Gora Tutorial

Table of contents

Author : Enis Söztutar, enis [at] apache [dot] org

1 Introduction

This is the official tutorial for Apache Gora. For this tutorial, we will be implementing a system to store our web server logs in Apache HBase, and analyze the results using Apache Hadoop and store the results either in HSQLDB or MySQL.

In this tutorial we will first look at how to set up the environment and configure Gora and the data stores. Later, we will go over the data we will use and define the data beans that will be used to interact with the persistency layer. Next, we will go over the API of Gora to do some basic tasks such as storing objects, fetching and querying objects, and deleting objects. Last, we will go over an example program which uses Hadoop MapReduce to analyze the web server logs, and discuss the Gora MapReduce API in some detail.

1.1 Introduction to Gora

The Apache Gora open source framework provides an in-memory data model and persistence for big data. Gora supports persisting to column stores, key value stores, document stores and RDBMSs, and analyzing the data with extensive Apache Hadoop MapReduce support. In Avro, the beans to hold the data and RPC interfaces are defined using a JSON schema. In mapping the data beans to data store specific settings, Gora depends on mapping files, which are specific to each data store. Unlike other ORM implementations, Gora the data bean to data store specific schema mapping is explicit. This has the advantage that, when using data models such as HBase and Cassandra, you can always know how the values are persisted.

Gora has a modular architecture. Most of the data stores in Gora, has it's own module, such as `gora-hbase`, `gora-cassandra`, and `gora-sql`. In your projects, you need to only include the artifacts from the modules you use. You can consult the Setting up your project section in the quick start guide.

2 Setting up the environment

2.1 Setting up Gora

As a first step, we need to download and compile the Gora source code. The source codes for the tutorial is in the `gora-tutorial` module. If you have already downloaded Gora, that's cool, otherwise, please go over the steps at the Quick Start guide for how to download and compile Gora.

Now, after the source code for Gora is at hand, let's have a look at the files under the directory `gora-tutorial`.

```
$ cd gora-tutorial
$ tree
```

```
|-- build.xml
|-- conf
|   |-- gora-hbase-mapping.xml
|   |-- gora-sql-mapping.xml
|   `-- gora.properties
|-- ivy
|   `-- ivy.xml
`-- src
    |-- examples
    |   `-- java
    |-- main
    |   |-- avro
    |   |   |-- metricdatum.json
    |   |   `-- pageview.json
    |   |-- java
    |   |   `-- org
    |   |       `-- apache
    |   |           `-- gora
    |   |               `-- tutorial
    |   |                   `-- log
    |   |                       |-- KeyValueWritable.java
    |   |                       |-- LogAnalytics.java
    |   |                       |-- LogManager.java
    |   |                       |-- TextLong.java
    |   |                       `-- generated
    |   |                           |-- MetricDatum.java
    |   |                           `-- Pageview.java
    |   `-- resources
    |       `-- access.log.tar.gz
    `-- test
        |-- conf
        `-- java
```

Since gora-tutorial is a top level module of Gora, it depends on the directory structure imposed by Gora's main build scripts (`build.xml` and `build-common.xml` with Ivy and pom.xml for Maven). The Java source code resides in directory `src/main/java/`, avro schemas in `src/main/avro/`, and data in `src/main/resources/`.

2.2 Setting up HBase

For this tutorial we will be using HBase to store the logs. For those of you not familiar with HBase, it is a NoSQL column store with an architecture very similar to Google's BigTable.

If you don't already have already HBase setup, you can go over the steps at HBase Overview documentation. Although Gora aims to support the most recent HBase versions, the above tutorial is specifically for HBase 0.20.6 (don't worry the principals are the same), so download a version from HBase releases. After extracting the file, cd to the hbase-${dist} directory and start the HBase server.

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

and make sure that HBase is available by using the Hbase shell.

```
$ bin/hbase shell
```

2.3 Configuring Gora

Gora is configured through a file in the classpath named `gora.properties`. We will be using the following file `gora-tutorial/conf/gora.properties`

```
gora.datastore.default=org.apache.gora.hbase.store.HBaseStore
gora.datastore.autocreateschema=true
```

This file states that the default store will be `HBaseStore`, and schemas(tables) should be automatically created.

More information for configuring different settings in gora.properties can be found here .

3 Modelling the data

3.1 Data for the tutorial

For this tutorial, we will be parsing and storing the logs of a web server. Some example logs are at `src/main/resources/access.log.tar.gz`, which belongs to the (now shutdown) server at http://www.buldinle.com/. Example logs contain 10,000 lines, between dates 2009/03/10 - 2009/03/15.
The first thing, we need to do is to extract the logs.

```
$ tar zxvf src/main/resources/access.log.tar.gz -C src/main/
resources/
```

You can also use your own log files, given that the log format is Combined Log Format.
Some example lines from the log are:

```
88.254.190.73 - - [10/Mar/2009:20:40:26 +0200] "GET /
HTTP/1.1" 200 43 "http://www.buldinle.com/" "Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5; .NET CLR
2.0.50727; InfoPath.2)"
78.179.56.27 - - [11/Mar/2009:00:07:40 +0200] "GET /index.php?
i=3&a=1__6x39kovbji8&k=3750105 HTTP/1.1" 200 43 "http://
www.buldinle.com/index.php?i=3&a=1__6X39Kovbji8&k=3750105"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET
CLR 2.0.50727; OfficeLiveConnector.1.3; OfficeLivePatch.0.0)"
```

```
78.163.99.14 - - [12/Mar/2009:18:18:25 +0200] "GET /
index.php?a=3__x7l72c&k=4476881 HTTP/1.1" 200 43 "http://
www.buldinle.com/index.php?a=3__x7l72c&k=4476881" "Mozilla/4.0
(compatible; MSIE 7.0; Windows NT 5.1; InfoPath.1)"
```

The first fields in order are: User's ip, ignored, ignored, Date and time, HTTP method, URL, HTTP Method, HTTP status code, Number of bytes returned, Referrer, and User Agent.

3.2 Defining data beans

Data beans are the main way to hold the data in memory and persist in Gora. Gora needs to explicitly keep track of the status of the data in memory, so we use Apache Avro for defining the beans. Using avro gives us the possibility to explicitly keep track object's persistency state, and a way to serialize object's data.

Defining data beans is a very easy task, but for the exact syntax, please consult to Avro Specification.

First, we need to define the bean `Pageview` to hold a single URL access in the logs. Let's go over the class at `src/main/avro/pageview.json`

```
{
 "type": "record",
 "name": "Pageview",
 "namespace": "org.apache.gora.tutorial.log.generated",
 "fields" : [
   {"name": "url", "type": "string"},
   {"name": "timestamp", "type": "long"},
   {"name": "ip", "type": "string"},
   {"name": "httpMethod", "type": "string"},
   {"name": "httpStatusCode", "type": "int"},
   {"name": "responseSize", "type": "int"},
   {"name": "referrer", "type": "string"},
   {"name": "userAgent", "type": "string"}
 ]
}
```

Avro schemas are declared in JSON. Records are defined with type `"record"`, with a name as the name of the class, and a namespace which is mapped to the package name in Java. The fields are listed in the `"fields"` element. Each field is given with its type.

3.3 Compiling Avro Schemas

The next step after defining the data beans is to compile the schemas into Java classes. For that we will use `GoraCompiler`. Invoking the Gora compiler by (from Gora top level directory)

```
$ bin/gora compile
```

results in:

```
$ Usage: SpecificCompiler <schema file> <output dir>
```

so we will issue :

```
$ bin/gora compile gora-tutorial/src/main/avro/pageview.json
gora-tutorial/src/main/java/
```

to compile the Pageview class into `gora-tutorial/src/main/java/org/apache/gora/tutorial/log/generated/Pageview.java`. However, the tutorial java classes are already committed, so you do not need to do that now.

Gora compiler extends Avro's `SpecificCompiler` to convert JSON definition into a Java class. Generated classes extend the Persistent interface. Most of the methods of the `Persistent` interface deal with bookkeeping for persistence, and state tracking, so most of the time they are not used explicitly by the user. Now, let's look at the internals of the generated class `Pageview.java`.

```
public class Pageview extends PersistentBase {

  private Utf8 url;
  private long timestamp;
  private Utf8 ip;
  private Utf8 httpMethod;
  private int httpStatusCode;
  private int responseSize;
  private Utf8 referrer;
  private Utf8 userAgent;

  ...

  public static final Schema _SCHEMA = Schema.parse("{\"type\":\"record\", ... ");
  public static enum Field {
    URL(0,"url"),
    TIMESTAMP(1,"timestamp"),
    IP(2,"ip"),
    HTTP_METHOD(3,"httpMethod"),
    HTTP_STATUS_CODE(4,"httpStatusCode"),
    RESPONSE_SIZE(5,"responseSize"),
    REFERRER(6,"referrer"),
    USER_AGENT(7,"userAgent"),
    ;
    private int index;
    private String name;
    Field(int index, String name) {this.index=index;this.name=name;}
    public int getIndex() {return index;}
    public String getName() {return name;}
    public String toString() {return name;}
  };
  public static final String[] _ALL_FIELDS = {"url","timestamp","ip","httpMethod"
    ,"httpStatusCode","responseSize","referrer","userAgent",};

  ...
  }
```

We can see the actual field declarations in the class. Note that Avro uses `Utf8` class as a placeholder for string fields. We can also see the embedded Avro Schema declaration and an inner enum named `Field`. This enum and the `_ALL_FIELDS` field will come in handy when we will use them to query the datastore for specific fields.

3.4 Defining data store mappings

Gora is designed to flexibly work with various types of data modeling, including column stores(such as HBase, Cassandra, etc), SQL databases, flat files(binary, JSON, XML encoded), and key-value stores. The mapping between the data bean and the data store is thus defined in XML mapping files. Each data store has its own mapping format, so that data-store specific settings can be leveraged more easily. The mapping files declare how the fields of the classes declared in Avro schemas are serialized and persisted to the data store.

3.4.1 HBase mappings

HBase mappings are stored at file named `gora-hbase-mappings.xml`. For this tutorial we will be using the file `gora-tutorial/conf/gora-hbase-mappings.xml`.

```
<gora-orm>
  <table name="Pageview"> <!-- optional descriptors for tables -->
    <family name="common"/> <!-- This can also have params like compression, bloom filters
 -->
    <family name="http"/>
    <family name="misc"/>
  </table>

  <class name="org.apache.gora.tutorial.log.generated.Pageview" keyClass="java.lang.Long"
 table="AccessLog">
    <field name="url" family="common" qualifier="url"/>
    <field name="timestamp" family="common" qualifier="timestamp"/>
    <field name="ip" family="common" qualifier="ip" />
    <field name="httpMethod" family="http" qualifier="httpMethod"/>
    <field name="httpStatusCode" family="http" qualifier="httpStatusCode"/>
    <field name="responseSize" family="http" qualifier="responseSize"/>
    <field name="referrer" family="misc" qualifier="referrer"/>
    <field name="userAgent" family="misc" qualifier="userAgent"/>
  </class>

  ...

</gora-orm>
```

Every mapping file starts with the top level element `<gora-orm>`. Gora HBase mapping files can have two type of child elements, `table` and `class` declarations. All of the table and class definitions should be listed at this level.

`table` declaration is optional and most of the time, Gora infers the table declaration from the `class` sub elements. However, some of the HBase specific table configuration such as compression, blockCache, etc can be given here, if Gora is used to auto-create the tables. The exact syntax for the file can be found here.

In Gora, data store access is always done in a key-value data model, since most of the target backends support this model. DataStore API expects to know the class names of the key and persistent classes, so that they can be instantiated. The key value pair is declared in the `class` element. The `name` attribute is the fully qualified name of the class, and the `keyClass` attribute is the fully qualified class name of the key class.

Children of the `<class>` element are `<field>` elements. Each field element has a `name` and `family` attribute, and an optional `qualifier` attribute. `name` attribute contains the name of the field in the persistent class, and `family` declares the column family of the HBase data model. If the qualifier is not given, the name of the field is used as the column qualifier. Note that map and array type fields are stored in unique column families, so the configuration should be list unique column families for each map and array type, and no qualifier should be given. The exact data model is discussed further at the gora-hbase documentation.

4 Basic API

4.1 Parsing the logs

Now that we have the basic setup, we can see Gora API in action. As you can notice below the API is pretty simple to use. We will be using the class `LogManager` (which is located at `gora-tutorial/src/main/java/org/apache/gora/tutorial/log/LogManager.java`) for parsing and storing the logs, deleting some lines and querying.

First of all, let us look at the constructor. The only real thing it does is to call the `init()` method. `init()` method constructs the `DataStore` instance so that it can be used by the `LogManager`'s methods.

```
public LogManager() {
  try {
    init();
  } catch (IOException ex) {
    throw new RuntimeException(ex);
  }
}
private void init() throws IOException {
  dataStore = DataStoreFactory.getDataStore(Long.class, Pageview.class);
}
```

DataStore is probably the most important class in the Gora API. `DataStore` handles actual object persistence. Objects can be persisted, fetched, queried or deleted by the DataStore methods. Every data store that Gora supports, defines its own subclass of the DataStore class. For example `gora-hbase` module defines `HBaseStore`, and `gora-sql` module defines `SqlStore`. However, these subclasses are not explicitly used by the user.

DataStores always have associated key and value(persistent) classes. Key class is the class of the keys of the data store, and the value is the actual data bean's class. The value class is almost always generated by Avro schema definitions using the Gora compiler.

Data store objects are created by DataStoreFactory. It is necessary to provide the key and value class. The datastore class is optional, and if not specified it will be read from the configuration (gora.properties).

For this tutorial, we have already defined the avro schema to use and compiled our data bean into `Pageview` class. For keys in the data store, we will be using `Long`s. The keys will hold the line of the pageview in the data file.

Next, let's look at the main function of the `LogManager` class.

```
  public static void main(String[] args) throws Exception {
   if(args.length  2) {
     System.err.println(USAGE);
     System.exit(1);
   }

   LogManager manager = new LogManager();

   if("-parse".equals(args[0])) {
     manager.parse(args[1]);
   } else if("-query".equals(args[0])) {
     if(args.length == 2)
       manager.query(Long.parseLong(args[1]));
     else
       manager.query(Long.parseLong(args[1]), Long.parseLong(args[2]));
   } else if("-delete".equals(args[0])) {
     manager.delete(Long.parseLong(args[1]));
   } else if("-deleteByQuery".equalsIgnoreCase(args[0])) {
     manager.deleteByQuery(Long.parseLong(args[1]), Long.parseLong(args[2]));
   } else {
     System.err.println(USAGE);
     System.exit(1);
   }

   manager.close();
  }
```

We can use the example log manager program from the command line (in the top level Gora directory):

```
$ bin/gora logmanager
```

which lists the usage as:

```
LogManager -parse <input_log_file>
           -get <lineNum>
           -query <lineNum>
           -query <startLineNum> <endLineNum>
           -delete <lineNum>
           -deleteByQuery <startLineNum> <endLineNum>
```

So to parse and store our logs located at `gora-tutorial/src/main/resources/access.log`, we will issue:

```
$ bin/gora logmanager -parse gora-tutorial/src/main/resources/access.log
```

This should output something like:

```
10/09/30 18:30:17 INFO log.LogManager: Parsing file:gora-tutorial/src/main/resources/
access.log
10/09/30 18:30:23 INFO log.LogManager: finished parsing file. Total number of log
 lines:10000
```

Now, let's look at the code which parses the data and stores the logs.

```
  private void parse(String input) throws IOException, ParseException {
    BufferedReader reader = new BufferedReader(new FileReader(input));
    long lineCount = 0;
    try {
      String line = reader.readLine();
      do {
        Pageview pageview = parseLine(line);

        if(pageview != null) {
          //store the pageview
          storePageview(lineCount++, pageview);
        }

        line = reader.readLine();
      } while(line != null);

    } finally {
      reader.close();
    }
  }
```

The file is iterated line-by-line. Notice that the `parseLine(line)` function does the actual parsing converting the string to a `Pageview` object defined earlier.

```
  private Pageview parseLine(String line) throws ParseException {
```

```
    StringTokenizer matcher = new StringTokenizer(line);
    //parse the log line
    String ip = matcher.nextToken();
    ...

    //construct and return pageview object
    Pageview pageview = new Pageview();
    pageview.setIp(new Utf8(ip));
    pageview.setTimestamp(timestamp);
    ...

    return pageview;
  }
```

`parseLine()` uses standard `StringTokenizer`s for the job and constructs and returns a `Pageview` object.

4.2 Storing objects in the DataStore

If we look back at the `parse()` method above, we can see that the `Pageview` objects returned by `parseLine()` are stored via `storePageview()` method.

The storePageview() method is where magic happens, but if we look at the code, we can see that it is dead simple.

```
  /** Stores the pageview object with the given key */
  private void storePageview(long key, Pageview pageview) throws IOException {
    dataStore.put(key, pageview);
  }
```

All we need to do is to call the put() method, which expects a long as key and an instance of `Pageview` as a value.

4.3 Closing the DataStore

`DataStore` implementations can do a lot of caching for performance. However, this means that data is not always flushed to persistent storage all the times. So we need to make sure that upon finishing storing objects, we need to close the datastore instance by calling it's close() method. LogManager always closes it's datastore in it's own `close()` method.

```
  private void close() throws IOException {
    //It is very important to close the datastore properly, otherwise
    //some data loss might occur.
    if(dataStore != null)
      dataStore.close();
  }
```

If you are pushing a lot of data, or if you want your data to be accessible before closing the data store, you can also the flush() method which, as expected, flushes the data to the underlying data store. However, the actual flush semantics can vary by the data store backend. For example, in SQL flush calls `commit()` on the jdbc `Connection` object, whereas in Hbase, `HTable#flush()` is called. Also note that even if you call `flush()` at the end of all data manipulation operations, you still need to call the `close()` on the datastore.

4.4 Persisted data in HBase

Now that we have stored the web access log data in HBase, we can look at how the data is stored at HBase. For that, start the HBase shell.

```
$ cd ../hbase-0.20.6
$ bin/hbase shell
```

If you have a fresh HBase installation, there should be one table.

```
hbase(main):010:0> list
```

```
AccessLog

1 row(s) in 0.0470 seconds
```

Remember that AccessLog is the name of the table we specified at `gora-hbase-mapping.xml`. Looking at the contents of the table:

```
hbase(main):010:0> scan 'AccessLog', {LIMIT=>1}
```

```
ROW                          COLUMN+CELL

 \x00\x00\x00\x00\x00\x00\x0 column=common:ip, timestamp=1285860617341,
 value=88.240.129.183
 0\x00

 \x00\x00\x00\x00\x00\x00\x0 column=common:timestamp, timestamp=1285860617341, value=
\x00\x00\x01\x1F\xF1\xAEl
 0\x00                       P

 \x00\x00\x00\x00\x00\x00\x0 column=common:url, timestamp=1285860617341, value=/index.php?
a=1__wwv40pdxdpo&k=2
 0\x00                       18978

 \x00\x00\x00\x00\x00\x00\x0 column=http:httpMethod, timestamp=1285860617341, value=GET

 0\x00

 \x00\x00\x00\x00\x00\x00\x0 column=http:httpStatusCode, timestamp=1285860617341, value=
\x00\x00\x00\xC8
```

```
 0\x00

 \x00\x00\x00\x00\x00\x00\x0 column=http:responseSize, timestamp=1285860617341, value=
\x00\x00\x00+
 0\x00

 \x00\x00\x00\x00\x00\x00\x0 column=misc:referrer, timestamp=1285860617341, value=http://
www.buldinle.com/inde
 0\x00                       x.php?a=1__WWV40pdxdpo&k=218978

 \x00\x00\x00\x00\x00\x00\x0 column=misc:userAgent, timestamp=1285860617341,
 value=Mozilla/4.0 (compatible; MS
 0\x00                       IE 6.0; Windows NT 5.1)
```

The output shows all the columns matching the first line with key 0. We can see the columns `common:ip, common:timestamp, common:url,` etc. Remember that these are the columns that we have described in the `gora-hbase-mapping.xml` file.

You can also count the number of entries in the table to make sure that all the records have been stored.

```
hbase(main):010:0> count 'AccessLog'
```

```
...
10000 row(s) in 1.0580 seconds
```

4.5 Fetching objects from data store

Fetching objects from the data store is as easy as storing them. There are essentially two methods for fetching objects. First one is to fetch a single object given it's key. The second method is to run a query through the data store.

To fetch objects one by one, we can use one of the overloaded get() methods. The method with signature `get(K key)` returns the object corresponding to the given key fetching all the fields. On the other hand `get(K key, String[] fields)` returns the object corresponding to the given key, but fetching only the fields given as the second argument.

When run with the argument -get `LogManager` class fetches the pageview object from the data store and prints the results.

```
  /** Fetches a single pageview object and prints it*/
  private void get(long key) throws IOException {
    Pageview pageview = dataStore.get(key);
    printPageview(pageview);
  }
```

To display the 42nd line of the access log :

```
$ bin/gora logmanager -get 42
```

```
org.apache.gora.tutorial.log.generated.Pageview@321ce053 {
  "url":"/index.php?i=0&a=1__rntjt9z0q9w&k=398179"
  "timestamp":"1236710649000"
  "ip":"88.240.129.183"
  "httpMethod":"GET"
  "httpStatusCode":"200"
  "responseSize":"43"
  "referrer":"http://www.buldinle.com/index.php?i=0&a=1__RnTjT9z0Q9w&k=398179"
  "userAgent":"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
}
```

4.6 Querying objects

DataStore API defines a Query interface to query the objects at the data store. Each data store implementation can use a specific implementation of the `Query` interface. Queries are instantiated by calling DataStore#newQuery(). When the query is run through the datastore, the results are returned via the Result interface. Let's see how we can run a query and display the results below in the the LogManager class.

```
  /** Queries and prints pageview object that have keys between startKey and endKey*/
  private void query(long startKey, long endKey) throws IOException {
    Query<Long, Pageview> query = dataStore.newQuery();
    //set the properties of query
    query.setStartKey(startKey);
    query.setEndKey(endKey);

    Result<Long, Pageview> result = query.execute();

    printResult(result);
  }
```

After constructing a Query, its properties are set via the setter methods. Then calling query.execute() returns the Result object.

Result interface allows us to iterate the results one by one by calling the next() method. The getKey() method returns the current key and get() returns current persistent object.

```
  private void printResult(Result<Long, Pageview> result) throws IOException {

    while(result.next()) { //advances the Result object and breaks if at end
      long resultKey = result.getKey(); //obtain current key
      Pageview resultPageview = result.get(); //obtain current value object

      //print the results
      System.out.println(resultKey + ":");
      printPageview(resultPageview);
```

```
    }

    System.out.println("Number of pageviews from the query:" + result.getOffset());
  }
```

With these functions defined, we can run the Log Manager class, to query the access logs at HBase. For example, to display the log records between lines 10 and 12 we can use

```
bin/gora logmanager -query 10 12
```

Which results in:

```
10:
org.apache.gora.tutorial.log.generated.Pageview@d38d0eaa {
  "url":"/"
  "timestamp":"1236710442000"
  "ip":"144.122.180.55"
  "httpMethod":"GET"
  "httpStatusCode":"200"
  "responseSize":"43"
  "referrer":"http://buldinle.com/"
  "userAgent":"Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.0.6) Gecko/2009020911
 Ubuntu/8.10 (intrepid) Firefox/3.0.6"
}
11:
org.apache.gora.tutorial.log.generated.Pageview@b513110a {
  "url":"/index.php?i=7&a=1__gefuumyhl5c&k=5143555"
  "timestamp":"1236710453000"
  "ip":"85.100.75.104"
  "httpMethod":"GET"
  "httpStatusCode":"200"
  "responseSize":"43"
  "referrer":"http://www.buldinle.com/index.php?i=7&a=1__GeFUuMyHl5c&k=5143555"
  "userAgent":"Mozilla/5.0 (Windows; U; Windows NT 5.1; tr; rv:1.9.0.7) Gecko/2009021910
 Firefox/3.0.7"
}
```

4.7 Deleting objects

Just like fetching objects, there are two main methods to delete objects from the data store. The first one is to delete objects one by one using the DataStore#delete(K) method, which takes the key of the object. Alternatively we can delete all of the data that matches a given query by calling the DataStore#deleteByQuery(Query) method. By using deleteByQuery, we can do fine-grain deletes, for example deleting just a specific field from several records.

Continueing from the LogManager class, the api's for both are given below.

```
  /**Deletes the pageview with the given line number */
  private void delete(long lineNum) throws Exception {
    dataStore.delete(lineNum);
    dataStore.flush(); //write changes may need to be flushed before
```

```
                        //they are committed
  }

  /** This method illustrates delete by query call */
  private void deleteByQuery(long startKey, long endKey) throws IOException {
    //Constructs a query from the dataStore. The matching rows to this query will be
deleted
    QueryLong, Pageview> query = dataStore.newQuery();
    //set the properties of query
    query.setStartKey(startKey);
    query.setEndKey(endKey);

    dataStore.deleteByQuery(query);
  }
```

And from the command line :

```
bin/gora logmanager -delete 12
bin/gora logmanager -deleteByQuery 40 50
```

5 MapReduce Support

Gora has first class MapReduce support for Apache Hadoop. Gora data stores can be used as inputs and outputs of jobs. Moreover, the objects can be serialized, and passed between tasks keeping their persistency state. For the serialization, Gora extends Avro DatumWriters.

5.1 Log analytics in MapReduce

For this part of the tutorial, we will be analyzing the logs that have been stored at HBase earlier. Specifically, we will develop a MapReduce program to calculate the number of daily pageviews for each URL in the site.

We will be using the `LogAnalytics` class to analyze the logs, which can be found at `gora-tutorial/src/main/java/org/apache/gora/tutorial/log/LogAnalytics.java`. For computing the analytics, the mapper takes in pageviews, and outputs tuples of <URL, timestamp> pairs, with 1 as the value. The timestamp represents the day in which the pageview occurred, so that the daily pageviews are accumulated. The reducer just sums up the values, and outputs `MetricDatum` objects to be sent to the output Gora data store.

5.2 Setting up the environment

We will be using the logs stored at HBase by the `LogManager` class. We will push the output of the job to an HSQL database, since it has a zero conf set up. However, you can also use MySQL or HBase for storing the analytics results. If you want to continue with HBase, you can skip the next sections.

5.2.1 Setting up the database

First we need to download HSQL dependencies. For that, uncomment the following line from `gora-tutorial/ivy/ivy.xml` (if using Maven hsqldb should already be available). Ofcourse MySQL users should uncomment the mysql dependency instead.

```
<!--<dependency org="org.hsqldb" name="hsqldb" rev="2.0.0" conf="*->default"/>-->
```

Then we need to run ant so that the new dependencies can be downloaded.

```
$ ant
```

If you are using Mysql, you should also setup the database server, create the database and give necessary permissions to create tables, etc so that Gora can run properly.

5.2.2 Configuring Gora

We will put the configuration necessary to connect to the database to `gora-tutorial/conf/gora.properties`.

```
#JDBC properties for gora-sql module using HSQL
gora.sqlstore.jdbc.driver=org.hsqldb.jdbcDriver
gora.sqlstore.jdbc.url=jdbc:hsqldb:hsql://localhost/goratest

#JDBC properties for gora-sql module using MySQL
#gora.sqlstore.jdbc.driver=com.mysql.jdbc.Driver
#gora.sqlstore.jdbc.url=jdbc:mysql://localhost:3306/goratest
#gora.sqlstore.jdbc.user=root
#gora.sqlstore.jdbc.password=
```

As expected the `jdbc.driver` property is the JDBC driver class, and `jdbc.url` is the JDBC connection URL. Moreover `jdbc.user` and `jdbc.password` can be specific is needed. More information for these parameters can be found at gora-sql documentation.

5.3 Modelling the data

5.3.1 Data Beans for Analytics

For web site analytics, we will be using a generic `MetricDatum` data structure. It holds a string `metricDimension`, a long `timestamp`, and a long `metric` fields. The first two fields are the dimensions of the web analytics data, and the last is the actual aggregate metric value. For example we might have an instance `{metricDimension="/index", timestamp=101, metric=12}`, representing that there have been 12 pageviews to the URL "/index" for the given time interval 101.

The avro schema definition for `MetricDatum` can be found at `gora-tutorial/src/main/avro/metricdatum.json`, and the compiled source code at `gora-tutorial/src/main/java/org/apache/gora/tutorial/log/generated/MetricDatum.java`.

```
{
  "type": "record",
  "name": "MetricDatum",
  "namespace": "org.apache.gora.tutorial.log.generated",
  "fields" : [
    {"name": "metricDimension", "type": "string"},
    {"name": "timestamp", "type": "long"},
    {"name": "metric", "type" : "long"}
  ]
}
```

5.3.2 Data store mappings

We will be using the SQL backend to store the job output data, just to demonstrate the SQL backend.

Similar to what we have seen with HBase, gora-sql plugin reads configuration from the `gora-sql-mappings.xml` file. Specifically, we will use the `gora-tutorial/conf/gora-sql-mappings.xml` file.

```
<gora-orm>
  ...
  <class name="org.apache.gora.tutorial.log.generated.MetricDatum"
 keyClass="java.lang.String" table="Metrics">
    <primarykey column="id" length="512"/>
    <field name="metricDimension" column="metricDimension" length="512"/>
    <field name="timestamp" column="ts"/>
    <field name="metric" column="metric/>
  </class>
</gora-orm>
```

SQL mapping files contain one or more `class` elements as the children of `gora-orm`. The key value pair is declared in the `class` element. The `name` attribute is the fully qualified name of the class, and the `keyClass` attribute is the fully qualified class name of the key class.

Children of the `class` element are `field` elements and one `primaryKey` element. Each `field` element has a `name` and `column` attribute, and optional `jdbc-type`, `length` and `scale` attributes. `name` attribute contains the name of the field in the persistent class, and `column` attribute is the name of the column in the database. The `primaryKey` holds the

actual key as the primary key field. Currently, Gora only supports tables with one primary key.

5.4 Constructing the job

In constructing the job object for Hadoop, we need to define whether we will use Gora as job input, output or both. Gora defines its own GoraInputFormat, and GoraOutputFormat, which uses `DataStore`'s as input sources and output sinks for the jobs. `Gora{In|Out}putFormat` classes define static methods to set up the job properly. However, if the mapper or reducer extends Gora's mapper and reducer classes, you can use the static methods defined in GoraMapper and GoraReducer since they are more convenient.

For this tutorial we will use Gora as both input and output. As can be seen from the `createJob()` function, quoted below, we create the job as normal, and set the input parameters via GoraMapper#initMapperJob(), and GoraReducer#initReducerJob() . `GoraMapper#initMapperJob()` takes a store and an optional query to fetch the data from. When a query is given, only the results of the query is used as the input of the job, if not all the records are used. The actual Mapper, map output key and value classes are passed to `initMapperJob()` function as well. `GoraReducer#initReducerJob()` accepts the data store to store the job's output as well as the actual reducer class. `initMapperJob` and `initReducerJob` functions have also overriden methods that take the data store class rather than data store instances.

```
public Job createJob(DataStore<Long, Pageview> inStore
    , DataStore<String, MetricDatum> outStore, int numReducer) throws IOException {
  Job job = new Job(getConf());

  job.setJobName("Log Analytics");
  job.setNumReduceTasks(numReducer);
  job.setJarByClass(getClass());

  /* Mappers are initialized with GoraMapper.initMapper() or
   * GoraInputFormat.setInput()*/
  GoraMapper.initMapperJob(job, inStore, TextLong.class, LongWritable.class
      , LogAnalyticsMapper.class, true);

  /* Reducers are initialized with GoraReducer#initReducer().
   * If the output is not to be persisted via Gora, any reducer
   * can be used instead. */
  GoraReducer.initReducerJob(job, outStore, LogAnalyticsReducer.class);

  return job;
}
```

5.5 Gora mappers and using Gora an input

Typically, if Gora is used as job input, the Mapper class extends GoraMapper. However, currently this is not forced by the API so other class hierarchies can be used instead. The mapper receives the key value pairs that are the results of the input query, and emits the results of the custom map task. Note that output records from map are independent from the input and output data stores, so any Hadoop serializable key value class can be used. However, Gora persistent classes are also Hadoop serializable. Hadoop serialization is handled by the PersistentSerialization class. Gora also defines a StringSerialization class, to serialize strings easily.

Coming back to the code for the tutorial, we can see that `LogAnalytics` class defines an inner class `LogAnalyticsMapper` which extends `GoraMapper`. The map function receives `Long` keys which are the line numbers, and `Pageview` values as read from the input data store. The map simply rolls up the timestamp up to the day (meaning that only the day of the timestamp is used), and outputs the key as a tuple of `<URL,day>`.

```
private TextLong tuple;

protected void map(Long key, Pageview pageview, Context context)
  throws IOException ,InterruptedException {

  Utf8 url = pageview.getUrl();
  long day = getDay(pageview.getTimestamp());

  tuple.getKey().set(url.toString());
  tuple.getValue().set(day);

  context.write(tuple, one);
};
```

5.6 Gora reducers and using Gora as output

Similar to the input, typically, if Gora is used as job output, the Reducer extends GoraReducer. The values emitted by the reducer are persisted to the output data store as a result of the job.

For this tutorial, the `LogAnalyticsReducer` inner class, which extends `GoraReducer`, is used as the reducer. The reducer just sums up all the values that correspond to the `<URL,day>` tuple. Then the metric dimension object is constructed and emitted, which will be stored at the output data store.

```
protected void reduce(TextLong tuple
    , Iterable<LongWritable> values, Context context)
  throws IOException ,InterruptedException {
```

```
    long sum = 0L; //sum up the values
    for(LongWritable value: values) {
      sum+= value.get();
    }

    String dimension = tuple.getKey().toString();
    long timestamp = tuple.getValue().get();

    metricDatum.setMetricDimension(new Utf8(dimension));
    metricDatum.setTimestamp(timestamp);

    String key = metricDatum.getMetricDimension().toString();
    metricDatum.setMetric(sum);

    context.write(key, metricDatum);
  };
```

5.7 Running the job

Now that the job is constructed, we can run the Hadoop job as usual. Note that the `run` function of the `LogAnalytics` class parses the arguments and runs the job. We can run the program by

```
$ bin/gora loganalytics [<input data store> [<output data store>]]
```

5.7.1 Running the job with SQL

Now, let's run the log analytics tools with the SQL backend(either Hsql or MySql). The input data store will be `org.apache.gora.hbase.store.HBaseStore` and output store will be `org.apache.gora.sql.store.SqlStore`. Remember that we have already configured the database connection properties and which database will be used at the Setting up the environment section.

```
$ bin/gora loganalytics org.apache.gora.hbase.store.HBaseStore org.apache.gora.sql.store.SqlStore
```

Now we should see some logging output from the job, and whether it finished with success. To check out the output if we are using HSQLDB, below command can be used.

```
$ java -jar gora-tutorial/lib/hsqldb-2.0.0.jar
```

In the connection URL, the same URL that we have provided in gora.properties should be used. If on the other hand MySQL is used, than we should be able to see the output using the mysql command line utility.

The results of the job are stored at the table Metrics, which is defined at the `gora-sql-mapping.xml` file. Running a select query over this data confirms that the daily pageview metrics for the web site is indeed stored. To see the most popular pages, run:

```
> SELECT METRICDIMENSION, TS, METRIC FROM metrics order by metric desc
```

METRICDIMENSION	TS	METRIC
/	1236902400000	220
/	1236988800000	212
/	1236816000000	191
/	1237075200000	155
/	1241395200000	111
/	1236643200000	110
/	1236729600000	95
/index.php?a=3__x8g0vi&k=5508310	1236816000000	45
/index.php?a=1__5kf9nvgrzos&k=208773	1236816000000	37
...	...	...

As you can see, the home page (`/`) for varios days and some other pages are listed. In total 3033 rows are present at the metrics table.

5.7.2 Running the job with HBase

Since HBaseStore is already defined as the default data store at `gora.properties` we can run the job with HBase as:

```
$ bin/gora loganalytics
```

The outputs of the job will be saved in the Metrics table, whose layout is defined at `gora-hbase-mapping.xml` file. To see the results:

```
hbase(main):010:0> scan 'Metrics', {LIMIT=>1}
```

```
ROW                          COLUMN+CELL
 /?a=1__-znawtuabsy&k=96804_ column=common:metric, timestamp=1289815441740, value=
\x00\x00\x00\x00\x00\x00\x00
 1236902400000               \x09
```

```
 /?a=1__-znawtuabsy&k=96804_ column=common:metricDimension, timestamp=1289815441740,
 value=/?a=1__-znawtuabsy&
 1236902400000               k=96804
 /?a=1__-znawtuabsy&k=96804_ column=common:ts, timestamp=1289815441740, value=
\x00\x00\x01\x1F\xFD \xD0\x00
 1236902400000
1 row(s) in 0.0490 seconds
```

6 More Examples

Other than this tutorial, there are several places that you can find examples of Gora in action.

The first place to look at is the examples directories under various Gora modules. All the modules have a `<gora-module>/src/examples/` directory under which some example classes can be found. Especially, there are some classes that are used for tests under `<gora-core>/src/examples/`

Second, various unit tests of Gora modules can be referred to see the API in use. The unit tests can be found at `<gora-module>/src/test/`

The source code for the projects using Gora can also be checked out as a reference. Apache Nutch is one of the first class users of Gora; so looking into how Nutch uses Gora is always a good idea.

Please feel free to grab our poweredBy sticker and embedded it in anything backed by Apache Gora.

7 Feedback

At last, thanks for trying out Gora. If you find any bugs or you have suggestions for improvement, do not hesitate to give feedback on the dev@gora.apache.org mailing list.