LIGO Event Trigger Database

Steven Dorsher

December 4, 2016

1 Introduction

2 Requirements for the LIGO Event Trigger Database

The requirements for the LIGO Event Trigger Database were described by Gabriela Gonzalez and Duncan Macleod from the LIGO Scientific Collaboration and were summarized in a document by Seung-Jong Park and Kisung Lee from the Division of Computer Science at Louisiana state University. Paraphrased below:

- Hundreds of input time streams will be necessary, corresponding to the various detector channels with up to 16384 samples per second. It is my understanding that these inputs are batched in cycles of about 20 minutes, at the moment, though that may not be the long term goal.
- One channel that needs to be stored is the calibrated GW channel. There are around 1000 auxiliary channels, though far fewer are in the test data I have used.
- The database will catalogue the outputs from various frequency-time based event finders, such as HVeto and Omicron Scans. These searches find "hot" pixels on the frequency-time map based upon some statistic that varies from search to search. How the combination of channels is handled also varies from search to search, but one or multiple channels may be recorded at a given frequency and time.
- Currently these searches use XML files for data and storage, resulting in hundreds of thousands of small files. The bottle neck is writing to and reading from these files. It would be desirable to find a more efficient storage and retrieval solution.

Dr. Park and Dr. Lee estimate that one event is about 100 bytes. They estimate one day is

$$80\text{GB} = 100\text{B/record} * 10\text{records/second} * 1000\text{channels} * 86400\text{seconds/day} \quad (1)$$

They estimate in three months, there will be 7 TB of data.

It is Dr. Park and Dr. Lee's understanding that there will be batch entries to the database every tenth of a minute, with a 100 kB batch size. Duncan notes that current XML databases make entries every 20 minutes, so this would be an expansion upon the current LIGO event trigger storage.

Dr. Lee and Dr. Park specify that there will less than 10 concurrent queries read. Presumably that means there will be fewer than ten queries still resolving in the same clock cycle, though queries probably take on the order of thousands of clock cycles to resolve (or possible much more).

Dr. Lee and Dr. Park note what the LIGO group has requested: the ability to read from and write to the database in higher-level databases. To the LIGO group, this especially means python.

3 Requirements for three simple queries

Based upon knowledge of the existing software used to generate triggers within the LIGO collaboration, Duncan Macleod specified the following three common queries we would like to perform on our database:

- All triggers between time A and time B in channel X, possibly in a given frequency range, possibly above a given SNR threshold,...
- HVeto may be interested in all triggers in channel X close to any trigger in channel Y in time.
- Tertiarily, multiple trigger-generators may be interested in the loudest trigger in all channels close to a specified time.

Other searches may also be interested in other custom queries, or in writing their own queries, as database use expands. This needs to be handled at the HBase Java API level, rather than the external other programming language API level that the LIGO Event Trigger Database will provide.

4 Relational databases

HBase is a non-relational database. The simplest variety of database is a relational database, like XML tables (currently used) or SQL. The tables in these databases can be thought of as just that: tables, like those in an Excel worksheet or in a paper. Data is contained in named columns. The only difference is that each row must be labelled by a unique key. This key may be numeric or may be a string, the only requirement is that it is unique.

In a relational database, if many rows would repeat the same information for several columns, multiple tables can be used to reduce the necessary storage. These tables are linked by foreign keys. The main table is the parent table which stores both its own key and its own data as well as foreign keys to relate its structure to the child table. foreign keys are the keys to the child tables. They are stored as a a column in the parent table, and may repeat many times. By storing foreign keys instead of the entire data from the child table in the parent table, unnecessary storage of repeating information can be avoided. It is possible to have multiple child tables, if the data is not simply related. In our XML files, there are two child tables and one parent table.

5 HBase fundamentals

5.1 Benefits of HBase

Dr. Lee and Dr. Park proposed HBase as a solution. I have relied upon their suggestion. However, I believe HBase is well-motivated.

HBase is a non-relational (NoSQL) database. It is optimized for handling sparse data. In that sense, it cannot be imagined as a table. If a column doesn't have data in it for a specific row, that column doesn't exist for that row and no memory is used. Additionally, columns are organized into column families, which are essentially just categories of columns, based on the type of data and the size of data being stored.

HBase is a database that can be, and usually is, built on top of HDFS (Hadoop Distributed File System). HDFS makes use of a cluster of several computers to store data in a redundant, scalable, manner. It is scalable because the architecture is distributed, in the same sense as a supercomputer. Unlike a supercomputer, the cluster can be large or small and another server (computer) can simply be added to the configuration and HBase will adapt to the additional storage through software that it runs automatically. This software, called Zookeeper, partitions the servers into regions suitable for storage in memory when data is manipulated.

HBase manages the storage of data and the computation of data differently than the canonical supercomputer. The canonical supercomputer, managed using MPI, stores data at a head node (master computer) and sends the data, as needed, to other computers to do computations. In contrast, HBase stores data on its DataNodes and does some simple but common computations there. After doing the basic computations on those servers, it retrieves the partial result and sends it back to the NameNode (like a head node in a supercomputer), which combines and completes the computations. The difference is where the data is stored– at the origin of the computation (head node, in the MPI use of a supercomputers) or at the place where the parallel processing takes place (DataNodes, in HBase). The API for this process in HBase is handled through

Hadoop's MapReduce or HBase's Coprocessors. Zookeeper is the process that runs in the background that manages splitting data into regions on the servers and allocating them to specific computation threads. In the architecture of HBase, Zookeeper replaces the need for MPI or OpenMP in a supercomputer, though it does more than that by also managing the database regions within the computer to keep them within the size usable by the computer's memory.

There is three-fold redundancy built into HBase, by default. Three copies of each piece of data is stored, each on a separate DataNode. This is important to keep the cluster online in case one DataNode fails.

If the NameNode fails, recent writes to the database are stored in the Write Ahead Log (WAL) on the Secondary NameNode, so transactions in process may be recovered once the Secondary NameNode comes online. This is similar to the data stored in one of the XML child tables, only it is handled automatically with HBase through the WAL.

Another benefit of HBase is that it provides compact storage and quick retrieval of data along certain dimensions. All values in the database are stored in binary, allowing compact storage. Keys must be unique, so in a LIGO Event trigger database, we would need a combination of detector, search, channel, time, the nanoseconds part of time, and frequency. Depending on the order of these channels in the key, it could be faster to search on different channels. It is faster to search on a key value than a column value because it doesn't require a full database scan. It is faster to search on the first channel in the key than the last for two reasons: there is a PrefixFilter that specifically allows a first portion of a key to be selected and when searched using a regular expression filter (pattern matcher for strings) it finds the first part of the compound key first.

HBase has built in revision control. It stores, by default, the last three versions of the data that have been written to the database and the time at which the data was written to the database. Don't confuse the HBase timestamp with the LIGO timestamp! This storage of recent history is important because updates and reads are asynchronous. They do not happen all at the same time. Therefore, unless one accesses records that are a bit older than the data currently being written, one may get some strange results. HBase also stores a history of operations on the database in the Write Ahead Log (WAL).

To recap: the benefits of HBase are that it is scalable to larger cluster sizes by simply adding a computer, it is optimized for sparse data (where only certain channels may be represented in a given row), it stores a revision history automatically, it has a mechanism for remaining online at all times (the Secondary Name Node), it has a mechanism for duplication of data so that it is never lost, it stores data compactly in binary, and quick retrieval for certain queries is possible depending upon the key chosen.

5.2 Other things to know before beginning

There are two primary ways to write code in HBase: at the shell, or using the Java API. It is also possible to use other languages such as Ruby, though it will be harder to find documentation for these methods. It is evidently possible to script in the HBase shell, but if we want to interface to external networks, read directly from XML files, or make use of Hadoop's MapReduce, it is probably better to use the Java API because it is a more general programming language.

There are three modes in which HBase can be run. The most complex is fully distributed mode. This involves having a cluster of at least five computers: a NameNode, three Data Nodes (Region Servers) for a redundancy factor of three, and a Secondary NameNode in case the NameNode fails to keep the cluster online at all times. This is the mode that would need to be used for a production cluster, since it has a full filesystem with parallelism, redundancy, and scalability.

The second mode mimics the fully distributed mode, but using only one computer. It is called pseudo-distributed, and could be used for second stage testing of the filesystem, MapReduce, and AggregationClients and Coprocessors prior to buying expensive hardware.

The least complicated mode does not make use of the HDFS filesystem at all but allows the user to test HBase commands using a Java interpreter. That mode is called stand-alone mode, and that is the mode I have been using to develop the preliminary database reads and writes. However, this mode cannot be used for MapReduce or AggregationClients, so soon it would be necessary to shift to the fully distributed mode.

HBase is currently at the 1.2.3 version. I am using HBase 1.2.1 with the documentation for HBase 1.2.2. When HBase 2.0 is released, several crucial deprecated methods and objects will go away. I have been updating some of the examples I found online to use methods that will not go away in HBase 2.0.

5.3 HBase structure

In HBase, if the database grows too large, it is partitioned into regions of consecutive rows by Zookeeper. This is handled dynamically as rows are added. Within each row there are a number of column families. Column families are like categories for columns, that the programmer can select at the time the database is created. There can only be a small number of column families. For instance, column families for an Event Trigger Database might include “times” and “statistics”. Within each column family there can be an arbitrarily large number of columns. Columns could be, for example, “snr” or “confidence”. HBase is a column-oriented architecture, which means that it stores data on the servers ordered along the columns within each region rather than along the rows. If data does not exist for a given row, column family, and column, it simply is not stored.

5.4 Hadoop MapReduce

As a reminder, one proposal for the row key was detector, search, channel, time, time ns, frequency.

HBase runs on top of Hadoop, which provides a function known as MapReduce. MapReduce makes use of the parallel computation power of the cluster when more than a simple database query is needed. The Mapper takes as input a set of key, value pairs such as the row keys and their corresponding chisq column. A specifically written mapper could produce a (chisq, 1) or (chisq,0) pair of key, value for each chisq depending whether it is in the right detector and right search. The Reducer could create a histogram by counting the 1’s between certain boundary values of chisq. This would allow comparison of the chisq distribution to the expected chisq distribution. This is not a particularly good use of MapReduce, because it puts the bulk of the work on the Reducer, and a reduction, just like in supercomputing, is time consuming compared to a mapping because it must be done by a single computer combining inputs rather than by many computers in parallel. In Hadoop, Combiners can be used to mitigate this, in some cases, by distributing partial results over several computers. In this case, one might think that each histogram bin could be handled by a different computer, but that wouldn’t help, because each computer would still need to sort through all of the data to find which data points are in the bin.

As a second example, the input to the Mapper could be the (row key, 1). The Mapper would process the row key, which contains both time and frequency information, to produce a (time, frequency) pair for all times and frequencies. Note that with this row key, it is not possible to obtain this information from a simple database query without using MapReduce unless time and frequency are also stored as columns! In this case, it would not be necessary to use a Reducer at all, and indeed it is possible to use MapReduce with either a Mapper or a Reducer alone.

6 LIGO database design

The database designed by Seung-Jong Park and Kisung Lee proposes Apache HBase as a candidate distributed NoSQL system because it is open source, it replicates data and is therefore fault tolerant, and it would be quick to scan for timestamps if they are used as keys. However, they note that it has no support for secondary indices, making it slow to find records of whatever the secondary part of the key is.

They note that that it requires a distributed file system, such as HDFS.

They suggest a cluster, starting with five commodity servers. They suggest each server should have three 2 TB HDDs, and the more HDDs the better.

They propose three alternative designs, that must be modified somewhat in light of the requirement for a unique key. Their design is given below.

6.1 Key: timestamp, channel id

Columns: channel id, central frequency, bandwidth, ...

The advantage is that it would be fast to find all the records for a given time range; however, it would be slower to find the records of a specific channel, and even slower to find the records based on another column.

6.2 Key: channel id, timestamp

Columns: channel id, central frequency, bandwidth, ...

In this case it would be fast to find the records of a specific channel and a given time range, using range scans. However, one would need to use separate range scans for every channel. Dr. Park and Dr. Lee note that since records are distributed, this would not be a big problem. I agree that this may be the preferred option, provided a solution can be found to make the key unique.

6.3 Key: timestamp only

Columns: records of all the channels. This would involve storing a large number of sparsely populated columns. This would not be a problem for HBase.

The advantage would be that it would have fewer rows in the database, and thus would be fast to read all the records of a specific timestamp. However, it would be slow to choose all the records of a specific channel, since it would require a full scan. Dr. Park and Dr. Lee note that this method is only better than the previous two methods if there are many entries sharing the same timestamp.

It would be necessary to modify this approach because HBase requires a unique key. A proposal for modification is given in the next subsection.

7 Ultimate proposal key: detector, search, channel id, time, time in ns, frequency

Two proposals for channels:

1. detector, search, channel id, time, time in ns, frequency, SNR, ...
2. detector, search, all channels, time, time in ns, frequency, SNR, ...

Duncan and I propose this structure because it would allow for a unique key, necessary in HBase. The specific order of these components in the key has not been fixed definitively. We would duplicate the data in the columns of the database to avoid unnecessary parsing of the key as a string once the row has been retrieved. In proposal one for the channels stored, we would store only the channel id. In the second, which would be better if some triggers could have multiple channels, we would need to store all the channels and the data would be sparse.

8 Simple query design

8.1 All triggers between time A and time B in channel X, possibly in a given frequency range, possibly above a given SNR threshold,...

This query needs to:

- Select rows with times above a minimum
- Select rows with times below a maximum
- Select rows with times with a certain channel id or in a certain channel column

- Select rows with frequencies above a minimum frequency
- Select rows with frequencies below a maximum frequency
- Select rows with SNR's above a minimum SNR

All this can be done with a single Filter<List>and Get combination. A simple example of this query has been implemented, though the general form has not.

8.2 HVeto: all triggers in channel X close to any trigger in channel Y in time

For this query, all triggers in channel X must be selected. This either requires a Get for a column storing channel X, or a Get for a column storing channel id with X as input. All triggers in channel Y will also need to be selected. It is possible that MapReduce could be used in combination with an HBase query on channel Y to do the comparison in parallel, distributed over the X data, but this will need to be investigated further.

8.3 The loudest trigger in all channels close to a specified time

This query involves finding the maximum value of a channel (SNR). There is no simple way to do this in HBase, in a distributed manner. Maximums are implemented in the AggregationClient object, which requires the use of Coprocessors. Coprocessors require the programmer to manage Zookeeper explicitly, which requires advanced system administration skills. The system needs to be reconfigured into pseudo-distributed mode (simulating a full cluster) or distributed mode (a full cluster) in order to make use of the distributed nature of Zookeeper, Coprocessors, and the AggregationClient. Reconfiguring in pseudo-distributed mode will require removing and restarting the database currently running in my computer, though the code written so far will remain usable.

9 Implementation progress

9.1 Toy CRUD database updated for HBase 1.x

I began by updating a toy database that performs the Create, Read, Update, and Delete (CRUD) operations that any standard database should perform. Much of the HBase documentation available was for deprecated table creation and handling methods (HBaseAdmin and HTableDescriptor). The toy database is based on examples found at `https://www.tutorialspoint.com/hbase`. It has been updated for methods that will be remain available in HBase 2.0 when it is released, and is available at `https://github.com/sdorsher/toyCRUDdatabase`. Also see Appendix A. It creates a table, inserts a row named "row1" with column family "personal", column "name", and value "raju". Likewise it it inserts some data for column families "personal" and "professional". It retrieves the data and converts it to a string for a couple of entries in the table, then prints it to the command line. In ScanTable, binary output from a full scan of the entire table is printed to the command line. DeleteData deletes a specific item from the table. DropTable disables the table itself and removes it from HBase. All of this has been verified by looking at the data the HBase shell at various stages in this process.

9.2 Structure of the temporary LIGO database

To begin exploring HBase, I created a temporary LIGO database with a small sample of data. It has a numerically increasing key, prepended by a search id (i.e. sngl_burst:event_id:4696). The table has four column families: channel, time, frquency, and statistics. The specific search (sngl_burst) will be used as a filter to ensure that all data in the database, at this time, is from only one search. Later the database will be expanded to more searches. This feature is not yet implemented.

The channel column family contains the ifo (detector) column and the channel (channel id) column. The time column family contains peak_time, peak_time_ns, start_time, start_time_ns, and duration. The frequency column family contains central_freq and bandwidth. The statistics column family contains the snr, confidence, chisq, and chisq_dof columns.

This structure is flexible, and was set based on a loosely logical basis based on the meaning of the data. It could be restructured based on the needs of the LSC or based on architectural needs of the database.

The code for the temporary LIGO database is available at `https://github.com/sdorsher/LIGOdatabase`. Also see Appendix B. The data file used to populate the database was H1-LDAS_STRAIN-968654552-10.xml. I used the command

```
ligolw_print -t sngl_burst H1-LDAS_STRAIN-968654552-10.xml
-c search -c event_id -c ifo -c channel -c peak_time
-c peak_time_ns -c central_freq -c start_time -c start_time_ns
-c duration -c amplitude -c snr -c confidence -c chisq
-c chisq_dof -c bandwidth > event.csv
```

to generate the file event.csv used as input to the LIGO HBase code. CreateTable.java contains the code to generate the structure for this table, and ReadFileandInsert.java reads the XML file and inserts the data into the table.

9.3 Retrieval mechanisms for the temporary LIGO database

Data can be retrieved from the temporary LIGO database by directly getting it from a given row, column family, and column, as is done in GetData.java. GetData.java retrieves all cells in one row in one column family, as well as all cells in the same row in another column family, for only one column. It then converts the value of the cell to strings and prints them to standard out.

It is also possible to retrieve cell values by using filters to specify a more complicated query. In Filter.java and FilterString.java, rows are selected based on whether or not a certain column matches a specified value. Then some columns from that row are output, which may or may not be the same as the column as used in the filter. It is possible to use a binary type (Filter.java) or a string type (FilterString.java) as the input to the filter, although the data in the database is always stored in binary form. Filter.java prints all SNR's greater than 1.0 to output. FilterString.java prints all channel ids that begin with the string "STRAIN" to output.

In FilterListClass.java, I have combined filters in a filter list to execute several compatible queries at once. I select all SNR greater than or equal to 0.97 from rows that have channel id's that start with "STRAIN" and that have row keys less than or equal to 5000 and greater than or equal to 4800. I find 11 triggers that meet this criterion.

In a preliminary attempt at programming "All triggers in channel X close to all triggers in channel Y", the design goal of Section , I attempted to select the output of one column based on the output of another filter in SecondScanner.java. I use the four part filter from FilterListClass.java and attempt to select only those rows from the output that have confidence values less than 25.3. However, it has two problems, one more fundamental than the other. The simple, technical detail is that conversion between binary and double format is not working correctly, resulting in the wrong output. The more fundamental problem of design is that all of the computation comparing X to Y is done on the NameNode, which undermines the benefit of the distributed nature of HBase.

10 Unresolved problems in the LIGO Event Trigger Database

There are several unresolved problems in the LIGO Event Trigger Database, ranging from straightforward tasks to serious design and system administration issues involving both HBase and Hadoop's MapReduce.

The first step toward implementing a full database would be to implement the full detector, search, channel, time, time in ns, frequency key. It will be necessary to do some kind of regular expression search over this key to be able to retrieve based on each of these values, or the most important ones. The specific structure of the ordering of this key will need to be decided based upon which values are most important to retrieve fastest. It will also be important to zero-pad any channel that has a variable number of digits in it, to make sure that it sorts properly when sorted in binary. Binary sorts by first digit rather than numeric order.

The XML Event Trigger Database currently has two child tables linked to the parent table by foreign keys. It will be necessary to form an HBase implementation of this structure, though the keys may not look exactly the same. For instance, the XML foreign keys are essentially numeric; however, an HBase foreign key could be build by compounding columns from the database, much like the row key to the parent table. It will be necessary to both design and implement the foreign keys and child tables depending on the needs of the specific searches. It is not clear at this time that it is desirable to store the same metadata, in the same manner, as in the current database.

Eventually, it will become necessary to reconfigure to pseudo-distributed mode for further testing of anything dependent upon Coprocessors, MapReduce, HDFS, or Zookeeper. This will allow simulation of the full cluster on one machine for the purposes of code development. Someone with intermediate system administration skills will be required for these purposes, but with advanced skills compared to many physicists.

The queries listed in Section 8 will need to be generalized or developed. In particular, it will be necessary to determine how best to perform the query specified in Section 9.3, looping over the results of a first query (all triggers in channel X near to to all triggers in channel Y in time). This may involve Hive or Hadoop's MapReduce. See the section on other software below for an explanation of Hive. It is possible to use the output of an HBase database query as input to a Hadoop MapReduce call, through the Java API; however, the specifics are not yet tested.

To perform the query specified in Section 8.3, (strongest signal in channel Y), maxima will need to be implemented. This requires the use of Coprocessors, AggregationClients, and Zookeeper. It can only be done in pseudo-distributed or fully distributed mode.

Duncan Macleod and I have decided that it would be desirable to solve the problem of interfacing with the LIGO Event Trigger Database using other languages through using an HTTP interface. This would allow a portable interface that would not be language specific. Furthermore, it would allow remote access to the database. However, allowing remote access to the database would probably ultimately require some kind of authentication (using Kerberos credentials?) and would require the programmer to understand Linux security administration.

It will also be necessary to convert existing data from XML to HBase binary storage format. Hadoop has the StreamXmlRecordReader, that may or may not be usable in the Java API with HBase. This would require the data to be streamed into the database, rather than read from a static file in a batch.

Eventually it would become desirable to set up a fully distributed cluster of at least five computers, as described in Dr. Park and Dr. Lee's design. This could be used for a fully developed HBase system, when it is sufficiently developed that the full LIGO data has been loaded onto it, it is reading new data, and searches are querying it for use in the code they are developing.

10.1 Other software

Hive is a software package that can be run on top of Hadoop or HBase to make its interface more like SQL. This is a simple interface, but it appears that it takes away some of the flexibility HBase allows in using MapReduce. It is not clear whether or not the LIGO Event Trigger database needs this flexibility. It depends whether we want to use the database to do complex queries like histograms in a distributed manner or whether we want to do that at the end user's side, or using a supercomputer. Hive might be a better approach if the three queries specified above are truly our main use cases, since it is more straightforward in Hive to make a query using the output of a previous query (case 2) or to generate a maximum (case 3). Maxima are built in functions that can simply be called and that are managed automatically, which would make Hive much nicer than HBase, assuming that Hive does not lose flexibility. In principle, Hive should retain the scalability of Hadoop and HBase and many of the benefits of these distributed, data-centered environments.

Maven is a package managing software that many people use to manage large software projects in Hadoop, HBase, or Java. It seems to be somewhat complex to learn how to create a Maven project, or even to run one from the command line. However, it would be ideal to be able to do this either from the command line or from an Integrated Development Environment (IDE), since it will make managing a large project much simpler.

Eclipse is an IDE that Dr. Lee and Dr. Park's graduate student Sayan Goswami suggested. Duncan and I found it simpler to work from the command line, but in a software development environment in industry, it is likely that they would use an IDE to make it easier to interface with other software packages, to make it easier to debug, to automate completion of variables and methods to avoid typos, and to integrate with revision control systems such as git.

See Appendix A for the toy CRUD database code listing. See Appendix B for the LIGO Event Trigger Database code listing. See Appendix C For directions on installing Hadoop and HBase and running these at the command line using the Java API in stand-alone mode. There are also instructions there for running the shell.

11 References

References

[1] TutorialsPoint. *Learn HBase: simply easy learning.* `https://www.tutorialspoint.com/hbase`

[2] Apache HBase Team. *Apache HBase Reference Guide*, Version 1.2.2. ApacheHBaseBook1.2.2.tar on Steven Dorsher's Google Drive.

[3] Lars George. *HBase: The Definitive Guide*, 2nd ed, early release. O'Reilly. `http://tinyurl.com/znfpc37`

[4] K Hong. *Hadoop 2.6 Installing on Ubuntu 14.04 (Single-Node Cluster).* `http://tinyurl.com/hrcrtzy`

[5] TutorialsPoint. *HBase – Installation.* `http://www.tutorialspoint.com/hbase/hbase_installation.htm`

[6] Cloudera. *Installing the ZooKeeper Packages.* `http://tinyurl.com/hoso5y4`

[7] Steven Dorsher. *toyCRUDdatabase GitHub Repository.* `https://github.com/sdorsher/toyCRUDdatabase`

[8] Steven Dorsher. *LIGOdatabase GitHub Repository.* `https://github.com/sdorsher/LIGOdatabase`

12 Appendix A: Toy CRUD Code

12.1 CreateTable.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;

public class CreateTable {

    public static void main(String[] args) throws IOException {
        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Admin admin = connection.getAdmin();

        TableName tableName = TableName.valueOf("emp");

        // Instantiating table descriptor class
        HTableDescriptor  tableDescriptor = new
          HTableDescriptor(tableName);

      // Adding column families to table descriptor
      tableDescriptor.addFamily(new HColumnDescriptor("personal"));
      tableDescriptor.addFamily(new HColumnDescriptor("professional"));

      // Execute the table through admin
      admin.createTable(tableDescriptor);
      System.out.println(" Table created ");
    }
}
```

12.2 InsertData.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.HTableDescriptor;

public class InsertData{

    public static void main(String[] args) throws IOException {

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.valueOf("emp"));

        // Instantiating Put class
        // accepts a row name.
        Put p = new Put(Bytes.toBytes("row1"));

        // adding values using add() method
        // accepts column family name, qualifier/row name ,value
        p.add(Bytes.toBytes("personal"),
              Bytes.toBytes("name"),Bytes.toBytes("raju"));

        p.add(Bytes.toBytes("personal"),
              Bytes.toBytes("city"),Bytes.toBytes("hyderabad"));

        p.add(Bytes.toBytes("professional"),Bytes.toBytes("designation"),
              Bytes.toBytes("manager"));

        p.add(Bytes.toBytes("professional"),Bytes.toBytes("salary"),
              Bytes.toBytes("50000"));

        // Saving the put Instance to the HTable.
        table.put(p);
        System.out.println("data inserted");

        // closing HTable
    }
}
```

12.3 RetriveData.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

public class RetriveData{

    public static void main(String[] args) throws IOException, Exception{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.valueOf("emp"));


        // Instantiating Get class
        Get g = new Get(Bytes.toBytes("row1"));
        // Reading the data
        Result result = table.get(g);

        // Reading values from Result class object
        byte [] value = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

        byte [] value1 = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

        // Printing the values
        String name = Bytes.toString(value);
        String city = Bytes.toString(value1);

        System.out.println("name: " + name + " city: " + city);
    }
}
```

12.4 ScanTable.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;


public class ScanTable{

    public static void main(String args[]) throws IOException{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.valueOf("emp"));

        // Instantiating the Scan class
        Scan scan = new Scan();

        // Scanning the required columns
        scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));
        scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("city"));
        // Getting the scan result
        ResultScanner scanner = table.getScanner(scan);

        // Reading values from scan result
        for (Result result = scanner.next(); result != null; result = scanner.next())

            System.out.println("Found row : " + result);
        //closing the scanner
        scanner.close();
    }
}
```

12.5 DeleteData.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.util.Bytes;

public class DeleteData {

    public static void main(String[] args) throws IOException {

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.valueOf("emp"));

        // Instantiating Delete class
        Delete delete = new Delete(Bytes.toBytes("row1"));
        delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));
        delete.deleteFamily(Bytes.toBytes("professional"));

        // deleting the data
        table.delete(delete);

        // closing the HTable object
        table.close();
        System.out.println("data deleted.....");
    }
}
```

12.6 DropTable.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.util.Bytes;

public class DropTable {

        public static void main(String[] args) throws IOException {
            Configuration config= HBaseConfiguration.create();

            Connection connection = ConnectionFactory.createConnection(config);
            Table table= connection.getTable(TableName.valueOf("emp"));
            Admin admin = connection.getAdmin();

            TableName tableName = TableName.valueOf("emp");
            admin.disableTable(tableName);
            admin.deleteTable(tableName);
            System.out.println("Table Deleted");
        }
}
```

13 Appendix B: LIGO Event Trigger Database Code

13.1 CreateTable.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;

public class CreateTable {

    public static void main(String[] args) throws IOException {
        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Admin admin = connection.getAdmin();

        TableName tableName = TableName.valueOf("gstlal_excesspower");

        // Instantiating table descriptor class
        HTableDescriptor  tableDescriptor = new
            HTableDescriptor(tableName);

        // Adding column families to table descriptor
        //sort on search
        tableDescriptor.addFamily(new HColumnDescriptor("channel"));
        tableDescriptor.addFamily(new HColumnDescriptor("time"));
        tableDescriptor.addFamily(new HColumnDescriptor("frequency"));
        tableDescriptor.addFamily(new HColumnDescriptor("statistics"));
        //other is amplitude and bandwidth

        // Execute the table through admin
        admin.createTable(tableDescriptor);
        System.out.println(" Table created ");
    }
}
```

13.2 ReadFileAndInsert.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.HTableDescriptor;


public class ReadFileAndInsert {

    public static void main(String[] args) {
        String csvFile = "/home/hduser/LIGOdatabase/event.csv";
        BufferedReader br = null;
        String line = "";
        String csvSplitBy = ",";

        String[] columns = {"search", "event_index", "ifo", "channel",
                          "peak_time", "peak_time_ns", "central_freq",
                          "start_time", "start_time_ns", "duration",
                          "amplitude", "snr", "confidence", "chisq",
                          "chisq_dof", "bandwidth"};

        String[] columnFamilies = {"filter", "row_key", "channel", "channel",
                                   "time", "time", "frequency", "time",
                                   "time", "time", "statistics", "statistics",
                                   "statistics", "statistics", "statistics",
                                   "frequency"};


        String filterValue = "gstlal_excesspower";
        int filterNumber = 0;
        int keyNumber = 1;


        try{

            Configuration config = HBaseConfiguration.create();
            Connection connection = ConnectionFactory.createConnection(config);
            Table table = connection.getTable(TableName.
                                          valueOf("gstlal_excesspower"));



            br = new BufferedReader(new FileReader(csvFile));
            while((line = br.readLine()) !=null)
                {
                    System.out.println(line);
                    String[] event = line.split(csvSplitBy, -1);
                    // -1 includes empty strings
                    if(event.length!=columns.length){
                        System.out.println("columns length error");
                    }
                    if(event.length!=columnFamilies.length){
                        System.out.println("columnFamilies length error");
                    }
                    if(event[filterNumber].equals(filterValue)){
                        Put p = new Put(Bytes.toBytes(event[1]));
                        for(int i=keyNumber+1; i<event.length; i++){
                            p.add(Bytes.toBytes(columnFamilies[i]),
                                  Bytes.toBytes(columns[i]),
                                  Bytes.toBytes(event[i]));
                        }
                        table.put(p);
                        System.out.println(p);
                        System.out.println("PUT");
                    }
                }
        }catch (FileNotFoundException e){
            e.printStackTrace();
        }catch (IOException e) {
            e.printStackTrace();
        }finally{

            if(br!=null){
                try{
                    br.close();
                } catch(IOException e){
                    e.printStackTrace();
                }
            }
        }
    }
}
```

13.3 GetData.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

public class GetData{

    public static void main(String[] args) throws IOException, Exception{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.valueOf(
                                   "gstlal_excesspower"));


        String[] columns = {"search", "event_index", "ifo", "channel",
                            "peak_time", "peak_time_ns", "central_freq",
                            "start_time", "start_time_ns", "duration",
                            "amplitude", "snr", "confidence", "chisq",
                            "chisq_dof", "bandwidth"};

        String[] columnFamilies = {"filter", "row_key", "channel", "channel",
                                   "time", "time", "frequency", "time",
                                   "time", "time", "statistics",
                                "statistics", "statistics", "statistics",
                                "statistics", "frequency"};

        // Instantiating Get class
        Get g = new Get(Bytes.toBytes("sngl_burst:event_id:4701"));
        g.addFamily(Bytes.toBytes("statistics"));
        g.addColumn(Bytes.toBytes("time"),Bytes.toBytes("peak_time"));

        // Reading the data
        Result result = table.get(g);

        // Reading values from Result class object
        byte [] value = result.getValue(Bytes.toBytes("statistics"),
                                        Bytes.toBytes("amplitude"));
        byte[] value1 = result.getValue(Bytes.toBytes("statistics"),
                                        Bytes.toBytes("snr"));
        byte[] value2 = result.getValue(Bytes.toBytes("statistics"),
                                        Bytes.toBytes("confidence"));
        byte[] value3 = result.getValue(Bytes.toBytes("statistics"),
                                        Bytes.toBytes("chisq"));
        byte[] value4 = result.getValue(Bytes.toBytes("statistics"),
                                        Bytes.toBytes("chisq_dof"));
        byte [] value5 = result.getValue(Bytes.toBytes("time"),
                                         Bytes.toBytes("peak_time"));

        // Printing the values
        String amplitude = Bytes.toString(value);
        String snr = Bytes.toString(value1);
        String confidence = Bytes.toString(value2);
        String chisq = Bytes.toString(value3);
        String chisq_dof = Bytes.toString(value4);
        String peak_time = Bytes.toString(value5);


        System.out.println("amp: " + amplitude + " snr: " + snr +
                           " conf " + confidence + " chisq " +
                           chisq + " dof " + chisq_dof+ " t " +
                           peak_time );
    }
}
```

13.4 Filter.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import java.util.List;


public class Filter{

    public static void main(String[] args) throws IOException, Exception{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.
                                    valueOf("gstlal_excesspower"));


        String[] columns = {"search", "event_index", "ifo", "channel",
                            "peak_time", "peak_time_ns", "central_freq",
                            "start_time", "start_time_ns", "duration",
                            "amplitude", "snr", "confidence", "chisq",
                            "chisq_dof", "bandwidth"};

        String[] columnFamilies = {"filter", "row_key", "channel",
                                   "channel", "time", "time", "frequency",
                                   "time", "time", "time", "statistics",
                                   "statistics", "statistics", "statistics",
                                   "statistics", "frequency"};

        new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[11]),
                                   Bytes.toBytes(columns[11]),
                                   CompareFilter.CompareOp.GREATER_OR_EQUAL,
                        new BinaryComparator(Bytes.toBytes("1.00000000")));

        Scan scan = new Scan();
        scan.setFilter(filter);
        ResultScanner scanner =table.getScanner(scan);
        for(Result result: scanner){
            List<Cell> colCells =
                result.getColumnCells(Bytes.toBytes(columnFamilies[11]),
                                      Bytes.toBytes(columns[11]));
            for(Cell cell : colCells){
                System.out.println("Cell:" + cell + ", Value: " +
                                   Bytes.toString(cell.getValueArray(),
                                              cell.getValueOffset(),
                                              cell.getValueLength()));
            }
        }
        scanner.close();
    }
}
```

13.5 FilterString.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import java.util.List;


public class FilterString{

    public static void main(String[] args) throws IOException, Exception{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.
                                 valueOf("gstlal_excesspower"));


        String[] columns = {"search", "event_index", "ifo", "channel",
                           "peak_time", "peak_time_ns", "central_freq",
                           "start_time", "start_time_ns", "duration",
                           "amplitude", "snr", "confidence", "chisq",
                           "chisq_dof", "bandwidth"};

        String[] columnFamilies = {"filter", "row_key", "channel",
                                  "channel", "time", "time", "frequency",
                                  "time", "time", "time", "statistics",
                                "statistics", "statistics", "statistics",
                                "statistics", "frequency"};

        SingleColumnValueFilter filter =
             new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[3]),
                                         Bytes.toBytes(columns[3]),
                                         CompareFilter.CompareOp.EQUAL,
                                         new SubstringComparator("STRAIN"));


        Scan scan = new Scan();
        scan.setFilter(filter);
        ResultScanner scanner =table.getScanner(scan);
        for(Result result: scanner){
            List<Cell> colCells =
                result.getColumnCells(Bytes.toBytes(columnFamilies[3]),
                                      Bytes.toBytes(columns[3]));
            for(Cell cell : colCells){
                System.out.println("Cell:" + cell + ", Value: " +
                                Bytes.toString(cell.getValueArray(),
                                               cell.getValueOffset(),
                                               cell.getValueLength()));
            }
        }
        scanner.close();
    }
}
```

13.6 FilterListClass.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import java.util.List;
import org.apache.hadoop.hbase.filter.FilterList;
import java.util.ArrayList;
import org.apache.hadoop.hbase.filter.RowFilter;


public class FilterListClass{

    public static void main(String[] args) throws IOException, Exception{

        Configuration config= HBaseConfiguration.create();

        Connection connection = ConnectionFactory.createConnection(config);
        Table table= connection.getTable(TableName.
                                valueOf("gstlal_excesspower"));


        String[] columns = {"search", "event_index", "ifo", "channel",
                            "peak_time", "peak_time_ns", "central_freq",
                            "start_time", "start_time_ns", "duration",
                            "amplitude", "snr", "confidence", "chisq",
                            "chisq_dof", "bandwidth"};

        String[] columnFamilies = {"filter", "row_key", "channel", "channel",
                                   "time", "time", "frequency", "time",
                                   "time", "time", "statistics", "statistics",
                                   "statistics", "statistics", "statistics",
                                   "frequency"};

        FilterList filters = new FilterList(FilterList.Operator.MUST_PASS_ALL);

        SingleColumnValueFilter snrgt1 =

          new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[11]),
                                    Bytes.toBytes(columns[11]),
                              CompareFilter.CompareOp.GREATER_OR_EQUAL,
                        new BinaryComparator(Bytes.toBytes("0.97000000")));
        filters.addFilter(snrgt1);

        SingleColumnValueFilter chanSTRAIN =
          new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[3]),
                                    Bytes.toBytes(columns[3]),
                                    CompareFilter.CompareOp.EQUAL,
                                    new SubstringComparator("STRAIN"));
        filters.addFilter(chanSTRAIN);

        RowFilter rowlt5000=
            new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,
            new BinaryComparator(Bytes.toBytes("sngl_burst:event_id:5000")));
        filters.addFilter(rowlt5000);

        RowFilter rowgt4800 =
            new RowFilter(CompareFilter.CompareOp.GREATER_OR_EQUAL,
            new BinaryComparator(Bytes.toBytes("sngl_burst:event_id:4800")));
        filters.addFilter(rowgt4800);



        Scan scan = new Scan();
        scan.setFilter(filters);
        ResultScanner scanner =table.getScanner(scan);
        for(Result result: scanner){
            List<Cell> colCells =
                result.getColumnCells(Bytes.toBytes(columnFamilies[11]),
                                      Bytes.toBytes(columns[11]));
            for(Cell cell : colCells){
                System.out.println("Cell:" + cell + ", Value: " +
                                   Bytes.toString(cell.getValueArray(),
                                                  cell.getValueOffset(),
                                                  cell.getValueLength()));
            }
        }
        scanner.close();
    }
}
```

13.7 SecondScanner.java

```
import java.io.IOException;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import java.util.List;
import org.apache.hadoop.hbase.filter.FilterList;
import java.util.ArrayList;
import org.apache.hadoop.hbase.filter.RowFilter;
import java.nio.ByteBuffer;

public class SecondScanner{

    public static void main(String[] args) throws IOException, Exception{
```

```
Configuration  config= HBaseConfiguration.create();

Connection connection = ConnectionFactory.createConnection(config);
Table table= connection.getTable(TableName.valueOf("gstlal_excesspower"));


String[] columns = {"search", "event_index", "ifo", "channel",
                    "peak_time", "peak_time_ns", "central_freq",
                    "start_time", "start_time_ns", "duration",
                    "amplitude", "snr", "confidence", "chisq",
                    "chisq_dof", "bandwidth"};

String[] columnFamilies = {"filter", "row_key", "channel", "channel",
                           "time", "time", "frequency", "time",
                           "time", "time", "statistics", "statistics",
                           "statistics", "statistics", "statistics",
                           "frequency"};

FilterList filters = new FilterList(FilterList.Operator.MUST_PASS_ALL);

SingleColumnValueFilter snrgt1 =

  new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[11]),
                              Bytes.toBytes(columns[11]),
                        CompareFilter.CompareOp.GREATER_OR_EQUAL,
             new BinaryComparator(Bytes.toBytes("0.97000000")));
filters.addFilter(snrgt1);

SingleColumnValueFilter chanSTRAIN =
  new SingleColumnValueFilter(Bytes.toBytes(columnFamilies[3]),
                              Bytes.toBytes(columns[3]),
                              CompareFilter.CompareOp.EQUAL,
                              new SubstringComparator("STRAIN"));
filters.addFilter(chanSTRAIN);

RowFilter rowlt5000= new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,
    new BinaryComparator(Bytes.toBytes("sngl_burst:event_id:5000")));
filters.addFilter(rowlt5000);

RowFilter rowgt4800 = new RowFilter(CompareFilter.CompareOp.GREATER_OR_EQUAL,
     new BinaryComparator(Bytes.toBytes("sngl_burst:event_id:4800")));
filters.addFilter(rowgt4800);



Scan scan = new Scan();
scan.setFilter(filters);
ResultScanner scanner =table.getScanner(scan);
List<Cell> finalCells = new ArrayList<Cell>();
for(Result result: scanner){
```

```
                Cell cell = result.getColumnLatestCell(Bytes.
                                toBytes(columnFamilies[12]),
                              Bytes.toBytes(columns[12]));
                if (ByteBuffer.wrap(cell.getValueArray()).getDouble()
                    <=25.300000){
             System.out.println(ByteBuffer.wrap(cell.getValueArray()).getDouble());

                    finalCells.add(cell);
                    finalCells.add(result.getColumnLatestCell(
                                Bytes.toBytes(columnFamilies[11]),
                                Bytes.toBytes(columns[11])));
                }
            }
            scanner.close();
            for(Cell cl: finalCells){
                System.out.println("Cell:" + cl + ", Value: " +
                                  Bytes.toString(cl.getValueArray(),
                                                 cl.getValueOffset(),
                                                 cl.getValueLength()));
            }
        }
}
```

14 Appendix C: Installing and running HBase

14.1 Installation

To install and run HBase in stand-alone mode, first follow the directions given in Reference [4], skipping over the XML configuration steps, and moving on to "Formatting the New Hadoop Filesystem". Then install HBase according to the stand-alone mode directions in Reference [5]. When it becomes time to install in pseudo-distributed mode, you will need to install ZooKeeper, which I have installed according to the directions in Reference [6].

14.2 Initializing and running HBase

The directions for creating a Hadoop user are covered in Reference [4]. The directions for starting and running HBase are covered in References [1, 2, 3]. I will quickly summarize them here. Suppose you have installed HBase under the user hduser and cloned the LIGOdatabase GitHub repository in Reference [8].

- sudo su hduser
- ssh localhost
- cd LIGOdatabase
- pwd
- export HBASE_CLASSPATH=< your path >
- export CLASSPATH=< your path >
- cd /usr/local/hadoop/sbin
- ./start-all.sh

- cd /usr/local/hbase/bin
- ./start-hbase.sh
- cd ~/LIGOdatabase

At this point you must decide whether you wish to run the HBase shell or the Java API. To summarize what has been done so far: we have switched to the user hduser, ssh-ed to localhost, set the variable HBASE_CLASSPATH equal to the LIGOdatabase directory, started Hadoop, started HBase, and returned to the LIGOdatabase directory.

14.2.1 Java API

To compile CreateTable.java, for instance, in the Java API, run

```
javac -cp `hbase classpath`:`hadoop classpath` CreateTable.java
```

To run it in stand-alone mode, run

```
javac -cp `hbase classpath`:`hadoop classpath` CreateTable
```

To run it, it must first be compiled. To compile it, the appropriate classpaths must first be set, and Hadoop and HBase must be started. All of this must be done in the right order.

14.2.2 HBase shell

There are several shell commands that I have used to verify the output from my Java API code.

- list – *Lists all the tables in the database*
- create – *I have not used this, but it is possible to create a table in the shell.*
- get 'table', 'row'
- get 'table', 'row', {COLUMN=>'column'}
- *It is also possible to filter in the shell using gets, but I have not.*
- scan 'table' – *Returns binary forms of all the values of all the rows, column families, and columns.*
- delete 'table', 'row', 'column'
- disable 'table' – *Disables an entire table, making it no longer able to accept changes.*
- drop 'table' – *Deletes a disabled table from the database.*