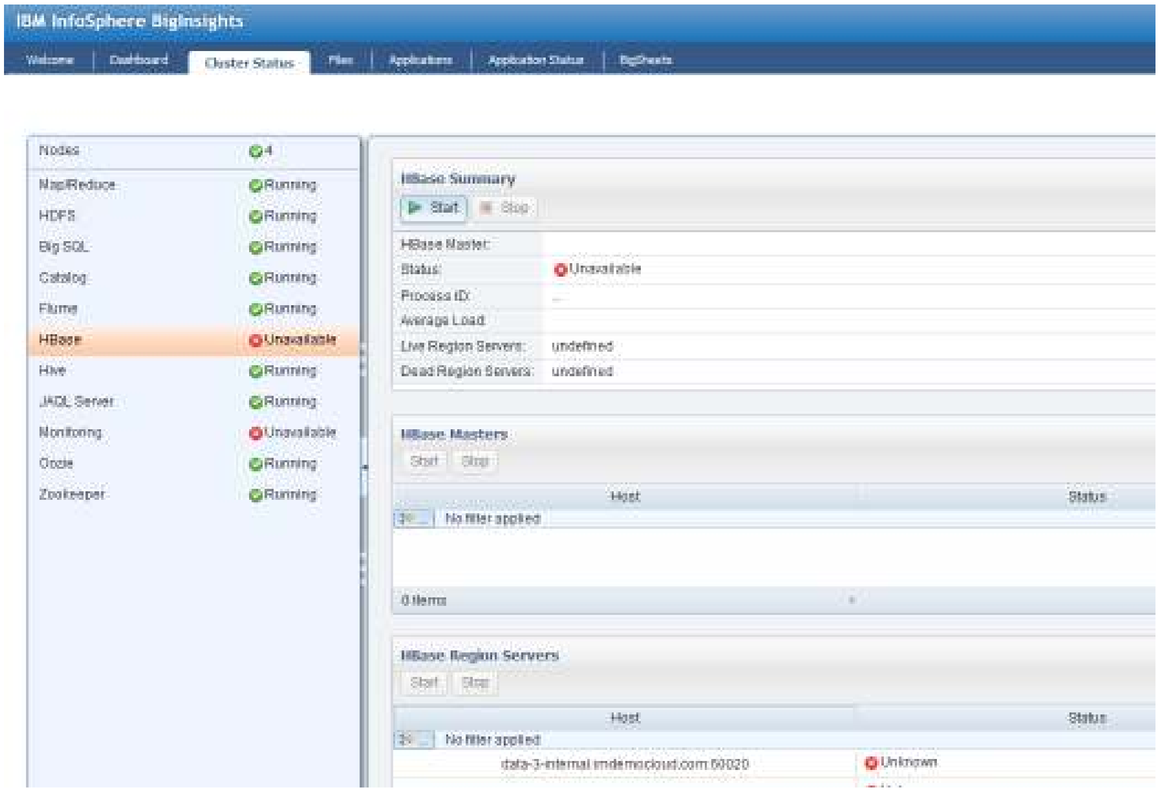
While HBase can be used for a wide variety of data storage scenarios, it is also being used as an alternative for data warehouses. The lab covers various design options and provides information to compare them.

Before diving into design considerations, you need to be familiar with running and monitoring HBase server and basic operations.

3.2 HBase Server

BigInsights Enterprise Edition provides both command-line tools and a user interface to manage HBase server. In this section, we will briefly go over the user interface which is part of BigInsights Web Console.

1. Log into BigInsights web console and open the *Cluster Status* tab. Select HBase to view the status of HBase master and region servers.



Nodes	4
MapReduce	Running
HDFS	Running
Big SQL	Running
Catalog	Running
Flume	Running
HBase	Unavailable
Hive	Running
JAQL Server	Running
Monitoring	Unavailable
Oozie	Running
Zookeeper	Running

HBase Summary

Start Stop

HBase Master:	
Status:	Unavailable
Process ID:	--
Average Load:	
Live Region Servers:	undefined
Dead Region Servers:	undefined

HBase Masters

Start Stop

Host	Status
No filter applied	

0 items

HBase Region Servers

Start Stop

Host	Status
No filter applied	
data-3-internal.imdemocloud.com:60020	Unknown

Make sure to start the HBase server if it is stopped.

2. Use Master and RegionServer web interfaces to visualize tables, regions and other metrics. By default, Master web interface is started on port 60010; RegionServer on 60030.

Master web interface: http://bigdata:60010
RegionServer web interface: http://bigdata:60030

Some interesting information from the web interfaces which we will check later in the lab are:

- HBase root directory – This can be used to find the size of an HBase table.
- List of tables with descriptions.
- For each table, it displays the list of regions with start and end keys. This information can be used to compact or split tables as needed.
- Metrics for each region server. These can be used to determine if there hot regions which are serving the majority of requests to a table. Such regions can be split. It also helps determine the effects and effectiveness of block cache, bloom filters and memory settings. Based on usage, region servers can be added/removed using the commands:

$BIGINSIGHTS_HOME/bin/addnode.sh hbase <node1> <node2>… -type=<nodetype>

$BIGINSIGHTS_HOME/bin/removenode.sh hbase <node1> <node2>… -type=<nodetype>

where node1, node2 etc can be IP address or host name and type can be regionserver or master. BigInsights allows specifying more than one master for handling failover. However, for this lab, only one master will be active.

3. Perform a health check of HBase which is different from the status checks done above. It verifies the health of the functionality.
4. Launch a console as biadmin (you can find a link on the Linux desktop)

$BIGINSIGHTS_HOME/bin/healthcheck.sh hbasez

3.3 Storing and querying data

Before creating tables that correspond to the warehouse data, we will first look at creation of simple tables and understand how HBase stores data.

1. Launch hbase shell using the command:

```
$HBASE_HOME/bin/hbase shell
```

2. Create an HBase table with three column families by using the *create* command:

```
create 'table_1', 'column_family1', 'column_family2', 'column_family3'
```

Tables in HBase are made of rows and columns. All columns in HBase belong to, and are grouped into a particular column family. From the command above you will notice that we have specified 3 column families associated with table_1.

The create command only requires the name of table and one or more column families. Column families must be declared up front at schema definition time whereas columns can be added dynamically to the table. Each row can also have a different set of columns.

It is important to note that all column family members are stored together on the filesystem. The reason this is relevant will become apparent in the following steps.

3. Check the default properties used for column families using *describe command:*

```
hbase(main):002:0> describe 'table_1'
```

```
DESCRIPTION                                                                     ENABLED
 {NAME => 'table_1', FAMILIES => [{NAME => 'column_family1', REPLICATION_SCOPE => '0' true
 , KEEP_DELETED_CELLS => 'false', COMPRESSION => 'NONE', ENCODE_ON_DISK => 'true', BL
 OCKCACHE => 'true', MIN_VERSIONS => '0', DATA_BLOCK_ENCODING => 'NONE', IN_MEMORY =>
  'false', BLOOMFILTER => 'NONE', TTL => '2147483647', VERSIONS => '3', BLOCKSIZE =>
 '65536'}, {NAME => 'column_family2', REPLICATION_SCOPE => '0', KEEP_DELETED_CELLS =>
  'false', COMPRESSION => 'NONE', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true', MIN
 _VERSIONS => '0', DATA_BLOCK_ENCODING => 'NONE', IN_MEMORY => 'false', BLOOMFILTER =
 > 'NONE', TTL => '2147483647', VERSIONS => '3', BLOCKSIZE => '65536'}, {NAME => 'col
 umn_family3', REPLICATION_SCOPE => '0', KEEP_DELETED_CELLS => 'false', COMPRESSION =
 > 'NONE', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true', MIN_VERSIONS => '0', DATA_
 BLOCK_ENCODING => 'NONE', IN_MEMORY => 'false', BLOOMFILTER => 'NONE', TTL => '21474
 83647', VERSIONS => '3', BLOCKSIZE => '65536'}]}
1 row(s) in 0.0630 seconds
```

As you can see here, there are several properties that are configurable for each column family created. We won't go into every single property, but over the next couple set of steps, we will look at modifying the highlighted options for *column_family1.*

4. Specify compression for a column family in the table. For HBase packaged with BigInsights, only gzip compression can be used out of the box. Use *alter* command. To alter a table, it has to be disabled first.

```
hbase(main):003:0> disable 'table_1'
hbase(main):004:0> alter 'table_1', {NAME => 'column_family1', COMPRESSION => 'GZ'}
```

SNAPPY and LZO are the two other compression algorithms that are supported by HBase but would need extra configuration. Note that GZIP is slower than LZO because it generally compresses better. This would add to query latency, but in some cases it better compression may be preferred.

Generally, production systems should use compression with their ColumnFamily definitions. Although, the data size will be smaller on disk, when accessed in memory for example, it will be inflated. Therefore just using compression will not eliminate the problem of over-sized column family names or over-sized column names. Care must be taken when determining the schema. We will mention some of these best practices later on as well.

5. Specify IN_MEMORY option for a column family that will be queried frequently.

```
hbase(main):005:0> alter 'table_1', {NAME => 'column_family1', IN_MEMORY => 'true'}
```

This does not ensure that all the data will always be in memory. Data will always be persisted to disk. It only gives priority for the corresponding data within the column family to stay in the cache longer.

6. Specify the required number of versions as 1 for the column family.

```
hbase(main):006:0> alter 'table_1', {NAME => 'column_family1', VERSIONS => 1}
```

HBase table cells (tuples) are versioned. That means multiple versions of a cell can be stored. If your application does not require multiple versions, specify VERSIONS => 1. When mapping data from relational DBMS, multiple versions may not be required. By default, 3 versions of a value will be stored.

It is possible to have multiple cells where the row keys and columns are identical and only differs in its version dimension. Versions are specified using a long integer, i.e., timestamp.

7. Run *describe against the table* again to verify the above changes. Before you can load data, you will have to *enable* the table.

```
hbase(main):007:0> enable 'table_1'
hbase(main):008:0> describe 'table_1'
```

```
DESCRIPTION                                                                      ENABLED
 {NAME => 'table_1', FAMILIES => [{NAME => 'column_family1', REPLICATION_SCOPE => '0' true
 , KEEP_DELETED_CELLS => 'false', COMPRESSION => 'GZ', ENCODE_ON_DISK => 'true', BLOC
 KCACHE => 'true', MIN_VERSIONS => '0', DATA_BLOCK_ENCODING => 'NONE', IN_MEMORY => '
 true', BLOOMFILTER => 'NONE', TTL => '2147483647', VERSIONS => '1', BLOCKSIZE => '65
 536'}, {NAME => 'column_family2', REPLICATION_SCOPE => '0', KEEP_DELETED_CELLS => 'f
 alse', COMPRESSION => 'NONE', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true', MIN_VE
 RSIONS => '0', DATA_BLOCK_ENCODING => 'NONE', IN_MEMORY => 'false', BLOOMFILTER => '
 NONE', TTL => '2147483647', VERSIONS => '3', BLOCKSIZE => '65536'}, {NAME => 'column
 _family3', REPLICATION_SCOPE => '0', KEEP_DELETED_CELLS => 'false', COMPRESSION => '
 NONE', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true', MIN_VERSIONS => '0', DATA_BLO
 CK_ENCODING => 'NONE', IN_MEMORY => 'false', BLOOMFILTER => 'NONE', TTL => '21474836
 47', VERSIONS => '3', BLOCKSIZE => '65536'}]}
1 row(s) in 0.0640 seconds
```

8. Insert data using put command. As mentioned earlier, each row can have a different set of columns. Below are a set of put commands which will insert two rows with a different set of column names.

```
hbase(main):009:0> put 'table_1', 'row1', 'column_family1:c11', 'r1v11'
hbase(main):010:0> put 'table_1', 'row1', 'column_family1:c12', 'r1v12'
hbase(main):011:0> put 'table_1', 'row1', 'column_family2:c21', 'r1v21'
hbase(main):012:0> put 'table_1', 'row1', 'column_family3:c31', 'r1v31'
hbase(main):013:0> put 'table_1', 'row2', 'column_family1:d11', 'r2v11'
hbase(main):014:0> put 'table_1', 'row2', 'column_family1:d12', 'r2v12'
hbase(main):015:0> put 'table_1', 'row2', 'column_family2:d21', 'r2v21'
```

The following figure shows a strictly CONCEPTUAL view of table_1:

Row Key	**ColumnFamily *column_family1***	**ColumnFamily *column_family2***	**ColumnFamily *column_family3***	**timestamp**
row1	*column_family1:c11= r1v11*			*1358178763530*
row1	*column_family1:c12= r1v12*			*1358178763635*
row1		*column_family2:c21= r1v21*		*1358178763673*
row1			*column_family3:c31= r1v31*	*1358178845385*
row2	*column_family1:*d11=r2v11			*1358178856624*
row2	*column_family1:*d12=r2v12			*1358178856676*
row2		*column_family2:*d21=r2v21		*1358178856706*

From a purely conceptual perspective, tables may be viewed as a sparse set of rows as shown above. However, physically they are stored on a per-column family basis as mentioned previously.

It is very important to note in the diagram above that the empty cells are not stored since they need not be in a column-oriented storage format. This is different from RDBM's in the sense HBase doesn't store nulls.

The physical view is as follows:

Row Key	**ColumnFamily *column_family1***	**timestamp**
row1	*column_family1:c11= r1v11*	*1358178763530*
row1	*column_family1:c12= r1v12*	*1358178763635*
row2	*column_family1:*d11=r2v11	*1358178856624*
row2	*column_family1:*d12=r2v12	*1358178856676*

Row Key	**ColumnFamily *column_family2***	**timestamp**
row1	*column_family2:*c21= *r1v21*	*1358178763673*
row2	*column_family2:*d21=r2v21	*1358178856706*

Row Key	**ColumnFamily *column_family3***	**timestamp**
row1	*column_family3:c31= r1v31*	*1358178845385*

9. Verify two rows have been inserted using the *count* command. The *count* command works for small tables. For much larger tables, you can use the RowCounter mapreduce job which we will use later in the lab.

```
hbase(main):016:0> count 'table_1'
```

```
2 row(s) in 0.0350 seconds
```

10. To view the data, you may use *get,* which returns attributes for a specified row, or *scan* which allows iteration over multiple rows.

```
hbase(main):017:0> get 'table_1', 'row1'
```

```
COLUMN                             CELL
 column_family1:c11                timestamp=1370827678574, value=r1v11
 column_family1:c12                timestamp=1370827686016, value=r1v12
 column_family2:c21                timestamp=1370827690946, value=r1v21
 column_family3:c31                timestamp=1370827697126, value=r1v31
4 row(s) in 0.0290 seconds
```

In the above example, the 4 rows correspond to the values of 4 columns for row1.

11. Now, run the scan command to see a per-row listing of values.

```
hbase(main):018:0> scan 'table_1'
```

```
ROW                                 COLUMN+CELL
 row1                               column=column_family1:c11, timestamp=1370827678574, value=r1v11
 row1                               column=column_family1:c12, timestamp=1370827686016, value=r1v12
 row1                               column=column_family2:c21, timestamp=1370827690946, value=r1v21
 row1                               column=column_family3:c31, timestamp=1370827697126, value=r1v31
 row2                               column=column_family1:d11, timestamp=1370827702989, value=r2v11
 row2                               column=column_family1:d12, timestamp=1370827709205, value=r2v12
 row2                               column=column_family2:d21, timestamp=1370827715400, value=r2v21
2 row(s) in 0.0260 seconds
```

The above scan results show that HBase tables do not require a set schema. This is good for some applications that need to store arbitrary data. To put this in other words, HBase does not store null values. If a value for a column is null (e.g. values for d11, d12, d21 are null for row1), it is not stored. This is one aspect that makes HBase work well with sparse data.

Note that, in addition to the actual column value (*r1v11*), each result row has the row key value (*row1*), column family name (*column_family1*), column qualifier/column (*c11*) and timestamp. These pieces of information are also stored physically for each value. Having a large number of columns with values for all rows (in other words, dense data) would mean this information gets repeated. Also, larger row key values, longer column family and column names would increase the storage space used by a table.

Tips:

Try to use smaller row key values, column family and qualifier names.

Try to use fewer columns if you have dense data.

12. HBase does not have an *update* command or API. An *update* is same as *put* with another set of column values for the same row key. Update values in column_family1 (with VERSIONS => 1) and column_family2 (with VERSIONS => 3).

```
hbase(main):019:0> put 'table_1', 'row1', 'column_family1:c11', 'n1v11'
hbase(main):020:0> put 'table_1', 'row2', 'column_family1:d11', 'n2v11'
hbase(main):021:0> put 'table_1', 'row2', 'column_family2:d21', 'n2v21'
```

A scan will show rows updated values.

```
hbase(main):022:0> scan 'table_1'
```

```
ROW                                COLUMN+CELL
 row1                              column=column_family1:c11, timestamp=1370827903257, value=n1v11
 row1                              column=column_family1:c12, timestamp=1370827686016, value=r1v12
 row1                              column=column_family2:c21, timestamp=1370827690946, value=r1v21
 row1                              column=column_family3:c31, timestamp=1370827697126, value=r1v31
 row2                              column=column_family1:d11, timestamp=1370827909185, value=n2v11
 row2                              column=column_family1:d12, timestamp=1370827709205, value=r2v12
 row2                              column=column_family2:d21, timestamp=1370827915441, value=n2v21
2 row(s) in 0.0320 seconds
```

Notice the old versions of the cells are not shown in this result set. By default, if you don't specify an explicit version when doing a scan, the cell(s) whose version has the largest value will be returned.

13. View values with different versions stored for column_family2 by explicitly requesting multiple versions.

```
hbase(main):023:0> scan 'table_1', {VERSIONS => 2}
```

```
ROW                                COLUMN+CELL
 row1                              column=column_family1:c11, timestamp=1370827903257, value=n1v11
 row1                              column=column_family1:c12, timestamp=1370827686016, value=r1v12
 row1                              column=column_family2:c21, timestamp=1370827690946, value=r1v21
 row1                              column=column_family3:c31, timestamp=1370827697126, value=r1v31
 row2                              column=column_family1:d11, timestamp=1370827909185, value=n2v11
 row2                              column=column_family1:d12, timestamp=1370827709205, value=r2v12
 row2                              column=column_family2:d21, timestamp=1370827915441, value=n2v21
 row2                              column=column_family2:d21, timestamp=1370827715400, value=r2v21
2 row(s) in 0.0340 seconds
```

The first row in the result for row2|column_family2:d21 is the new value and the second row is the old value. Don't forget that column family 1 versioning was set to 1. Therefore, we only see the latest (and only) version for that column family.

3.4 Sample data and application

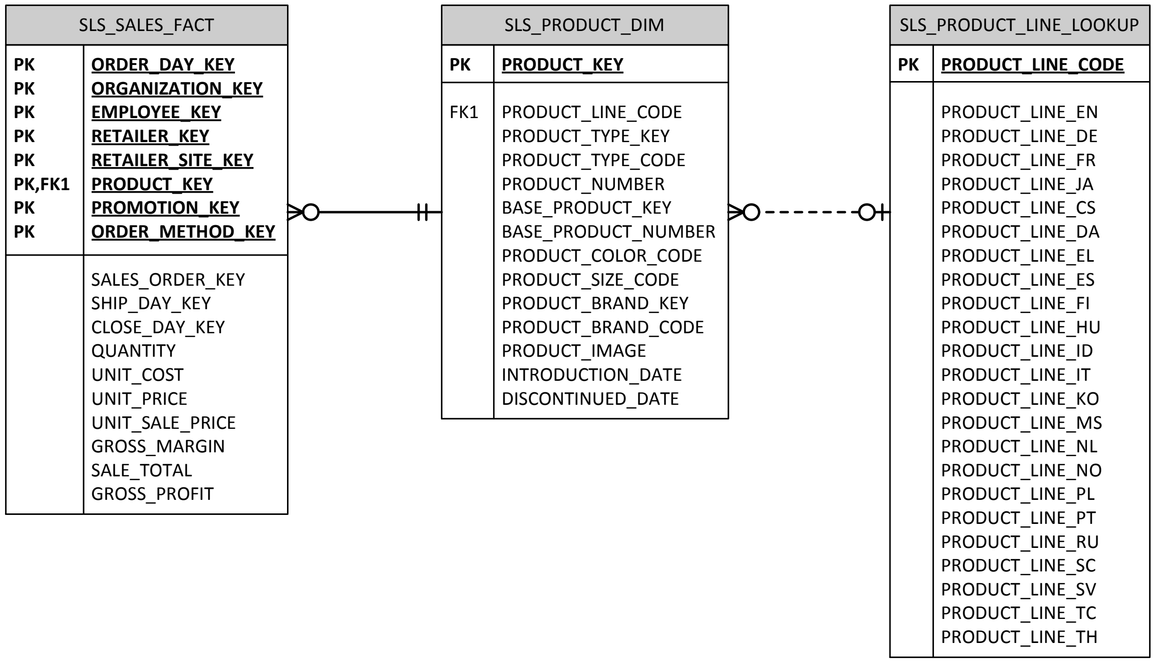
In this lab, we will use three tables from GSDB's GOSALESDW database.

- SLS_SALES_FACT
- SLS_PRODUCT_DIM

- SLS_PRODUCT_LINE_LOOKUP

3.4.1 Schema

The details of the tables along with primary and foreign key information are listed below:



3.4.2 Cardinality

Another aspect to consider in the design is the size of the each table.

Table	**Cardinality**
SLS_SALES_FACT	446023
SLS_PRODUCT_DIM	274
SLS_PRODUCT_LINE_LOOKUP	5

For checking how the database performs with bigger dataset, we will also check with a scale of 10.

3.5 Offloading data from a warehouse table into HBase

Among the numerous options that HBase provides to load data, bulk loading is very useful for offloading warehouse data. In such scenarios, an administrator would have exported the warehouse data into some delimited format. Let us assume, we already have data files in tsv format from the relations mentioned in the previous section. To load this data into HBase, we can use the ImportTsv and CompleteBulkLoad utilities provided by HBase.

ImportTsv is a utility that will load data in TSV format into HBase. It has two distinct usages: (A) loading data from TSV format in HDFS into HBase via Puts, and (B) preparing StoreFiles to be loaded via the completebulkload.

Let us try this for one table SLS_SALES_FACT to see how it works, first loading data via Puts (non-bulk load), and than later using StoreFiles with CompleteBulkLoad (bulk load).

In this section, we will also try to understand how HBase handles row keys and some pitfalls that users may encounter when moving data from relational database to HBase tables. We will also try some useful options like pre-creating regions to see how it can help with data loading and queries.

3.5.1 One-to-one mapping:

In this step, we will use a one-to-one mapping of the columns in a relational table to an HBase table row key and columns. This is not a recommended approach. The goal of this exercise is to demonstrate the inefficiency and pitfalls that can occur with such a mapping.

1. Examine the number of lines (rows) in the data file. (You need to quit the HBase console with ‘quit’ or open up another linux terminal).

```
wc -l ~/labs/hbase/gosalesdw/data/SLS_SALES_FACT.txt
```

```
446023 /home/biadmin/labs/hbase/gosalesdw/data/SLS_SALES_FACT.txt
```

We would expect to have same number of rows once in HBase.

2. Copy data file to be loaded into HBase onto hdfs:

```
$BIGINSIGHTS_HOME/IHC/bin/hadoop fs -copyFromLocal
~/labs/hbase/gosalesdw/data/SLS_SALES_FACT.txt
hdfs:/hbase_lab/sales_fact/SLS_SALES_FACT.txt
```

3. Create a table sales_fact with a single column family that stores only one verision of its values.

```
$HBASE_HOME/bin/hbase shell
hbase(main):001:0> create 'sales_fact', {NAME => 'cf', VERSIONS => 1}
```

4. Use ImportTsv tool to load data. Note this step may take several minutes.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,cf:ok,cf:ek,cf:rk,cf:rsk,cf:pdk,cf:pmk,cf:omk,cf:sok,cf:s
dk,cf:cdk,cf:q,cf:uc,cf:up,cf:usp,cf:gm,cf:st,cf:gp -Dimporttsv.skip.bad.lines=false
'sales_fact' hdfs:/hbase_lab/sales_fact/SLS_SALES_FACT.txt
```

As mentioned previously this ImportTsv operation loads data into HBase via Puts (i.e., non-bulk loading):

5. Now, count the rows in the results

```
$HBASE_HOME/bin/hbase shell
hbase(main):001:0> count 'sales_fact'
```

```
440 row(s) in 0.7300 seconds
```

Observations

- There were no errors during the load but there it is obviously apparent that there are fewer rows in the HBase table than were in the original file. This is because HBase treats updates (i.e., versioning) differently. HBase enforces a unique row key. However, if a new put request (insert) comes in with same row key value, the row is updated. This is different from most relational database systems where users expect an error in such scenarios. This makes it important to ensure the rows have highly unique key values. Otherwise, there will be no indication of which rows were omitted. Shortly you will go through an approach which can handle such cases using JAQL.

3.5.2 Many-to-one mapping for row key:

In this step, we will use a many-to-one mapping of the columns in a relational table to an HBase table row key. The remaining columns are handled as individual columns in HBase table. We will refer to such an HBase row key as composite key. The composite key will have all columns which formed the primary key of the relational table. The values of these columns will be separated using something that will not occur in any of the individual data values. For example, we will use '/' as the separator. Now we need to transform the input data into composite key values.

A tool that is well suited for this purpose is Jaql. Jaql is a query language that can read data in many formats and perform aggregation, validation and filtering. The HBase module in Jaql provides wrappers around the HBase API to create, write and read HBase tables. The input from other sources like relational databases can be read using Jaql and piped to the HBase module functions for data load.

1. Launch the Jaql shell by using the command:

```
$JAQL_HOME/bin/jaqlshell
```

2. Define a variable to read the original data file using the Jaql "del" function. This converts each line of the file into a Json record. If you are new to Jaql, use the pipe operator and top command to print out and visualize a record.

```
jaql> sf = localRead(del("/hbase_lab/sales_fact/SLS_SALES_FACT.txt",
delimiter='\t'));
```

3. You can optionally use the top command to print and visualize a record.

```
jaql> sf -> top 1;
```

Results should look as follows:

```
[
  [
    "20040112",
    "11101",
    "4001",
    "6737",
    "5185",
    "30126",
    "5501",
    "602",
    "194778",
    "20040119",
    "20040119",
    "587",
    "34.90",
    "76.86",
    "71.48",
    "0.5118",
    "41958.76",
    "21472.46"
  ]
]
```

4. Transform the records into the required format. For the first element, which will be used as the HBase row key, concatenate the values of the columns that form the primary key of the table: `ORDER_DAY_KEY`, `ORGANIZATION_KEY`, `EMPLOYEE_KEY`, `RETAILER_KEY`, `RETAILER_SITE_KEY`, `PRODUCT_KEY`, `PROMOTION_KEY`, `ORDER_METHOD_KEY`. This will comprise our composite key. For the remaining columns, use names that we can later use in the output schema.

```
jaql> sft = sf -> transform { key:
strcat($[0],"/",$[1],"/",$[2],"/",$[3],"/",$[4],"/",$[5],"/",$[6],"/",$[7]), sok: $[8], sdk: $[9],
cdk: $[10], q: $[11], uc: $[12], up: $[13], usp: $[14], gm: $[15], st: $[16], gp: $[17] };
```

5. Write the transformed result into HDFS using the “del” function. Specify the schema based on the element names used in the transform step above.

```
jaql> sft -> write(del("/hbase_lab/sales_fact_composite/SLS_SALES_FACT_COMPOSITE_KEY.txt",
delimiter='\t', quoted = false, schema=schema { key, sok, sdk, cdk, q, uc, up, usp, gm,
st, gp }));
```

6. Verify the contents of output file using the “top” command to limit for only a few rows.

```
jaql> read(del("/hbase_lab/sales_fact_composite/SLS_SALES_FACT_COMPOSITE_KEY.txt",
delimiter='\t')) -> top 1;
```

```
[
  [
    "20040112/11101/4001/6737/5185/30126/5501/602",
    "194778",
    "20040119",
    "20040119",
    "587",
    "34.90",
    "76.86",
    "71.48",
    "0.5118",
    "41958.76",
    "21472.46"
  ]
]
```

Do a count on the new file we have created to ensure the same number of records are here as in the original file.

```
jaql> count(read(del("/hbase_lab/sales_fact_composite/SLS_SALES_FACT_COMPOSITE_KEY.txt",
delimiter='\t')));
```

```
446023
```

7. Now we have the file with composite row keys which should be unique since it is based on the primary keys of the table. Let us try to load this into an HBase table ‘sf_comp_key’. This table will have fewer columns but a longer row key. We will now use the two step process for bulk-loading. (A) prepare the StoreFiles using ImportTsv and (B) bulk load them.
   a) Use ImportTsv to prepare the StoreFiles for bulk loading.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,cf:sok,cf:sdk,cf:cdk,cf:q,cf:uc,cf:up,cf:usp,cf:gm,cf:st,
cf:gp -Dimporttsv.bulk.output=/hbase_lab/sf_comp_output -Dimporttsv.skip.bad.lines=false
sf_comp_key /hbase_lab/sales_fact_composite/SLS_SALES_FACT_COMPOSITE_KEY.txt
```

b) Use CompleteBulkLoad tool to bulk-load the StoreFiles from previous step.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles
/hbase_lab/sf_comp_output 'sf_comp_key'
```

Note that this step creates the HBase table if it does not exist. Also, it is very quick as bulk load simply copies over the prepared files from HDFS. This works well for large datasets when the data can be prepared before-hand.

8. Verify that the new table has the right number of rows. This time, use the RowCounter utility which uses a mapreduce job to perform the count.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter -
Dhbase.client.scanner.caching=10000 'sf_comp_key'
```

You should see “ROWS=446023”:

```
13/06/09 22:18:23 INFO mapred.JobClient:   org.apache.hadoop.hbase.mapreduce.RowCounter$RowCounterMapper$Counters
13/06/09 22:18:23 INFO mapred.JobClient:     ROWS=446023
```

a) This step takes a while and works best for big tables. For smaller tables, the overhead of spawning mapreduce job would outweigh the benefit.
b) The setting `–Dhbase.client.scanner.caching=10000` is used to indicate to the HBase scan API to retrieve 10000 rows in a single RPC call. This reduces the number of network roundtrips and reduces the time taken for the scan operations.
c) To see the difference, you may run without this setting. Instead of ~25s, it now takes ~100s.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter 'sf_comp_key'
```

d) Increasing the scanner caching to a very high value would result in increased memory usage and could lead to potential out of memory errors. If the application processes the retrieved results before getting more and this step is slow, it may also lead to scanner timeout exceptions.

3.5.3 Precreating regions

In the previous sections, we’ve observed that HBase stores rows of data in tables.

Once they reach a set limit, tables are sharded into **regions** (chunks of rows) and these regions are distributed amongst the cluster and made available via **RegionServers**. They are used as a mechanism to distribute the read and write load across the cluster. A table can consist of many regions, hosted by many region servers (although, every region is hosted by only a *single* region server).

When you first create a table as we did in the previous sections, HBase creates only one region for the table. Thus, any requests will go trough only one region server, regardless of the number of region servers that exist. This is why you can't utilize the whole capacity of the cluster when loading data. Therefore, in some scenarios like bulk loading, it can be much more efficient to pre-create regions so that the load operation can take place in parallel.

The sales data we are using is for 4 years ranging from 2004-2007. We can pre-create regions by specifying splits in create table command. We will repeat the bulk load exercise with the same table, SLS_SALES_FACT, with 10 times more data. This exercise is intended to show the effectiveness of pre-creating regions as well as to understand how HBase queries scale.

1. From HBase shell, create a table sales_fact_10x by specifying splits.

```
hbase(main):001:0> create 'sales_fact_10x', 'cf', {SPLITS => ['2004', '2005',
'2006', '2007']}
```

2. Check the details of table from the Master web interface using URL noted in HBase server section. Click on the table name in the list of tables in the master's web page. It shows 5 pre-created regions.

http://bigdata:60010/master-status

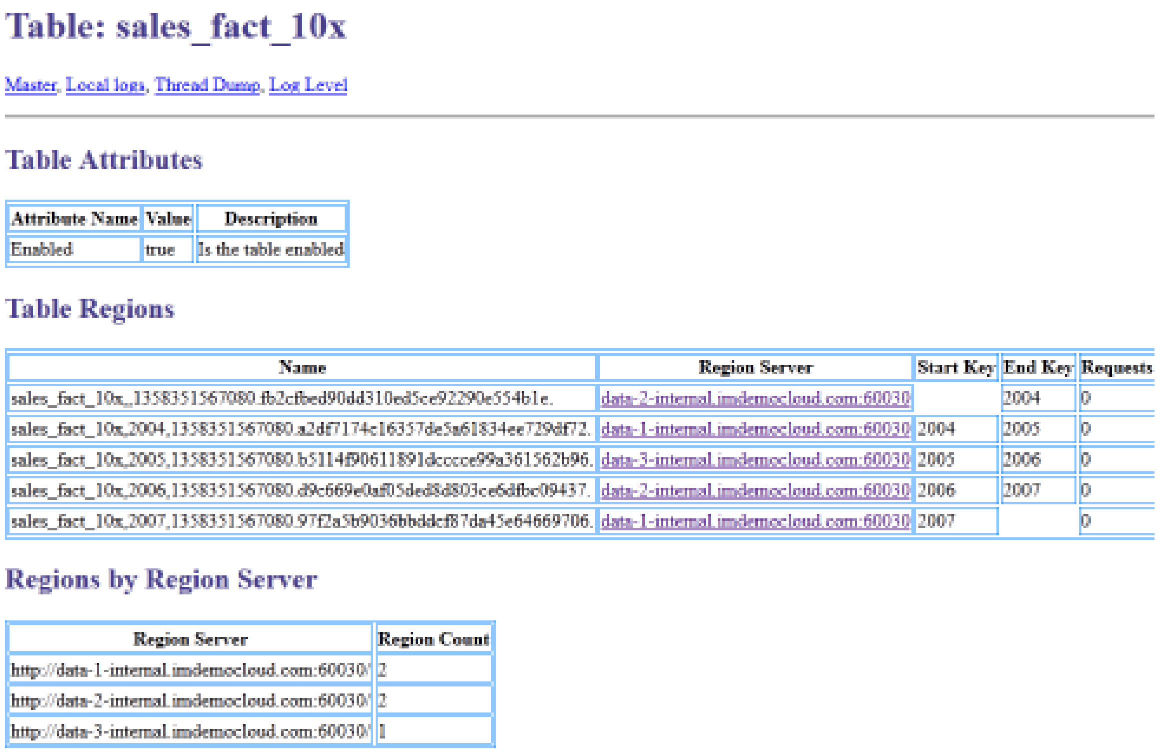


Table: sales_fact_10x

Master, Local logs, Thread Dump, Log Level

Table Attributes

Attribute Name	Value	Description
Enabled	true	Is the table enabled

Table Regions

Name	Region Server	Start Key	End Key	Requests
sales_fact_10x,,1358351567080.fb2cfbed90dd310ed5ce92290e554b1e.	data-2-internal.imdemocloud.com:60030		2004	0
sales_fact_10x,2004,1358351567080.a2df7174c16357de5a61834ee729df72.	data-1-internal.imdemocloud.com:60030	2004	2005	0
sales_fact_10x,2005,1358351567080.b5114f90611891dcccce99a361562b96.	data-3-internal.imdemocloud.com:60030	2005	2006	0
sales_fact_10x,2006,1358351567080.d9c669e0af05ded8d803ce6dfbc09437.	data-2-internal.imdemocloud.com:60030	2006	2007	0
sales_fact_10x,2007,1358351567080.97f2a5b9036bbddcf87da45e64669706.	data-1-internal.imdemocloud.com:60030	2007		0

Regions by Region Server

Region Server	Region Count
http://data-1-internal.imdemocloud.com:60030/	2
http://data-2-internal.imdemocloud.com:60030/	2
http://data-3-internal.imdemocloud.com:60030/	1

3. Copy the data file to hdfs. In the prevous section, we used jaql function to to read from local mode. Since the data set is now x10, reading in local mode will not work well for such a big file.

```
$BIGINSIGHTS_HOME/IHC/bin/hadoop fs -copyFromLocal
~/labs//hbase/gosalesdw/data10/SLS_SALES_FACT.txt
hdfs:/hbase_lab/sales_fact10/SLS_SALES_FACT.txt
```

4. Open a JAQL shell and define a variable to read the original data file using Jaql “del” function.

```
jaql> sf10=read(del("/hbase_lab/sales_fact10/SLS_SALES_FACT.txt", delimiter='\t'));
```

5. Transform the record into the required format as we did previously. For the first element, which will be used as the HBase row key, concatenate the values of the columns that form the primary key of the table - `ORDER_DAY_KEY`, `ORGANIZATION_KEY`, `EMPLOYEE_KEY`, `RETAILER_KEY`, `RETAILER_SITE_KEY`, `PRODUCT_KEY`, `PROMOTION_KEY`, `ORDER_METHOD_KEY`. This will comprise our composite key. For the remaining columns, use names that we can later use in the output schema.

```
jaql> sft10 = sf10 -> transform {key:
strcat($[0],"/",$[1],"/",$[2],"/",$[3],"/",$[4],"/",$[5],"/",$[6],"/",$[7]),sok: $[8],
sdk: $[9], cdk: $[10], q: $[11], uc: $[12], up: $[13],usp: $[14], gm: $[15], st: $[16],
gp: $[17]};
```

6. Write the transformed result into HDFS using the “del” function. Specify the schema based on the element names used in the transform step above.

```
jaql> sft10 ->
write(del("/hbase_lab/sales_fact_composite10/SLS_SALES_FACT_COMPOSITE_KEY.txt",
delimiter='\t', quoted = false, schema=schema { key, sok, sdk, cdk, q, uc, up, usp, gm,
st, gp }));
```

You may track the map reduce jobs using BigInsights web console or directly using the job tracker URL job tracker url.

7. Now we have the file with composite row keys which should be unique since it is based on the primary keys of the table. Let us try to load this into an HBase table ‘sales_fact_10x’. This table will have fewer columns but a longer row key. We will now use the two step process for bulk-loading. (A) prepare the StoreFiles using ImportTsv and (B) bulk load them.

Switch to the Linux shell

   a) Use ImportTsv to prepare the StoreFiles for bulk loading.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,cf:sok,cf:sdk,cf:cdk,cf:q,cf:uc,cf:up,cf:usp,cf:gm,cf:st,
cf:gp -Dimporttsv.bulk.output=/hbase_lab/sf_10_output -Dimporttsv.skip.bad.lines=false
sales_fact_10x /hbase_lab/sales_fact_composite10/SLS_SALES_FACT_COMPOSITE_KEY.txt
```

b) Use CompleteBulkLoad tool to bulk-load the StoreFiles from previous step.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles
/hbase_lab/sf_10_output 'sales_fact_10x'
```

8. Verify that the new table has the right number of rows (x10 what we have seen before). Use the RowCounter utility to utilize mapreduce to perform the count.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter -
Dhbase.client.scanner.caching=10000 'sales_fact_10x'
```

In the results, you should see “ROWS=4460230”

```
13/06/09 22:36:54 INFO mapred.JobClient:   org.apache.hadoop.hbase.mapreduce.RowCounter$RowCounterMapper$Counters
13/06/09 22:36:54 INFO mapred.JobClient:     ROWS=446023
```

3.5.4 Dividing columns among column families

The concept of column families is very beneficial; however the cost of operations will grow linearly with the number of column families that exist. This is because of certain operation (such as flushing and compactions) take place on a per region basis. Even if only one column family is responsible for triggering such operations, all of the column families will need to be affected.

Also consider cardinality (i.e., number of rows) where multiple ColumnFamilies exist in a single table. If two column famililes have a vast difference in cardinality, this can make mass scans less efficient for the column family with a lower cardinality. This is because all of the regions that contain the table data will need to be examined.

Currently, HBase will not performance well with more than 2 or 3 Column families. Therefore, in general, **the recommendation is to keep the number of column families small if possible**. Only introduce additional column families in the cases where queries involve mutually exclusive subset of columns (i.e. you query one column family or the other but usually not both at the one time.)

Regardless of this, the concept behind column families can still be quite advantageous to implement for some specific scenarios; mainly configuring various parts of the table differently. For example:

- requiring some columns to have a time to live (TTL), while others should never expire
- versioning on only certain columns of the table

We will work with the scenario of introducing a column family in the case where our data access is column scoped. For e.g, let us assume that PRODUCT_IMAGE column in the SLS_PRODUCT_DIM table is always queried alone and the size of this column is considerably large compared to the combined size of all other columns. In this scenario, it would be good to create a new column family to store PRODUCT_IMAGE column. The queries can specify the column family for a scan.

1. Check number of lines (274) in the data file.

```
wc -l ~/labs/hbase/gosalesdw/data/SLS_PRODUCT_DIM.txt
```

```
274 /home/biadmin/labs/hbase/gosalesdw/data/SLS_PRODUCT_DIM.txt
```

2. Create the data file on HDFS.

```
$BIGINSIGHTS_HOME/IHC/bin/hadoop fs -copyFromLocal
~/labs/hbase/gosalesdw/data/SLS_PRODUCT_DIM.txt
hdfs:/hbase_lab/sales_product_dim/SLS_PRODUCT_DIM.txt
```

3. From HBase shell, create a table SALES_PRODUCT_DIM with two column families – cf1 to hold all columns except PRODUCT_IMAGE which will be in cf2.

```
hbase(main):001:0> create 'sales_product_dim', {NAME => 'cf1', VERSIONS => 1}, {NAME =>
'cf2', VERSIONS => 1}
```

4. Use ImportTsv tool to load data (in the Linux shell). We will use a one-to-one mapping as this table has a single column primary key.

```
$HBASE_HOME/bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,cf1:plc,cf1:ptk,cf1:ptc,cf1:pn,cf1:bpk,cf1:bpn,cf1:p
cc,cf1:psc,cf1:pbk,cf1:pbc,cf2:pi,cf1:id,cf1:dd -Dimporttsv.skip.bad.lines=false
'sales_product_dim' /hbase_lab/sales_product_dim
```

5. Now, count the rows in the results.

```
hbase(main):021:0> count 'sales_product_dim'
```

```
274 row(s) in 0.4610 seconds
```

6. Run a scan specifying only cf2. This will only look at store files for cf2 column family.

```
hbase(main):002:0> scan  'sales_product_dim', {COLUMNS => 'cf2', LIMIT => 1}
```

```
ROW                                COLUMN+CELL
 30001                             column=cf2:pi, timestamp=1370832114082, value=P01CE1CG1.jpg
1 row(s) in 0.0870 seconds
```

3.6 Query options

HBase APIs provide many options to speed up data access. The most efficient queries in HBase are row key lookups. This makes row key design very important. There are also a variety of filters and other options which reduce the network transfers and provide efficiency where a full table scan cannot be avoided. In this section, we will go over the various query options using hbase shell.

1. **Point query using scan**: If you know the exact row key, specifying it in the scan can return results very quickly. Let us try this with sf_comp_key table that was created earlier and use a row key "*20040112/11101/4001/6737/5185/30126/5501/602"* that we printed out. Specify `STARTROW` and `STOPROW` to use the same value.

```
hbase(main):021:0* scan 'sf_comp_key', {STARTROW =>
'20040112/11101/4001/6737/5185/30126/5501/602', STOPROW =>
'20040112/11101/4001/6737/5185/30126/5501/602'}
```

This returns results in fractions of seconds.

```
ROW                                           COLUMN+CELL
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:cdk, timestamp=1370829366319, value=20040119
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:gm, timestamp=1370829366319, value=0.5118
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:gp, timestamp=1370829366319, value=21472.46
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:q, timestamp=1370829366319, value=587
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:sdk, timestamp=1370829366319, value=20040119
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:sok, timestamp=1370829366319, value=194778
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:st, timestamp=1370829366319, value=41958.76
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:uc, timestamp=1370829366319, value=34.90
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:up, timestamp=1370829366319, value=76.86
 20040112/11101/4001/6737/5185/30126/5501/602    column=cf:usp, timestamp=1370829366319, value=71.48
1 row(s) in 0.0220 seconds
```

2. **Point query using get**: Repeat the same query using get. A get is a special type of scan which gets a row based on the row key.

```
hbase(main):022:0> get 'sf_comp_key', '20040112/11101/4001/6737/5185/30126/5501/602'
```

This query returns the same results formatted differently.

```
COLUMN                                              CELL
 cf:cdk                                             timestamp=1370829366319, value=20040119
 cf:gm                                              timestamp=1370829366319, value=0.5118
 cf:gp                                              timestamp=1370829366319, value=21472.46
 cf:q                                               timestamp=1370829366319, value=587
 cf:sdk                                             timestamp=1370829366319, value=20040119
 cf:sok                                             timestamp=1370829366319, value=194778
 cf:st                                              timestamp=1370829366319, value=41958.76
 cf:uc                                              timestamp=1370829366319, value=34.90
 cf:up                                              timestamp=1370829366319, value=76.86
 cf:usp                                             timestamp=1370829366319, value=71.48
10 row(s) in 0.0110 seconds
```

3. **Range query**: If we need to get all rows in a range, specify the range using start row (inclusive) and stop row (exclusive). The below query returns 3 rows. This is also very quick as HBase uses row key lookup and the range is narrow.

```
hbase(main):023:0> scan 'sf_comp_key', {STARTROW =>
'20040112/11101/4001/6737/5185/30126/5501/602', STOPROW =>
'20040112/11101/4001/6737/5185/30332/9999/999'}
```

4. **Partial range query**: If we have only a part of row key, we can still use range scans. For example, if we had to query all orders for a day (*20040112*), we can specify it as start row and the next value as stop row. This is also handled as a row scan and avoids full table scan.

```
scan 'sf_comp_key', {STARTROW => '20040115', STOPROW => '20040116'}
```

5. **Query using SingleColumnValue filter:** If we have a value for a column, specifying a filter would perform the filtering in the server. It still does a full table scan but only rows that match are returned to the client.

Try a query that returns rows with close date ‘20040117. This should return 8 rows in ~ 6s.

```
scan 'sf_comp_key', {FILTER => "SingleColumnValueFilter ('cf', 'cdk', =,
'binary:20040117', true, false)"}
```

Repeat the query with ‘20040115’. Even when no rows are returned, the query takes a similar time which shows how efficient a point query is compared to filter usage.

```
scan 'sf_comp_key', {FILTER => "SingleColumnValueFilter ('cf', 'cdk', =,
'binary:20040115', true, false)"}
```

`SingleColumnValueFilter` can be used to test column values for equivalence, inequality, or ranges.

6. **Query using ColumnPrefix filters:** HBase provides a column prefix filter which will return only rows with matching column name. Here the filter is on the name and not the value. We will try two queries – one where this can be used efficiently with arbitrary data with varying columns across rows and one where this acts like a column projection in a SQL query.

If we have different set of columns across rows, specifying a column prefix will help in getting rows with certain columns. For the next query, we will use the initial table 'table_1' and try to find the row that has column named 'd11'.

```
scan 'table_1', {FILTER => "ColumnPrefixFilter ('d11') "}
```

This should return 'row2'.

```
ROW                                           COLUMN+CELL
 row2                                         column=column_family1:d11, timestamp=1370827909185, value=n2v11
1 row(s) in 0.0320 seconds
```

If we need to only get values for quantity column for orders that closed on '20040117', we can use `MultipleColumnPrefixFilter` combined with `SingleColumnValueFilter`. We need to use both quantity and close date columns in column prefix list. This is because `SingleColumnValueFilter` requires the close date.

```
scan 'sf_comp_key', {FILTER => "(MultipleColumnPrefixFilter ('cdk', 'q') AND
SingleColumnValueFilter ('cf', 'cdk', =, 'binary:20040117', true, false))"}
```

When a table has same set of columns across rows, specifying a column list works better than use of `MultipleColumnPrefixFilter`. In this case, we can specify a different set of columns from the one in the `SingleColumnValueFilter`.

```
scan 'sf_comp_key', { COLUMNS => ['cf:q'], FILTER => "SingleColumnValueFilter
('cf', 'cdk', =, 'binary:20040117', true, false)"}
```

For a table that has different set of columns across rows, `MultipleColumnPrefixFilter` will perform differently when filtering on a column that exists only in few rows.

7. **Query using row filters:** A row returned by an HBase scan contains the rowkey as part of key value pair for all requested columns. If we need only the row key, the following filters can be used - KeyOnlyFilter, FirstKeyOnlyFilter. KeyOnlyFilter retrieves only the key part for all columns. FirstKeyOnlyFilter retrieves only the first column in a row for all rows. By combining these two filters, we can retrieve just the row key part for all rows. This significantly reduces the amount of data transferred back over the network. When using joins across tables, this can be applied to get just the row key from one table and use it as input to the scan on second table.

Use `KeyOnlyFilter` to understand the results it returns. Adding a `KeyOnlyFilter` to a query with a predicate on a column value will return key value pairs for all columns that match the column predicate. The returned key value pair will not have the value part.

```
scan 'sf_comp_key', { FILTER => "( KeyOnlyFilter() AND SingleColumnValueFilter
('cf', 'cdk', =, 'binary:20040117', true, false))"}
```

Here is a single row from the query results:

```
20040116/11169/4139/7202/5681/30017/5501/606    column=cf:cdk, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:gm, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:gp, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:q, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:sdk, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:sok, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:st, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:uc, timestamp=1370829366319, value=
 20040116/11169/4139/7202/5681/30017/5501/606    column=cf:up, timestamp=1370829366319, value=
20040116/11169/4139/7202/5681/30017/5501/606    column=cf:usp, timestamp=1370829366319, value=
```

Now add a `FirstKeyOnlyFilter` to above query. It returns Use `KeyOnlyFilter` to understand the results it returns. Adding a `KeyOnlyFilter` to a query with a predicate on a column value will return key value pairs for all columns that match the column predicate. The returned key value pair will not have the value part.

```
scan 'sf_comp_key', {FILTER => "( FirstKeyOnlyFilter() AND KeyOnlyFilter() AND
SingleColumnValueFilter ('cf', 'cdk', =, 'binary:20040117', true, false))"}
```

3.7 Hive and HBase

Switch over to a Hive shell (`$HIVE_HOME/bin/hive`). You might want to have two shells; one for HBase one for Hive.

1. In Hive shell:

```
hive> set hive.security.authorization.enabled = false;
```

2. Enter the following statement in Hive shell:

```
CREATE TABLE hive_hbase_table (
   key    int,
   value1 string,
   value2 int,
   value3 int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
 "hbase.columns.mapping" = ":key,f1:c1,f1:c2,f2:c"
)
TBLPROPERTIES (
   "hbase.table.name" = "hive_hbase_table"
); ;
```

3.
```
hive> show tables;
```

4. In HBase shell:

```
Hbase(main)> list
hbase(main):003:0> put 'hive_hbase_table', '1', 'f1:c1', 'stringvalue'
```

```
0 row(s) in 0.1080 seconds
```

```
hbase(main):004:0> scan 'hive_hbase_table'
```

```
ROW                                    COLUMN+CELL
 1                                     column=f1:c1, timestamp=1358288311128,
value=stringvalue
1 row(s) in 0.0380 seconds
```

```
hbase(main):005:0> put 'hive_hbase_table', '1', 'f1:c2', '10'
hbase(main):006:0> put 'hive_hbase_table', '1', 'f2:c', '100'
```

```
hbase(main):007:0> scan 'hive_hbase_table'
```

```
ROW                                      COLUMN+CELL
 1                                       column=f1:c1, timestamp=1358288311128, value=stringvalue
 1                                       column=f1:c2, timestamp=1358288359887, value=10
 1                                       column=f2:c, timestamp=1358288371327, value=100
1 row(s) in 0.0170 seconds
```

5. Switch over to Hive shell

```
hive> select * from hive_hbase_table;
```

```
OK
1       stringvalue     10      100
Time taken: 0.069 seconds
```

6. In hive shell, create another table with complex type map.

```
CREATE TABLE hive_hbase_map (
   key    int,
   value1 string,
   value2 map<string,int>)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
   "hbase.columns.mapping" = ":key,f1:c1,f2:"
);
```

7.

```
hive> show tables;
```

```
OK
hive_hbase_map
hive_hbase_table
Time taken: 0.176 second
```

8. Go to hbase shell:

```
hbase(main):008:0> list
```

```
hbase(main):009:0> put 'hive_hbase_map', '1', 'f1:c1', 'stringvalue'
```

```
hbase(main):010:0> scan 'hive_hbase_map'
```

```
ROW                                  COLUMN+CELL
 1                                   column=f1:c1, timestamp=1358289262766, value=stringvalue
1 row(s) in 0.0160 seconds
```

```
hbase(main):011:0> put 'hive_hbase_map', '1', 'f2:c1', '111'
hbase(main):012:0> put 'hive_hbase_map', '1', 'f2:c2', '112'
hbase(main):013:0> put 'hive_hbase_map', '1', 'f2:c3', '113'
```

```
hbase(main):014:0> scan 'hive_hbase_map'
```

```
ROW                                  COLUMN+CELL
 1                                   column=f1:c1, timestamp=1358289262766, value=stringvalue
 1                                   column=f2:c1, timestamp=1358289610370, value=111
 1                                   column=f2:c2, timestamp=1358289621340, value=112
 1                                   column=f2:c3, timestamp=1358289630006, value=113
1 row(s) in 0.0200 seconds
```

9. Go to hive shell:

```
hive> select * from hive_hbase_map;
```

```
OK
1       stringvalue     {"c1":111,"c2":112,"c3":113}
Time taken: 0.297 seconds
```

4 Summary