Apache Avro With Apache Hadoop

By Abdulbasit F Shaikh

Avro

Apache Avro™ is a data serialization system.

Avro provides:

- Rich data structures.
- A compact, fast, binary data format.
- A container file, to store persistent data.
- Remote procedure call (RPC).
- Simple integration with dynamic languages. Code generation is not required to read or write data files nor to use or implement RPC protocols. Code generation as an optional optimization, only worth implementing for statically typed languages.

Schemas

Avro relies on *schemas*. When Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing.

When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be easily resolved, since both schemas are present.

When Avro is used in RPC, the client and server exchange schemas in the connection handshake. (This can be optimized so that, for most calls, no schemas are actually transmitted.) Since both client and server both have the other's full schema, correspondence between same named fields, missing fields, extra fields, etc. can all be easily resolved.

Avro schemas are defined with JSON . This facilitates implementation in languages that already have JSON libraries.

Comparison with other systems

Avro provides functionality similar to systems such as Thrift, Protocol Buffers, etc. Avro differs from these systems in the following fundamental aspects.

- *Dynamic typing*: Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation, static datatypes, etc. This facilitates construction of generic data-processing systems and languages.
- *Untagged data*: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.
- *No manually-assigned field IDs*: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names.

Steps For Installation Of Avro

1. Avro implementations for C, C++, C#, Java, PHP, Python, and Ruby can be downloaded from the http://apache.cs.utah.edu/avro/stable/ page.For the examples in this guide, download *avro-tools-1.7.4.jar from* *http://mvnrepository.com/artifact/org.apache.avro/avro-tools/1.7.4*.
2. Extract it.

Integration of Avro with HADOOP

Avro provides a convenient way to represent complex data structures within a Hadoop MapReduce job. Avro data can be used as both input to and output from a MapReduce job, as well as the intermediate format. The example in this guide uses Avro data for all three, but it's possible to mix and match; for instance, MapReduce can be used to aggregate a particular field in an Avro record.

This guide assumes basic familiarity with both Hadoop MapReduce and Avro.

Setup

The code from this guide is included in the Avro docs under *examples/mr-example*. The example is set up as a Maven project that includes the necessary Avro and MapReduce dependencies and the Avro Maven plugin for code generation, so no external jars are needed to run the example. In particular, the POM includes the following dependencies:

```
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>1.7.4</version>
</dependency>
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-mapred</artifactId>
  <version>1.7.4</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-core</artifactId>
  <version>1.1.0</version>
</dependency>
```

And the following plugin:

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>1.7.4</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
```

```
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/../</sourceDirectory>
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Alternatively, Avro jars can be downloaded directly from the http://avro.apache.org/releases.html page. The relevant Avro jars for this guide are *avro-1.7.4.jar* and *avro-mapred-1.7.4.jar*, as well as *avro-tools-1.7.4.jar* for code generation and viewing Avro data files as JSON. In addition, you will need to install Hadoop in order to use MapReduce.

Example: ColorCount

Below is a simple example of a MapReduce that uses Avro. This example can be found in the Avro docs under *examples/mr-example/src/main/java/example/ColorCount.java*. We'll go over the specifics of what's going on in subsequent sections.

```
package example;

import java.io.IOException;

import org.apache.avro.*;
import org.apache.avro.Schema.Type;
import org.apache.avro.mapred.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

import example.avro.User;
```

```
public class ColorCount extends Configured implements Tool {

  public static class ColorCountMapper extends AvroMapper<User,
Pair<CharSequence, Integer>> {
    @Override
    public void map(User user, AvroCollector<Pair<CharSequence, Integer>>
collector, Reporter reporter)
        throws IOException {
      CharSequence color = user.getFavoriteColor();
      // We need this check because the User.favorite_color field has type ["string",
"null"]
      if (color == null) {
        color = "none";
      }
      collector.collect(new Pair<CharSequence, Integer>(color, 1));
    }
  }

  public static class ColorCountReducer extends AvroReducer<CharSequence,
Integer,
                                                    Pair<CharSequence, Integer>> {
    @Override
    public void reduce(CharSequence key, Iterable<Integer> values,
                       AvroCollector<Pair<CharSequence, Integer>> collector,
                       Reporter reporter)
        throws IOException {
      int sum = 0;
      for (Integer value : values) {
        sum += value;
      }
      collector.collect(new Pair<CharSequence, Integer>(key, sum));
    }
  }

  public int run(String[] args) throws Exception {
```

```
    if (args.length != 2) {
      System.err.println("Usage: ColorCount <input path> <output path>");
      return -1;
    }

    JobConf conf = new JobConf(getConf(), ColorCount.class);
    conf.setJobName("colorcount");

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    AvroJob.setMapperClass(conf, ColorCountMapper.class);
    AvroJob.setReducerClass(conf, ColorCountReducer.class);

    // Note that AvroJob.setInputSchema and AvroJob.setOutputSchema set
    // relevant config options such as input/output format, map output
    // classes, and output key class.
    AvroJob.setInputSchema(conf, User.SCHEMA$);
    AvroJob.setOutputSchema(conf,
Pair.getPairSchema(Schema.create(Type.STRING),
        Schema.create(Type.INT)));

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new ColorCount(), args);
    System.exit(res);
  }
}
```

ColorCount reads in data files containing User records, defined in *examples/user.avsc*, and counts the number of instances of each favorite color. (This example draws

inspiration from the canonical WordCount MapReduce application.) The User schema is defined as follows:

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

This schema is compiled into the User class used by ColorCount via the Avro Maven plugin (see *examples/mr-example/pom.xml* for how this is set up).

ColorCountMapper essentially takes a User as input and extracts the User's favorite color, emitting the key-value pair <*favoriteColor*, 1>. ColorCountReducer then adds up how many occurrences of a particular favorite color were emitted, and outputs the result as a Pair record. These Pairs are serialized to an Avro data file.

Running ColorCount

The ColorCount application is provided as a Maven project in the Avro docs under *examples/mr-example*. To build the project, including the code generation of the User schema, run:

```
mvn compile
```

Next, run GenerateData to create an Avro data file, *input/users.avro*, containing 20 Users with favorite colors chosen randomly from a list:

```
mvn exec:java -q -Dexec.mainClass=example.GenerateData
```

P.S : If you are getting an error of asm.x.x.jar,then follow the below steps

1) Download asm.x.x.jar from
http://www.java2s.com/Code/Jar/a/Downloadasm31jar.htm

2) copy the asm.x.x.jar to your .m2/repository/asm/asm/asm.x.x.jar.It will replace older jar with newer jar

3) Then try mvn test.

4) You will get Build Is Succesfull.

Besides creating the data file, GenerateData prints the JSON representations of the Users generated to stdout, for example:

```
{"name": "user", "favorite_number": null, "favorite_color": "red"}
{"name": "user", "favorite_number": null, "favorite_color": "green"}
{"name": "user", "favorite_number": null, "favorite_color": "purple"}
{"name": "user", "favorite_number": null, "favorite_color": null}
...
```

Now we're ready to run ColorCount. We specify our freshly-generated *input* folder as the input path and *output* as our output folder (note that MapReduce will not start a job if the output folder already exists):

```
mvn exec:java -q -Dexec.mainClass=example.ColorCount -Dexec.args="input output"
```

Once ColorCount completes, checking the contents of the new *output* directory should yield the following:

```
$ ls output/
part-00000.avro  _SUCCESS
```

You can check the contents of the generated Avro file using the avro-tools jar:

```
$ java -jar /path/to/avro-tools-1.7.4.jar tojson output/part-00000.avro
```

```
{"value": 3, "key": "blue"}
{"value": 7, "key": "green"}
{"value": 1, "key": "none"}
{"value": 2, "key": "orange"}
{"value": 3, "key": "purple"}
{"value": 2, "key": "red"}
{"value": 2, "key": "yellow"}
```

Now let's go over the ColorCount example in detail.

AvroMapper

The easiest way to use Avro data files as input to a MapReduce job is to subclass AvroMapper. An AvroMapper defines a map function that takes an Avro datum as input and outputs a key/value pair represented as a Pair record. In the ColorCount example, ColorCountMapper is an AvroMapper that takes a User as input and outputs a Pair<CharSequence, Integer>>, where the CharSequence key is the user's favorite color and the Integer value is 1.

```
public static class ColorCountMapper extends AvroMapper<User,
Pair<CharSequence, Integer>> {
  @Override
  public void map(User user, AvroCollector<Pair<CharSequence, Integer>>
collector, Reporter reporter)
      throws IOException {
    CharSequence color = user.getFavoriteColor();
    // We need this check because the User.favorite_color field has type ["string",
"null"]
    if (color == null) {
      color = "none";
    }
    collector.collect(new Pair<CharSequence, Integer>(color, 1));
  }
}
```

In order to use our AvroMapper, we must call AvroJob.setMapperClass and AvroJob.setInputSchema.

```
AvroJob.setMapperClass(conf, ColorCountMapper.class);
AvroJob.setInputSchema(conf, User.SCHEMA$);
```

Note that AvroMapper does not implement the Mapper interface. Under the hood, the specified Avro data files are deserialized into AvroWrappers containing the actual data, which are processed by a Mapper that calls the configured AvroMapper's map function. AvroJob.setInputSchema sets up the relevant configuration parameters needed to make this happen, thus you should not need to call JobConf.setMapperClass, JobConf.setInputFormat, JobConf.setMapOutputKeyClass, JobConf.setMapOutputValueClass, or JobConf.setOutputKeyComparatorClass.

AvroReducer

Analogously to AvroMapper, an AvroReducer defines a reducer function that takes the key/value types output by an AvroMapper (or any mapper that outputs Pairs) and outputs a key/value pair represented a Pair record. In the ColorCount example, ColorCountReducer is an AvroReducer that takes the CharSequence key representing a favorite color and the Iterable<Integer> representing the counts for that color (they should all be 1 in this example) and adds up the counts.

```
public static class ColorCountReducer extends AvroReducer<CharSequence,
Integer,
                                                  Pair<CharSequence, Integer>> {
 @Override
 public void reduce(CharSequence key, Iterable<Integer> values,
                    AvroCollector<Pair<CharSequence, Integer>> collector,
                    Reporter reporter)
   throws IOException {
  int sum = 0;
  for (Integer value : values) {
   sum += value;
  }
  collector.collect(new Pair<CharSequence, Integer>(key, sum));
```

```
  }
}
```

In order to use our AvroReducer, we must call AvroJob.setReducerClass and AvroJob.setOutputSchema.

```
AvroJob.setReducerClass(conf, ColorCountReducer.class);
AvroJob.setOutputSchema(conf,
Pair.getPairSchema(Schema.create(Type.STRING),
                                 Schema.create(Type.INT)));
```

Note that AvroReducer does not implement the Reducer interface. The intermediate Pairs output by the mapper are split into AvroKeys and AvroValues, which are processed by a Reducer that calls the configured AvroReducer's reduce function. AvroJob.setOutputSchema sets up the relevant configuration parameters needed to make this happen, thus you should not need to call JobConf.setReducerClass, JobConf.setOutputFormat, JobConf.setOutputKeyClass, JobConf.setMapOutputKeyClass, JobConf.setMapOutputValueClass, or JobConf.setOutputKeyComparatorClass.

Learning more

It's also possible to implement your own Mappers and Reducers directly using the public classes provided in these libraries. See the AvroWordCount application, found under *examples/mr-example/src/main/java/example/AvroWordCount.java* in the Avro documentation, for an example of implementing a Reducer that outputs Avro data.

Using Avro in MapReduce Jobs With Hadoop

Example data

We are using a small, Twitter-like data set as input for our example MapReduce jobs.

Avro schema

Twitter.avsc defines a basic schema for storing tweets:

```
{
 "type" : "record",
 "name" : "Tweet",
 "namespace" : "com.miguno.avro",
 "fields" : [ {
   "name" : "username",
   "type" : "string",
   "doc"  : "Name of the user account on Twitter.com"
 }, {
   "name" : "tweet",
   "type" : "string",
   "doc"  : "The content of the user's Twitter message"
 }, {
   "name" : "timestamp",
   "type" : "long",
   "doc"  : "Unix epoch time in seconds"
 } ],
 "doc:" : "A basic schema for storing Twitter messages"
}
```

Avro data files

The actual data is stored in the following files:

- twitter.avro – encoded (serialized) version of the example data in binary Avro format, compressed with Snappy
- twitter.json – JSON representation of the same example data

Here is a snippet of the twitter.avro:

{"username":"miguno","tweet":"Rock: Nerf paper, scissors is fine.","timestamp": 1366150681 }
{"username":"BlizzardCS","tweet":"Works as intended. Terran is IMBA.","timestamp": 1366154481 }
{"username":"DarkTemplar","tweet":"From the shadows I come!","timestamp": 1366154681 }
{"username":"VoidRay","tweet":"Prismatic core online!","timestamp": 1366160000 }

Preparing the input data

The example input data we are using is twitter.avro. Upload twitter.avro to HDFS to make the input data available to our MapReduce jobs.

Go to bin directory of hadoop and start hadoop job if it is not started by running ./start-all.sh script.Then type the below commands,

```
# Upload the input data
$ hadoop fs -mkdir examples/input
$ hadoop fs -copyFromLocal src/test/resources/avro/twitter.avro examples/input
```

Examples

TweetCount

TweetCount implements a MapReduce job that counts the number of tweets created by Twitter users.

TweetCount: Usage: TweetCount <input path> <output path>

TweetCountTest

TweetCountTest is very similar to TweetCount. It uses twitter.avro as its input and runs a unit test on it with the same MapReduce job as TweetCount. The unit test includes comparing the actual MapReduce output (in Snappy-compressed Avro format) with expected output..

Hadoop Streaming

1. Download hadoop-streaming-2.0.0-mr1-cdh4.3.0.jar from https://repository.cloudera.com/artifactory/cloudera-repos/org/apache/hadoop/hadoop-streaming/2.0.0-mr1-cdh4.3.0/hadoop-streaming-2.0.0-mr1-cdh4.3.0.jar
2. Download avro-1.7.4.jar from www.eu.apache.org/dist/avro/avro-1.7.4/java/avro-1.7.4.jar
3. Download avro-mapred-1.7.4-hadoop1.jar from www.eu.apache.org/dist/avro/avro-1.7.4/java/avro-mapred-1.7.4-hadoop1.jar

Reading Avro, writing plain-text

The following command reads Avro data from the relative HDFS directory examples/input/ (which normally resolves to /user/<your-unix-username>/examples/input/). It writes the deserialized version of each data record as is to the output HDFS directory streaming/output/. For this simple demonstration we are using the IdentityMapper as a naive map step implementation – it outputs its input data unmodified (equivalently we coud use the Unix tool cat, here) . We do not need to run a reduce phase here, which is why we disable the reduce step via the option -D mapred.reduce.tasks=0

1) Download snappy-java-1.0.3-rc2.jar from http://mvnrepository.com/artifact/org.xerial.snappy/snappy-java/1.0.3-rc2 and put it in the lib directory of hadoop(/usr/local/hadoop/lib)

2) Go to bin directory of hadoop and type below command,

```
# Run the streaming job
$ hadoop jar hadoop-streaming-2.0.0-mr1-cdh4.3.0.jar \
    -D mapred.job.name="avro-streaming" \
    -D mapred.reduce.tasks=0 \
    -files avro-1.7.4.jar,avro-mapred-1.7.4-hadoop1.jar \
    -libjars avro-1.7.4.jar,avro-mapred-1.7.4-hadoop1.jar \
    -input  examples/input/ \
    -output streaming/output/ \
    -mapper org.apache.hadoop.mapred.lib.IdentityMapper \
    -inputformat org.apache.avro.mapred.AvroAsTextInputFormat
```

P.S : Give proper path for all the jars which are used in the above command.i.e hadoop-streaming-2.0.0-mr1-cdh4.3.0.jar,avro-1.7.4.jar,avro-mapred-1.7.4-hadoop1.jar.

Once the job completes you can inspect the output data as follows:

```
$ hadoop fs -cat streaming/output/part-00000 | head -4
```

```
{"username": "miguno", "tweet": "Rock: Nerf paper, scissors is fine.", "timestamp": 1366150681}
{"username": "BlizzardCS", "tweet": "Works as intended.  Terran is IMBA.", "timestamp": 1366154481}
{"username": "DarkTemplar", "tweet": "From the shadows I come!", "timestamp": 1366154681}
{"username": "VoidRay", "tweet": "Prismatic core online!", "timestamp": 1366160000}
```

Please be aware that the output data just happens to be JSON. This is because we opted not to modify any of the input data in our MapReduce job. And since the input data to our MapReduce job is deserialized by Avro into JSON, the output turns out to be JSON, too. With a different MapReduce job you could of course write the output data in TSV or CSV format, for instance.

Integrating Avro with Apache Pig

Preliminaries

Important: The examples below assume you have access to a running Hadoop cluster.

Examples

Prerequisites

First we must register the required jar files to be able to work with Avro. In this example I am using the jar files shipped with CDH4. If you are not using CDH4 just adapt the paths to match your Hadoop distribution.

```
REGISTER /app/cloudera/parcels/CDH/lib/pig/piggybank.jar
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/avro-*.jar
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/jackson-core-asl-*.jar
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/jackson-mapper-asl-*.jar
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/json-simple-*.jar
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/snappy-java-*.jar
```

Note: If you also want to work with Python UDFs in PiggyBank you must also register the Jython jar file:

```
REGISTER /app/cloudera/parcels/CDH/lib/pig/lib/jython-standalone-*.jar
```

Reading Avro

To read input data in Avro format you must use AvroStorage. The following statements show various ways to load Avro data.

```
-- Easiest case: when the input data contains an embedded Avro schema (our
example input data does).
-- Note that all the files under the directory should have the same schema.
records = LOAD 'examples/input/' USING
org.apache.pig.piggybank.storage.avro.AvroStorage();

--
-- Next commands show how to manually specify the data schema
--

-- Using external schema file (stored on HDFS), relative path
records = LOAD 'examples/input/'
        USING
org.apache.pig.piggybank.storage.avro.AvroStorage('no_schema_check',
            'schema_file', 'examples/schema/twitter.avsc');

-- Using external schema file (stored on HDFS), absolute path
records = LOAD 'examples/input/'
        USING org.apache.pig.piggybank.storage.avro.AvroStorage(
          'no_schema_check',
          'schema_file',
'hdfs:///user/YOURUSERNAME/examples/schema/twitter.avsc');

-- Using external schema file (stored on HDFS), absolute path with explicit HDFS
namespace
records = LOAD 'examples/input/'
        USING org.apache.pig.piggybank.storage.avro.AvroStorage(
          'no_schema_check',
          'schema_file',
'hdfs://namenode01:8020/user/YOURUSERNAME/examples/schema/twitter.av
sc');
```

About "no_schema_check": AvroStorage assumes that all Avro files in sub-directories of an input directory share the same schema, and by default AvroStorage performs a schema check. This process may take some time (seconds) when the input

directory contains many sub-directories and files. You can set the option "no_schema_check" to disable this schema check.

See TestAvroStorage.java(/../pig-0.11.1/contrib/piggybank/java/src/test/java/org/apache/pig/piggybank/test/storage/avro) for further examples.

Analyzing the data with Pig

The records relation is already in perfectly usable format – you do not need to manually define a (Pig) schema as you would usually do via LOAD ... AS (...schema follows...).

```
grunt> DESCRIBE records;
records: {username: chararray,tweet: chararray,timestamp: long}
```

Let us take a first look at the contents of the our input data. Note that the output you will see will vary at each invocation due to how ILLUSTRATE works.

```
grunt> ILLUSTRATE records;
<snip>
-------------------------------------------------------------------------------------
| records     | username:chararray     | tweet:chararray          | timestamp:long |
-------------------------------------------------------------------------------------
|             | DarkTemplar            | I strike from the shadows! | 1366184681 |
-------------------------------------------------------------------------------------
```

Now we can perform interactive analysis of our example data:

```
grunt> first_five_records = LIMIT records 5;
grunt> DUMP first_five_records;   <<< this will trigger a MapReduce job
[...snip...]
(miguno,Rock: Nerf paper, scissors is fine.,1366150681)
(VoidRay,Prismatic core online!,1366160000)
(VoidRay,Fire at will, commander.,1366160010)
(BlizzardCS,Works as intended.  Terran is IMBA.,1366154481)
(DarkTemplar,From the shadows I come!,1366154681)
```

List the (unique) names of users that created tweets:

```
grunt> usernames = DISTINCT (FOREACH records GENERATE username);
grunt> DUMP usernames;          <<< this will trigger a MapReduce job
[...snip...]
(miguno)
(VoidRay)
(Immortal)
(BlizzardCS)
(DarkTemplar)
```

Writing Avro

To write output data in Avro format you must use AvroStorage – just like for reading Avro data.

It is strongly recommended that you do specify an explicit output schema when writing Avro data. If you don't then Pig will try to infer the output Avro schema from the data's Pig schema – and this may result in undesirable schemas due to discrepancies of Pig and Avro data models (or problems of Pig itself).

```
-- Use the same output schema as an existing directory of Avro files (files should have the same schema).
-- This is helpful, for instance, when doing simple processing such as filtering the input data without modifying
-- the resulting data layout.
STORE records INTO 'pig/output/'
    USING org.apache.pig.piggybank.storage.avro.AvroStorage(
        'no_schema_check',
        'data', 'examples/input/');

-- Use the same output schema as an existing Avro file as opposed to a directory of such files
STORE records INTO 'pig/output/'
    USING org.apache.pig.piggybank.storage.avro.AvroStorage(
```

```
        'no_schema_check',
        'data', 'examples/input/twitter.avro');

-- Manually define an Avro schema (here, we rename 'username' to 'user' and 'tweet'
to 'message')
STORE records INTO 'pig/output/'
    USING org.apache.pig.piggybank.storage.avro.AvroStorage(
        '{
            "schema": {
                "type": "record",
                "name": "Tweet",
                "namespace": "com.miguno.avro",
                "fields": [
                    {
                        "name": "user",
                        "type": "string"
                    },
                    {
                        "name": "message",
                        "type": "string"
                    },
                    {
                        "name": "timestamp",
                        "type": "long"
                    }
                ],
                "doc:" : "A slightly modified schema for storing Twitter messages"
            }
        }');
```

If you need to store the data in two or more different ways (e.g. you want to rename fields) you must add the parameter "index" to the AvroStorage arguments. Pig uses this information as a workaround to distinguish schemas specified by different AvroStorage calls until Pig's StoreFunc provides access to Pig's output schema in the backend.

```
STORE records INTO 'pig/output-variant-A/'
```

```
    USING org.apache.pig.piggybank.storage.avro.AvroStorage(
      '{
         "index": 1,
         "schema": { ... }
      }');

STORE records INTO 'pig/output-variant-B/'
    USING org.apache.pig.piggybank.storage.avro.AvroStorage(
      '{
         "index": 2,
         "schema": { ... }
      }');
```

See TestAvroStorage.java(/../pig-0.11.1/contrib/piggybank/java/src/test/java/org/apache/pig/piggybank/test/storage/avro) for further examples.

Enabling compression of Avro output data

To enable compression add the following statements to your Pig script or enter them into the Pig Grunt shell:

```
-- We also enable compression of map output (which should be enabled by default anyways) because some Pig jobs
-- skip the reduce phase;  this ensures that we always generate compressed job output.
SET mapred.compress.map.output true;
SET mapred.output.compress true;
SET mapred.output.compression.codec
org.apache.hadoop.io.compress.SnappyCodec
SET avro.output.codec snappy;
```

To disable compression again in the same Pig script/Pig Grunt shell:

```
SET mapred.output.compress false;
```

```
-- Optionally: disable compression of map output (normally you want to leave this enabled)
SET mapred.compress.map.output false;
```

Further readings on Pig

- TestAvroStorage.java(/../pig-0.11.1/contrib/piggybank/java/src/test/java/org/apache/pig/piggybank/test/storage/avro) – many unit test examples that demonstrate how to use AvroStorage