HBase: Hadoop’s NoSQL Database and Working with Relational Data with Sqoop

Advanced Analytics with Machine Learning and Hadoop: Week Two Lecture

Last week we explored how to use Hive to perform batch-oriented, streaming analysis and data processing on large but structured data sets stored in HDFS. Hive is a very useful tool if you can ship your data to a Hadoop cluster and perform your analytics processing offline. However, as Hive does not offer real-time queries or row-level updates, it's not appropriate for use cases that require random, real-time read/write access to data, such as sensor data processing, streaming network analysis, or real-time streaming event analysis.

This week, we will learn about Hadoop's NoSQL database, HBase, and create an HBase table to track and manage link-sharing statistics in real-time. We will also introduce Sqoop, a tool that enables us to easily import data from a relational database to Hadoop.

Table of Contents

HBase - Hadoop's NoSQL database

In the previous week, we learned how we could use Hive to perform relational query-based analysis using a SQL-like language on large but structured data-sets stored in HDFS. However, we observed that while Hive provides a new (but familiar!) data manipulation paradigm within Hadoop, it does not change the processing paradigm, which still utilizes MapReduce and HDFS in a high-latency, batch-oriented fashion. Thus, for use cases that require random, real-time read/write access to data, such as Facebook's real-time analytics application "Insights for Websites" platform which tracks over 200,000 events per second[1] we need to look outside of standard MapReduce and Hive for our data persistence and processing layer.

The conventional relational approach also presents a modeling challenge for many data analytics applications. Applications like Facebook's analytics platform, or Stumbleupon's real-time recommendation system[2] are recording heavy volumes of data events from many sources, resulting in a long-tail data pattern comprised of temporal data with many possible structural variations. The data may be keyed on a certain value, like User, but the value is often represented as a collection of arbitrary metadata. Take for example two events, "Like" and "Share", which require different column values:

Event ID	Event Timestamp	Event Type	User ID	Post ID	Comment	Receiver User ID
1	1370139285	Like	jjones	521		
2	1371953685	Share	bsmith	237	This is hilarious!	342
3	1371952265	Share	emiller	963	Great article	

These types of data applications entail sparse data storage requirements. In a relational model, rows are sparse but columns are not. That is, upon inserting a new row to a

[1] Facebook "Building Realtime Insights": https://www.facebook.com/note.php?note_id=10150103900258920

[2] HBase at Stumbleupon: http://www.stumbleupon.com/blog/why-we-love-hbase/

table, the database allocates storage for every column regardless of whether a value exists for that field or not. However in applications where data is represented as a collection of arbitrary fields or sparse columns, each row may use only a subset of available columns, which can make a standard relational schema both a wasteful and awkward fit.

NoSQL and Column-Oriented Databases

NoSQL databases were developed in response to the scale and agility challenges that face many modern applications today. NoSQL is a broad term that generally refers to non-relational databases and encompasses a wide collection of data storage models including graph databases, document databases, key-value data stores and column-family databases.

HBase is a classified as a column-family or column-oriented database, modeled on Google's BigTable architecture[3]. Data is grouped in columns of data rather than as rows of data, and these columns are logically grouped into column families[4].

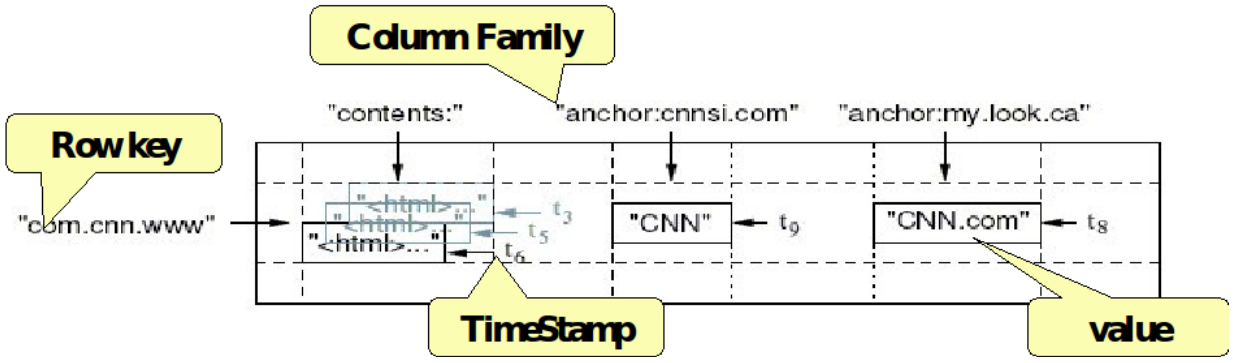

Figure 1 - HBase Column Family Schema[5]

[3] Apache HBase: https://hbase.apache.org/

[4] Bigtable: A Distributed Storage System for Structured Data: http://research.google.com/archive/bigtable.html

[5] Data Science Labs "HBase": http://www.datascience-labs.com/hbase/

Storing data in columns rather than rows has particular benefits for data warehouses and analytical databases where aggregates are computed over large numbers of similar data items with sparse values, where not all columns values are present. Although column families are very flexible, in practice a column family is not entirely schema-less. Each column family must be defined upfront at the time of table definition, but the actual columns that make up a row can be determined at runtime by the client application. Each row can have a different set of columns.

Table rows are sorted by their row key, similar to a primary key, and because row keys are byte arrays, almost anything can serve as a row key from strings to binary representations of longs or even serialized data structures[6]. HBase stores its data as key-value pairs, where all table lookups are performed via the table's row key, or unique identifier to the stored record data.

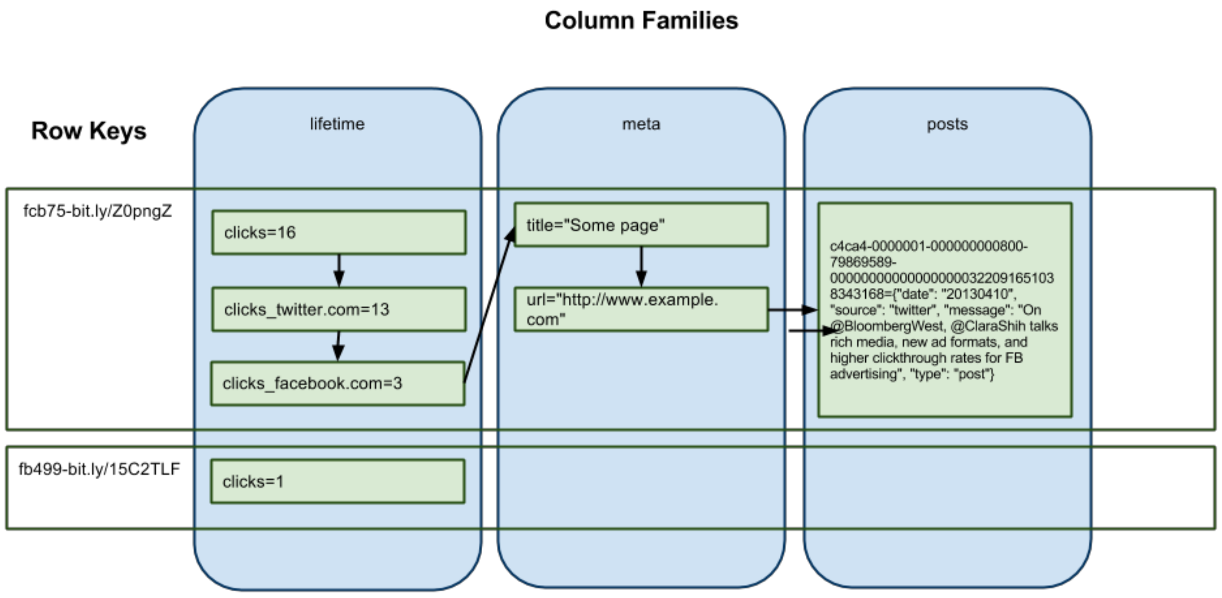
Column Families
Row Keys
lifetime
meta
posts
fcb75-bit.ly/Z0pngZ
clicks=16
clicks_twitter.com=13
clicks_facebook.com=3
title="Some page"
url="http://www.example.com"
c4ca4-0000001-000000000800-79869589-0000000000000000000032209165103 8343168={"date": "20130410", "source": "twitter", "message": "On @BloombergWest, @ClaraShih talks rich media, new ad formats, and higher clickthrough rates for FB advertising", "type": "post"}
fb499-bit.ly/15C2TLF
clicks=1

Figure 2 - Sample HBase Row Keys[7]

[6] Apache HBase Book "Data Model": http://hbase.apache.org/book/datamodel.html

[7] HBase Schema Design: http://chase-seibert.github.io/blog/2013/04/26/hbase-schema-design.html

Another interesting feature of HBase and BigTable-based column-oriented databases is that the table cells, or the intersection of row and column coordinates, are versioned by timestamp, stored as a long integer representing milliseconds since January 1, 1970 UTC. HBase is thus also described as being a multidimensional map where time provides the third dimension. The time dimension is indexed in decreasing order, so that when reading from an HBase store, the most recent values are found first. The contents of a cell can be referenced by a {rowkey, column, timestamp} tuple, or we can scan for a range of cell values by time range.

Architecture

A fully-distributed HBase configuration has one or more master nodes, called the HMaster, and many slave nodes, called RegionServers, which run on HDFS. Each RegionServer in an HBase cluster serves a set of regions, where a region represents a subset of the table's data, and is essentially a contiguously stored, sorted range of rows.

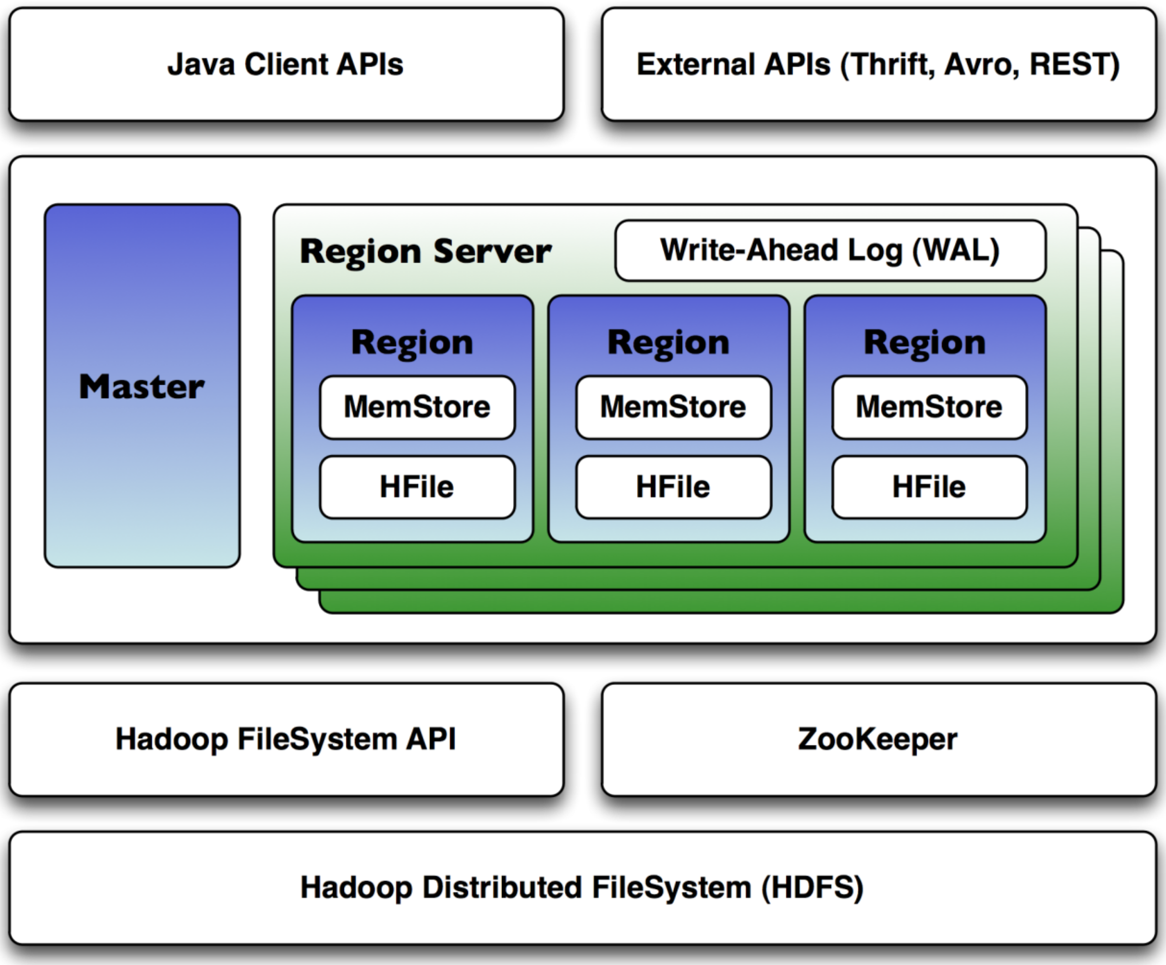

Figure 3 - HBase Architecture[8]

HBase utilizes ZooKeeper, a distributed coordination service, to manage region assignments to RegionServers, and to recover from RegionServer crashes by loading the crashed RegionServer's regions onto other functioning RegionServers.

In a fully-distributed cluster configuration the HMaster, RegionServers, and ZooKeeper Quorum Peer would be deployed on separate server nodes and communicate via well-known ports. In standalone and pseudo-distributed mode, which we will use in this

[8] DZone "Handling Big Data with HBase": http://java.dzone.com/articles/handling-big-data-hbase-part-3

chapter, HBase will run the HMaster and RegionServer daemons as well as a local ZooKeeper within the same JVM.

Setting Up HBase

Installation

HBase requires Hadoop 1.0.3+ and Java 1.6+ (with Oracle Java being recommended), as well as Apache Zookeeper if running in fully-distributed mode. For the purposes of this lesson, we will assume that you are working with Hadoop 1.2.1 under Ubuntu as a "single node cluster", or pseudo-distributed mode, and using the Oracle Java platform with JDK 1.7.

By default HBase runs in standalone mode using the local filesystem for storage. However, we will install HBase in pseudo-distributed mode using a local instance of HDFS. This is obviously not a sufficient setup for a production-ready set up of HBase, but will suffice for learning and testing purposes. For a complete overview of available HBase installations and configuration options, you should refer to the official Apache HBase Documentation, which details the prerequisities and steps for each configuration.

Let's begin by first choosing a download mirror from the list of HBase Apache Download Mirrors and finding the suggested mirror in the top link. Download the latest stable release tarball, which is as of this writing, currently at version 0.94.19. Decompress and untar your download and then change into the unpacked directory.

```
~$ wget http://apache.petsads.us/hbase/stable/hbase-0.94.19.tar.gz
~$ sudo mv hbase-0.94.19.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xvf hbase-0.94.19.tar.gz
/srv$ sudo chown -R analyst:analyst hbase-0.94.19
/srv$ sudo ln -s $(pwd)/hbase-0.94.19 $(pwd)/hbase
```

We will also need to create a local data directory for HBase to maintain ZooKeeper files. We should also update the permissions of the local ZooKeeper data directory so that HBase can write to it:

```
/srv$ mkdir /user/hbase/zookeeper
/srv$ sudo chown analyst:analyst -R /user/hbase/zookeeper
```

Let’s add the following environment variables to our bash profile for convenience:

```
export HBASE_HOME=/srv/hbase
export PATH=$PATH:$HBASE_HOME/bin
```

Within the HBase directory is a /conf directory that includes the configuration files for HBase. Let's edit the config file conf/hbase-site.xml, to configure HBase to run in pseudo-distributed mode with HDFS. To do this, we need to set the hbase.cluster.distributed property to true, and while we're at it let's also set the hbase.rootdir to the local HDFS host:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://localhost:9000/hbase</value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>localhost</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.clientPort</name>
        <value>2181</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/user/hbase/zookeeper</value>
    </property>
</configuration>
```

With this configuration, HBase will start up an HBase Master process, a ZooKeeper server, and a RegionServer process. By default, HBase configures all directories to a /tmp path which means you'll lose all your data whenever your server reboots unless you change it as most operating systems clear /tmp on restart. We thus updated the hbase.zookeeper.property.dataDir to write to the reliable data path we created earlier.

We also need to update our HBase env settings with the JAVA_HOME path, regionservers path and also configure HBase to manage its own instance of ZooKeeper. To do this, uncomment and modify the following settings in conf/hbase-env.sh:

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
export HBASE_REGIONSERVERS=${HBASE_HOME}/conf/regionservers
export HBASE_MANAGES_ZK=true
```

Before we start HBase, we need to ensure Hadoop is running by first attempting to stop all Hadoop processes and then starting them again:

```
/srv/hbase$ $HADOOP_HOME/bin/stop-all.sh
/srv/hbase$ $HADOOP_HOME/bin/start-all.sh
```

Now we can start up HBase!

```
/srv/hbase$ bin/start-hbase.sh
localhost: starting zookeeper, logging to
/srv/hbase/bin/../logs/hbase-analyst-zookeeper-analyst-virtualbox.out
starting master, logging to /srv/hbase/logs/hbase-analyst-master-
analyst-virtualbox.out
localhost: starting regionserver, logging to
/srv/hbase/bin/../logs/hbase-analyst-regionserver-analyst-
virtualbox.out
```

We can verify which processes are running by using the jps command, which should display the running Hadoop processes as well as the HBase HMaster and HRegionServer process:

```
/srv/hbase$ jps
1716 TaskTracker
1184 SecondaryNameNode
4867 Jps
907 DataNode
1340 JobTracker
4774 HRegionServer
656 NameNode
4503 HMaster
```

You can stop HBase at any time with the stop-hbase.sh script:

```
/srv/hbase$ bin/stop-hbase.sh
stopping hbase..................
```

HBase Shell

With HBase started, we can connect to the running instance with the HBase shell:

```
/srv/hbase$ bin/start-hbase.sh
/srv/hbase$ bin/hbase shell
```

You will be presented with a prompt:

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.94.19, r1588997, Tue Mar 22 00:21:01 UTC 2014

hbase(main):001:0>
```

For documentation on the commands that the HBase shell supports, type help followed by a return to get a listing of commands.

```
hbase(main):001:0> help
```

We can also check the status of our HBase cluster by using the status command:

```
hbase(main):002:0> status
1 servers, 0 dead, 2.0000 average load
```

To exit the shell, simply use the exit command.

```
hbase(main):003:0> exit
```

Realtime Analytics with HBase

HBase schemas can be created or updated the HBase Shell or with the Java API, using the HBaseAdmin interface class. Additionally, HBase supports a number of other clients that can be used to support non-Java programming languages, including a REST API interface, Thrift, and Avro[9]. These clients act as proxies that wrap the native Java API.

For the purposes of this HBase overview, we will define and work with the HBase shell to design a schema for a linkshare tracker that tracks the number of times a link has been shared. However, in a real-world setting you would write your application using the native Java API or supported client libraries. If considering an external gateway client, consider your use case carefully. Applications requiring high-throughput may find it advantageous to use a purely binary format like Thrift or Avro, while a REST API may be a better approach for a lower frequency of requests that are large in size.

Generating a Schema

When designing schemas in HBase, it's important to think in terms of the column-family structure of the data model and how it affects data access patterns. While schema definition for traditional relational databases is primarily driven by the accurate representation of the entities, relationships, joins, and indexes, successful HBase schema definition tends to be driven by the intended use cases of the application. Furthermore, since HBase doesn't support joins and provides only a single indexed rowkey, we must be careful to ensure that the schema can fully support all use cases. Often this involves de-normalization and data duplication with nested entities.

The good news is that since HBase allows dynamic column definition at runtime, we have quite a bit of flexibility even after table creation to modify and scale our schema.

[9] Apache HBase Book "External APIs": http://hbase.apache.org/book.html#external_apis

Namespaces, Tables, and Column Families

So what aspects of the schema must we carefully consider upfront? First, we need to declare the table name, and at least one column-family name at the time of table definition. We can also declare our own optional namespace (supported as of Apache HBase v0.96.0) to serve as a logical grouping of tables, analogous to a database in relation database systems[10]. If no namespace is declared, HBase will use the default namespace:

```
hbase> create 'linkshare', 'link'
0 row(s) in 1.5740 seconds
```

We just created a single table called linkshare in the default namespace with one column-family, named link. To alter the table after creation, such as changing or add column families, we need to first disable the table so that clients will not be able to access the table during the alter operation:

```
hbase> disable 'linkshare'
0 row(s) in 1.1340 seconds

hbase> alter 'linkshare', 'statistics'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.1630 seconds
```

We can then re-enable the table using the enable command:

```
hbase> enable 'linkshare'
0 row(s) in 1.1930 seconds
```

[10] Apache HBase Book "Namespace": http://hbase.apache.org/book.html#namespace

And then use the describe command to verify that the table contains the 2 expected column families with the default configurations:

```
hbase> describe 'linkshare'
DESCRIPTION
ENABLED
 'linkshare', {NAME => 'link', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0' true
 , VERSIONS => '3', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =>
'2147483647', KEEP_DELETED_CELLS => 'f
 alse', BLOCKSIZE => '65536', IN_MEMORY => 'false', ENCODE_ON_DISK =>
'true', BLOCKCACHE => 'true'}, {NAME =>
  'statistics', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'NONE',
REPLICATION_SCOPE => '0', VERSIONS => '
 3', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => '2147483647',
KEEP_DELETED_CELLS => 'false', BLOCKSIZ
 E => '65536', IN_MEMORY => 'false', ENCODE_ON_DISK => 'true',
BLOCKCACHE => 'true'}
1 row(s) in 0.0440 seconds
```

Inserting and Reading Data

Row Keys

So far we have created a single HBase table, linkshare, with 2 column families: link and statistics, but our table does not yet contain any rows. Before we insert row data, we need to determine how to design our row key.

Good row key design affects not only how we query the table, but the performance and complexity of data access. By default, HBase stores rows in sorted order by row key, so that similar keys are stored to the same RegionServer. While this enables faster range scans, it could also lead to uneven load on particular servers (called RegionServer hotspotting) during read/write operations. Thus, in addition to enabling our data access use cases, we also need to be mindful to account for row key distribution across regions.

For the current example, let's assume that we will use the unique reversed link URL for the row key. But we highly recommended that you read the Apache HBase Schema Design documentation for case studies on good row key design.

Inserting Data with Put

Now our table is ready to start storing data. In our linkshare application, we want to store descriptive data about the link, such as its title, while maintaining a frequency counter that tracks the number of time the link has been shared.

We can insert, or put, a value in a cell at the specified table/row/column and optionally timestamp coordinates. To put a cell value into table linkshare at row with row key org.hbase.www. under column-family link and column title marked with the current timestamp, do:

```
hbase> put 'linkshare', 'org.hbase.www', 'link:title', 'Apache HBase'
hbase> put 'linkshare', 'org.hadoop.www', 'link:title', 'Apache
Hadoop'
hbase> put 'linkshare', 'com.statistics.www', 'link:title',
'Statistics.com'
```

The put operation works great for inserting a value for a single cell, but for incrementing frequency counters, HBase provides a special mechanism to treat columns as counters. Otherwise under heavy load, we could face significant contention for these rows as we would need to lock the row, read the value, increment it, write it back, and finally unlock the row for other writers to be able to access the cell[11].

To increment a counter, we use the command incr instead of put:

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
COUNTER VALUE = 1
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:like', 1
COUNTER VALUE = 1
```

The last option passed is the increment value, which in this case is 1. Incrementing a counter will return the updated counter value, but you can also access a counter's current value anytime using the get_counter command, specifying the table name, row key and column:

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
COUNTER VALUE = 2
hbase> incr 'linkshare', 'com.statistics.www', 'statistics:like', 30
COUNTER VALUE = 30

hbase> get_counter 'linkshare', 'org.hbase.www', 'statistics:share'
COUNTER VALUE = 2
```

Getting Data

HBase provides two general methods to retrieve data from a table: the get command performs lookups by row key to retrieve attributes for a specific row, and the scan command which takes a set of filter specifications and iterates over multiple rows based on the specified specifications.

[11] George, Lars (2011-08-29). HBase: The Definitive Guide (Kindle Locations 4715-4716). O'Reilly Media. Kindle Edition.

Get Row or Cell Values

In its simplest form, the get command accepts the table name followed by the row key, and returns the most recent version timestamp and cell value for all columns in the row:

```
hbase> get 'linkshare', 'org.hbase.www'
COLUMN                                      CELL
 link:title                                 timestamp=1399886136325,
value=Apache HBase
 statistics:like                            timestamp=1399907428895,
value=\x00\x00\x00\x00\x00\x00\x00\x01
 statistics:share                           timestamp=1399906955305,
value=\x00\x00\x00\x00\x00\x00\x00\x02
3 row(s) in 0.0430 seconds
```

Note that the statistics:share column returns the value in its byte array representation, printing the separate bytes as hexadecimal values. To display the integer representation of the counter value, use the get_counter command mentioned above.

The get command also accepts an optional dictionary of parameters to specify the column(s), timestamp, timerange and version of the cell values we want to retrieve. For example, we can specify the column(s) of interest:

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => 'link:title'}
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => ['link:title',
'statistics:share']}
```

There is also a shortcut to specify column parameters in a get by just appending the comma-delimited column names after the row key:

```
hbase> get 'linkshare', 'org.hbase.www', 'link:title'
hbase> get 'linkshare', 'org.hbase.www', 'link:title',
'statistics:share'
hbase> get 'linkshare', 'org.hbase.www', ['link:title',
'statistics:share']
```

To specify a time range of values we are interested in, we pass in a TIMERANGE parameter with start and end timestamps in milliseconds:

```
hbase> get 'linkshare', 'org.hbase.www', {TIMERANGE => [1399887705673,
1400133976734]}
```

If instead of explicit timestamp ranges we want to pull back cell values based on a certain number of previous versions, we can specify the column of interest and use the VERSIONS parameter to specify the number of versions to retrieve.

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN =>
'statistics:share', VERSIONS => 2}
```

While this type of range query may not seem as interesting for a counter value that increments by 1, it does provide us the means to determine the rate at which the share counter is incremented, which we could use to determine the virality of the link. Additionally, these types of range filters can be especially useful for performing "as-of-time" queries, for example inspecting metrics identified between a certain time range of interest.

Scan Rows

A scan operation is akin to database cursors or iterators, and takes advantage of the underlying sequentially sorted storage mechanism, iterating through row data to match against the scanner specifications. With scan we can scan an entire HBase table or specify a range of rows to scan.

Using scan is similar to using the get command; it also accepts COLUMN, TIMESTAMP, TIMERANGE, and FILTER parameters. However, instead of specifying a single row key, you can specify an optional STARTROW and/or STOPROW parameter which can be used to limit the scan to a specific range of rows. If neither STARTROW nor STOPROW are provided, the scan operation will scan through the entire table.

You can in fact call scan with the table name to display all the contents of a table:

```
hbase> scan 'linkshare'
ROW                                             COLUMN+CELL
 com.statistics.www                             column=statistics:like,
timestamp=1399910604931, value=\x00\x00\x00\x00\x00\x00\x00\x1E
 com.statistics.www                             column=link:title,
timestamp=1399910589286, value=Statistics.com
 org.hadoop.www                                 column=link:title,
timestamp=1399910583124, value=Apache Hadoop
 org.hbase.www                                  column=link:title,
timestamp=1399886136325, value=Apache HBase
 org.hbase.www                                  column=statistics:like,
timestamp=1399907428895, value=\x00\x00\x00\x00\x00\x00\x00\x01
 org.hbase.www                                  column=statistics:share,
timestamp=1399906955305, value=\x00\x00\x00\x00\x00\x00\x00\x03
4 row(s) in 0.0880 seconds
```

Keep in mind that the rows in HBase are stored in lexicographical order[12]. For example, numbers going from 1 to 100 will be ordered like this:

1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,...,9,91,92,93,94,95,96,97,98,99

[12] Apache HBase "Data Model: Rows": http://wiki.apache.org/hadoop/Hbase/DataModel

Let's retrieve the link:title column but limit our scan to the rows starting with row key org.hbase.www:

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'], STARTROW =>
'org.hadoop.www'}
ROW                   COLUMN+CELL
org.hadoop.www        column=link:title, timestamp=1399910583124,
value=Apache Hadoop
org.hbase.www         column=link:title, timestamp=1399886136325,
value=Apache HBase
2 row(s) in 0.0490 seconds
```

But the STARTROW and ENDROW values do not require an exact match for the row key. It will match the first row key that is equal to or larger than the given start row and equal to or less than the end row; because these parameters are inclusive, we do not need to specify the ENDROW if it is the same as the STARTROW value:

```
scan 'linkshare', {COLUMNS => ['link:title'], STARTROW => 'org'}
ROW                   COLUMN+CELL
org.hadoop.www        column=link:title, timestamp=1399910583124,
value=Apache Hadoop
org.hbase.www         column=link:title, timestamp=1399886136325,
value=Apache HBase
2 row(s) in 0.0310 seconds
```

Filters

HBase provides a number of filter classes that can be applied to well, filter the row data returned from a get or scan operation. These filters can provide a much more efficient means of limiting the row data returned by HBase and offloading the row-filtering operations from the client to the server. Some of HBase's available filters include:

- RowFilter: data filtering based on row key values
- ColumnRangeFilter: allows efficient intra row scanning, can be used to efficiently get a 'slice' of the columns of a very wide row (i.e. you have a million columns in a row but you only want to look at columns bbbb-bbdd)
- SingleColumnValueFilter: used to filter cells based on column value
- RegexStringComparator: used to test if a given regular expression matches a cell value in the column

The HBase Java API provides a Filter interface and abstract FilterBase class[13] plus a number of specialized Filter subclasses[14]. Custom filters can also be created by subclassing the FilterBase abstract class and implementing the key abstract methods.

HBase filters are best applied by utilizing the HBase API within a Java program since they generally require importing several dependent filter and comparator classes, but we can demonstrate a simple example of a filter in the shell.

We first need to import the necessary classes, including the org.apache.hadoop.hbase.util.Bytes to convert our column family, column, and values into bytes, and the filter and comparator classes:

```
hbase> import org.apache.hadoop.hbase.util.Bytes
hbase> import org.apache.hadoop.hbase.filter.SingleColumnValueFilter
hbase> import org.apache.hadoop.hbase.filter.BinaryComparator
hbase> import org.apache.hadoop.hbase.filter.CompareFilter
```

---

[13] HBase Filter Class API Doc: https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/Filter.html

[14] Hbase Filter Package: https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/package-summary.html

Now let's create a filter that will limit the results to rows where the statistics:like counter column value is greater than or equal to 10:

```
hbase> likeFilter =
SingleColumnValueFilter.new(Bytes.toBytes('statistics'),
Bytes.toBytes('like'),
CompareFilter::CompareOp.valueOf('GREATER_OR_EQUAL'),
BinaryComparator.new(Bytes.toBytes(10)))
```

And since we don't have a value for this column for every row, we need to set a flag that will tell this filter to skip any rows without a value in this column:

```
hbase> likeFilter.setFilterIfMissing(true)
```

Now we can run our scan operation with the filter we configured:

```
hbase> scan 'linkshare', { FILTER => likeFilter }
```

This should return all rows that contain a column value for statistics:like that is greater than or equal to 10; this should include rowkey 'com.statistics.www' in this example.

Conclusion

HBase is useful when you need to store large volumes of streaming data with a flexible structure, and query that data in small chunks at a time while ensuring that:

- The data is kept "whole" (i.e - sales or financial data)
- The data may change over time
- Single rows and subsets of rows and columns can be queried and updated

HBase isn't intended a one-for-one replacement of an RDBMS, HDFS, or Hive, but does provide the means to leverage Hadoop's data scalability while enabling random access to that data. HBase can then be combined with traditional SQL or Hive to allow snapshots, ranges, or aggregate data to be queried.

In the next section, we will take a look at how we can load pre-existing data from an RDBMS into HBase.

Sqoop - Relational Data and Hadoop

One of Hadoop's greatest strengths is that it's inherently schema-less, and can work with any type or format of data regardless of structure (or lack of structure), from any source as long as you implement Hadoop's Writable or DBWritable interfaces and write your MapReduce code to parse the data correctly. However, in cases where the input data is structured because it resides in a relational database, it would be nice to have a tool that will enable us to import data into Hadoop in a more automated fashion.

Sqoop ("SQL-to-Hadoop") is a relational database import and export tool created by Cloudera[15], and is now an Apache project. Sqoop is designed to transfer data between relational databases like MySQL or Oracle into a Hadoop data sink including HDFS, Hive, and HBase. It automates most of the data transformation process, relying on the RDBMS to provide the schema description for the data to be imported. Sqoop then uses MapReduce to import and export the data to and from Hadoop[16].

Sqoop can be a very useful link in the analytics pipeline of an application stack that relies on a relational database as its primary data store, but uses Hadoop to perform ad-hoc queries against the entire data set without working on the production database. In this chapter we will review a few ways to import data from a MySQL database into Hadoop data sinks, including HDFS, Hive, and HBase.

Importing into HDFS

Installation

Let's start by downloading the latest stable release of Sqoop from the Apache Sqoop Download Mirrors (http://www.apache.org/dyn/closer.cgi/sqoop/1.4.4), which as of this writing is at curently at version 1.4.4. Make sure you grab the version of Sqoop that is compatible with your version of Hadoop (in this example, Hadoop 1.2.1):

[15] Cloudera - Introducing Sqoop: https://blog.cloudera.com/blog/2009/06/introducing-sqoop/

[16] Apache Sqoop User Guide: http://sqoop.apache.org/docs/1.4.4/SqoopUserGuide.html

```
~$ wget http://mirrors.sonic.net/apache/sqoop/1.4.4/sqoop-
1.4.4.bin__hadoop-1.0.0.tar.gz
~$ sudo mv sqoop-1.4.4.bin__hadoop-1.0.0.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xvf sqoop-1.4.4.bin__hadoop-1.0.0.tar.gz
/srv$ sudo chown -R analyst:analyst sqoop-1.4.4.bin__hadoop-1.0.0
/srv$ sudo ln -s $(pwd)/sqoop-1.4.4.bin__hadoop-1.0.0 $(pwd)/sqoop
```

We will also add the following environment variables to our bash profile for convenience:

```
export SQOOP_HOME=/srv/sqoop
export PATH=$PATH:$SQOOP_HOME/bin
```

And now we can verify that Sqoop is successfully running by running sqoop help from $SQOOP_HOME:

```
/srv$ cd $SQOOP_HOME
/srv/sqoop$ sqoop help

usage: sqoop COMMAND [ARGS]
Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
  help               List available commands
  import             Import a table from a database to HDFS
  import-all-tables  Import tables from a database to HDFS
  job                Work with saved jobs
  list-databases     List available databases on a server
  list-tables        List available tables in a database
  merge              Merge results of incremental imports
  metastore          Run a standalone Sqoop metastore
  version            Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

If you see any warnings displayed pertaining to HCatalog, you can safely ignore for now. As you can see, Sqoop provides a list of import and export-specific commands and tools that expect to connect with either a database or Hadoop data-source. The next step, which in real scenarios would be an existing prerequisite, is to set up a source database.

Sample MySQL Import

For this example, we will assume the existence of a MySQL database that resides on the same machine and is accessible via localhost. To install and configure a local MySQL database, please follow the official installation guides on the MySQL site: http://dev.mysql.com/doc/refman/5.5/en/linux-installation.html, or this concise guide on Ubuntu's site: https://help.ubuntu.com/12.04/serverguide/mysql.html.

Once you have MySQL setup, let's setup a sample database with some tables and data:

```
~$ mysql -uroot -p

mysql> CREATE DATABASE energydata;
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON energydata.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> GRANT ALL PRIVILEGES ON energydata.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE average_price_by_state(
    -> year INT NOT NULL,
    -> state VARCHAR(5) NOT NULL,
    -> sector VARCHAR(255),
    -> residential DECIMAL(10,2),
    -> commercial DECIMAL(10,2),
    -> industrial DECIMAL(10,2),
    -> transportation DECIMAL(10,2),
    -> other DECIMAL(10,2),
    -> total DECIMAL(10,2));
Query OK, 0 rows affected (0.02 sec)

mysql> quit;
```

The data that we will load is provided by the U.S. Energy Information Administration[17] and includes the annual data from 1990-2012 on the average energy price per kilowatt hour (KwH) by state and provider type. You can find the CSV named avgprice_kwh_state.csv in the Resources section. Download this CSV and load it into the MySQL table we just created:

```
~$ mysql energydata --local-infile=1

mysql> LOAD DATA LOCAL INFILE 'avgprice_kwh_state.csv' INTO TABLE
average_price_by_state FIELDS TERMINATED BY ',' LINES TERMINATED BY
'\n' IGNORE 1 LINES;
Query OK, 3272 rows affected, 0 warnings (0.04 sec)
```

[17] U.S. Energy Information Administration: http://www.eia.gov/electricity/data/state/

```
Records: 3272  Deleted: 0  Skipped: 0  Warnings: 0

mysql> quit;
```

Before we can import our MySQL table data into HDFS we will need to download the MySQL JDBC driver and add it to Sqoop's lib folder:

```
~$ wget http://dev.mysql.com/get/Downloads/Connector-J/mysql-
connector-java-5.1.30.tar.gz
~$ tar -xvf mysql-connector-java-5.1.30.tar.gz
~$ cd mysql-connector-java-5.1.30
~$ sudo cp mysql-connector-java-5.1.30-bin.jar /srv/sqoop/lib/
~$ cd $SQOOP_HOME
```

We'll also need to ensure Hadoop is running by first attempting to stop all Hadoop processes and then starting them again:

```
/srv/sqoop$ $HADOOP_HOME/bin/stop-all.sh
/srv/sqoop$ $HADOOP_HOME/bin/start-all.sh
```

And now we can import the data in table average_price_by_state into HDFS by using the following command:

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost/energydata -
-table average_price_by_state -m 1

14/05/13 21:32:33 INFO manager.MySQLManager: Preparing to use a MySQL
streaming resultset.
14/05/13 21:32:33 INFO tool.CodeGenTool: Beginning code generation
14/05/13 21:32:34 INFO manager.SqlManager: Executing SQL statement:
SELECT t.* FROM `average_price_by_state` AS t LIMIT 1
```

```
14/05/13 21:32:34 INFO manager.SqlManager: Executing SQL statement:
SELECT t.* FROM `average_price_by_state` AS t LIMIT 1
14/05/13 21:32:34 INFO orm.CompilationManager: HADOOP_MAPRED_HOME is
/srv/hadoop
Note: /tmp/sqoop-
analyst/compile/8ae454fc04f6aa2b84de3c9e8260ffb1/average_price_by_stat
e.java uses or overrides a deprecated API.

..snip..

14/05/13 21:32:57 INFO mapreduce.ImportJobBase: Retrieved 3272
records.
```

Sqoop's import command runs a MapReduce job that connects to the MySQL database, reads the table row-by-row, and writes the results across one or more files within a common directory. We need to specify that the import command should use a single map task since our table does not contain a primary key, which is required to split and merge multiple map tasks. Because we specified that the import task use 1 map task (-m 1), we should expect a single file in HDFS:

```
/srv/sqoop$ hadoop fs -cat average_price_by_state/part-m-00000
```

We have now successfully imported data from MySQL to HDFS! From here we can now run any further MapReduce processing on the imported data, or load the data into another Hadoop data source such as Hive, HBase or HCatalog.

Importing into Hive

Given that our data is already structured in a relational schema (MySQL in this case), it makes a lot of sense to import that data into a similar schema within Hive, especially if we intend to run relational queries on the data. Sqoop provides a couple ways to do this, either exporting to HDFS first and then loading the data into Hive using the LOAD

DATA HQL command in the Hive shell, or by using Sqoop to directly create the tables and load the relational database data into the corresponding tables in Hive.

Sqoop can generate a Hive table and load data based on the defined schema and table contents from a source database, using the import command. However, since Sqoop still actually utilizes MapReduce to implement the data load operation, we must first delete any preexisting data directory with the same output name before running the import tool:

```
/srv/sqoop$ hadoop dfs -rmr /user/analyst/average_price_by_state
```

We can then run Sqoop's import command passing it the JDBC connection string to the database, the table name, field delimiter, line terminator, and null string value:

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost/energydata -
-table average_price_by_state --hive-import --fields-terminated-by ','
--lines-terminated-by '\n' --null-string 'null' -m 1
```

Our double columns will be converted by Hive to float types, and any NOT NULL fields will not be enforced, but otherwise the structure will reflect the initial definition of the MySQL table for average_price_by_state using the same table name.

NOTE: If you are running HBase on the same machine and the HBASE_HOME environment variable is set, you may encounter the following error after running the above command:

```
INFO hive.HiveImport: Exception in thread "main"
java.lang.NoSuchMethodError:
org/apache/thrift/EncodingUtils.setBit(BIZ)B
```

This is due to the conflicting versions of Thrift between HBase and Hive, You can get around this error by temporarily setting HBASE_HOME to a non-existant path and then reloading your bash profile after the import:

```
/srv/sqoop$ export HBASE_HOME=/fake/path
```

(run all sqoop hive commands)

```
/srv/sqoop$ source ~/.profile
```

Recall that Hive will create the local metastore_db directory within the filesystem location from which it was run; in the above example the metastore_db will be created under the SQOOP_HOME (/srv/sqoop). Open the Hive shell and verify that the table average_price_by_state was created.

```
/srv/sqoop$ hive

hive> DESC average_price_by_state;
OK
year                    int                 None
state                   string              None
sector                  string              None
residential             double              None
commercial              double              None
industrial              double              None
transportation          double              None
other                   double              None
total                   double              None
Time taken: 0.59 seconds, Fetched: 9 row(s)
```

You can also run a COUNT query to verify that 3272 rows have been imported, or because this data set is relatively small, you can run a SELECT * FROM average_price_by_state to validate the data.

Importing into HBase

As we learned in the previous chapter, HBase is designed to handle large volumes of data for a large number of concurrent clients that need realtime access to row-level data. While relational databases also handle this requirement well in most low-to-modestly high scale data applications, if the storage requirements of an application start to demand a more scalable solution, we may consider offloading some of the high-scale and heavy-load data components to a distributed database like HBase.

Sqoop's import tool allows us to import data from a relational database to HBase. As with Hive, there are two approaches to importing this data. We can import to HDFS first and then use the HBase CLI or API to load the data into an HBase table, or we can use the --hbase-table option to instruct Sqoop to directly import to a table in HBase[18].

In this example, the data that we want to offload to HBase is a table of weblog stats where each record contains a primary key composed of the pipe-delimited IP address and year, and a column for each month which contains the number of hits for that IP and year. You can find the CSV named weblogs.csv in the Resources section. Download this CSV and load it into a MySQL table:

```
~$ mysql -uroot -p

mysql> CREATE DATABASE logdata;
Query OK, 1 row affected (0.00 sec)
```

[18] Apache Sqoop "Importing Data Into HBase": http://sqoop.apache.org/docs/1.4.4/SqoopUserGuide.html#_importing_data_into_hbase

```
mysql> GRANT ALL PRIVILEGES ON logdata.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON logdata.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE weblogs (ipyear varchar(255) NOT NULL PRIMARY KEY,
  january int(11) DEFAULT NULL,
  february int(11) DEFAULT NULL,
  march int(11) DEFAULT NULL,
  april int(11) DEFAULT NULL,
  may int(11) DEFAULT NULL,
  june int(11) DEFAULT NULL,
  july int(11) DEFAULT NULL,
  august int(11) DEFAULT NULL,
  september int(11) DEFAULT NULL,
  october int(11) DEFAULT NULL,
  november int(11) DEFAULT NULL,
  december int(11) DEFAULT NULL);
Query OK, 0 rows affected (0.02 sec)

mysql> quit;

~$ mysql logdata --local-infile=1

mysql> LOAD DATA LOCAL INFILE 'weblogs.csv' INTO TABLE weblogs FIELDS
TERMINATED BY ',' LINES TERMINATED BY '\n' IGNORE 1 LINES;
Query OK, 27300 rows affected (0.20 sec)
Records: 27300  Deleted: 0  Skipped: 0  Warnings: 0
```

```
mysql> quit;
```

Again, we need to verify that Hadoop is running, as well as HBase daemons:

```
~$ cd $HBASE_HOME
/srv/hbase$ bin/stop-hbase.sh
/srv/hbase$ bin/start-hbase.sh
```

We can then run Sqoop's import command passing it the JDBC connection string to the database, the table name, HBase table name, column family name and rowkey name:

```
bin/sqoop import --connect jdbc:mysql://localhost/logdata --table
weblogs --hbase-table weblogs --column-family traffic --hbase-row-key
ipyear --hbase-create-table -m 1
```

Once the import MapReduce job has completed, you should see a console message indicating INFO mapreduce.ImportJobBase: Retrieved 27300 records. We can verify that the HBase table and rows have been imported successfully in the HBase shell with the list and scan commands:

```
/srv/sqoop$ cd $HBASE_HOME
/srv/hbase$ bin/hbase shell

HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.94.19, r1588997, Tue Apr 22 00:21:01 UTC 2014

hbase(main):001:0> list
TABLE
weblogs
1 row(s) in 1.4170 seconds
```

```
hbase(main):002:0> scan 'weblogs'
```

.. snip ..

We have successfully used Sqoop to import relational data from MySQL to HDFS, Hive, and HBase, using the import tool, which actually generates a Java class that encapsulates one row of the imported table[19]. This class is used during the import process by Sqoop itself, but can also be used in subsequent MapReduce processing of the data. Thus, in addition to automating import/export to and from Hadoop and relational databases, Sqoop allows you to quickly develop MapReduce applications that use the HDFS-stored records in your processing pipeline. We encourage you to read more about Sqoop's features and capabilities on the Apache Sqoop User Guide.

[19] Apache Sqoop "Basic Usage":
http://sqoop.apache.org/docs/1.4.4/SqoopUserGuide.html#_basic_usage

Discussion Questions

1. What key considerations should one consider before migrating from a RDBMS to a NoSQL data store?
2. Given that HBase stores rowkeys in lexicographical order, what possible performance impacts should you keep in mind when designing a rowkey in HBase? Why might prefixing a rowkey with a timestamp be problematic? How could you design your rowkeys to minimize such impacts?
3. Many big data companies including Facebook, Trend Micro, and StumbleUpon integrate Hive and HBase to enable their data analytics platform (http://lanyrd.com/2011/oscon-data/sghmh/). Describe the motivations for combining Hive with HBase and provide example use cases for each in the context of a large, real-time distributed analytics system.

Assignment (Total 15 points)

The Resources section contains a set of CSV data containing one hour of aggregated traffic statistics from Wikipedia[20]. This record in the file contains a count of page views for a specific page on Wikipedia. The first column is the language code, second is the page name, third is the number of page views, and fourth is the size of the page in bytes.

(5 points for successful loading from MySQL to HBase)

Import the pagecounts CSV into a MySQL table using the following schema:

```
CREATE TABLE wikistats (
  language varchar(2) NOT NULL,
  pagename text NOT NULL,
  count int(11) DEFAULT NULL,
```

[20] Wikistats: http://www.mediawiki.org/wiki/Analytics/Wikistats

```
        size int(11) DEFAULT NULL);
```

After loading the data you should have a count of 1,461,294 total rows in the table. Use Sqoop's import tool and the –where argument: http://sqoop.apache.org/docs/1.4.4/SqoopUserGuide.html#_selecting_the_data_to_import to import the rows with language code "en" into an HBase table. If you did this correctly, your import should have retrieved 858,131 records.

(5 points for schema definition)

In your submission, indicate the following attributes of the HBase table (or copy and paste your import command):

1. rowkey
2. column families
    a. columns and any counters

(5 points for HBase commands)

Perform the following HBase commands:

1. Get the HBase row data for the row where pagename = "Hadoop" and provide results
2. Put a new column family, "meta", and column, "url", for the "Hadoop" row and set the value to http://en.wikipedia.org/wiki/Hadoop
3. Add a new row with pagename = "Doge (meme)", language = "en", size = 111, and increment or set the count to 50.

Submit the output of 'describe' on your HBase table and the above commands and result output.