Apache Avro# 1.8.2 Hadoop MapReduce guide

Table of contents

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. See the Hadoop documentation and the Avro getting started guide for introductions to these projects. This guide uses the old MapReduce API (`org.apache.hadoop.mapred`) and the new MapReduce API (`org.apache.hadoop.mapreduce`).

1. 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>1.8.2</version>
</dependency>
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-mapred</artifactId>
  <version>1.8.2</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>1.8.2</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      <configuration>
```

```
        <sourceDirectory>${project.basedir}/../</sourceDirectory>
<outputDirectory>${project.basedir}/target/generated-sources/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

If you do not configure the *sourceDirectory* and *outputDirectory* properties, the defaults will be used. The *sourceDirectory* property defaults to *src/main/avro*. The *outputDirectory* property defaults to *target/generated-sources*. You can change the paths to match your project layout.

Alternatively, Avro jars can be downloaded directly from the Apache Avro# Releases page. The relevant Avro jars for this guide are *avro-1.8.2.jar* and *avro-mapred-1.8.2.jar*, as well as *avro-tools-1.8.2.jar* for code generation and viewing Avro data files as JSON. In addition, you will need to install Hadoop in order to use MapReduce.

2. Example: ColorCount

Below is a simple example of a MapReduce that uses Avro. There is an example for both the old (*org.apache.hadoop.mapred*) and new (*org.apache.hadoop.mapreduce*) APIs under *examples/mr-example/src/main/java/example/*. *MapredColorCount* is the example for the older mapred API while *MapReduceColorCount* is the example for the newer mapreduce API. Both examples are below, but we will detail the mapred API in our subsequent examples.

MapredColorCount:

```
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 MapredColorCount 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: MapredColorCount <input path> <output
path>");
      return -1;
    }

    JobConf conf = new JobConf(getConf(), MapredColorCount.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.getClassSchema());
    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 MapredColorCount(),
args);
    System.exit(res);
  }
}
```

MapReduceColorCount:

```
package example;

import java.io.IOException;

import org.apache.avro.Schema;
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapred.AvroValue;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyInputFormat;
import org.apache.avro.mapreduce.AvroKeyValueOutputFormat;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import example.avro.User;

public class MapReduceColorCount extends Configured implements Tool {

  public static class ColorCountMapper extends
      Mapper<AvroKey<User>, NullWritable, Text, IntWritable> {

    @Override
    public void map(AvroKey<User> key, NullWritable value, Context context)
        throws IOException, InterruptedException {

      CharSequence color = key.datum().getFavoriteColor();
      if (color == null) {
        color = "none";
      }
      context.write(new Text(color.toString()), new IntWritable(1));
    }
  }
```

```
  public static class ColorCountReducer extends
      Reducer<Text, IntWritable, AvroKey<CharSequence>, AvroValue<Integer>>
{

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

      int sum = 0;
      for (IntWritable value : values) {
        sum += value.get();
      }
      context.write(new AvroKey<CharSequence>(key.toString()), new
AvroValue<Integer>(sum));
    }
  }

  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MapReduceColorCount <input path> <output
path>");
      return -1;
    }

    Job job = new Job(getConf());
    job.setJarByClass(MapReduceColorCount.class);
    job.setJobName("Color Count");

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

    job.setInputFormatClass(AvroKeyInputFormat.class);
    job.setMapperClass(ColorCountMapper.class);
    AvroJob.setInputKeySchema(job, User.getClassSchema());
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputFormatClass(AvroKeyValueOutputFormat.class);
    job.setReducerClass(ColorCountReducer.class);
    AvroJob.setOutputKeySchema(job, Schema.create(Schema.Type.STRING));
    AvroJob.setOutputValueSchema(job, Schema.create(Schema.Type.INT));

    return (job.waitForCompletion(true) ? 0 : 1);
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new MapReduceColorCount(), 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.) This example uses the old MapReduce API. See MapReduceAvroWordCount, found under *doc/examples/mr-example/src/main/java/example/* to see the new MapReduce API example. 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 `Pair`s are serialized to an Avro data file.

2.1. 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 from examples/mr-examples to create an Avro data file, *input/users.avro*, containing 20 `User`s with favorite colors chosen randomly from a list:

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

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.MapredColorCount
-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-1.8.2.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.

3. Mapper - org.apache.hadoop.mapred API

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.getClassSchema());
```

Note that `AvroMapper` does not implement the `Mapper` interface. Under the hood, the specified Avro data files are deserialized into `AvroWrapper`s 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`.

4. Mapper - org.apache.hadoop.mapreduce API

This document will not go into all the differences between the mapred and mapreduce APIs, however will describe the main differences. As you can see, ColorCountMapper is now a subclass of the Hadoop Mapper class and is passed an AvroKey as it's key. Additionally, the AvroJob method calls were slightly changed.

```
  public static class ColorCountMapper extends
      Mapper<AvroKey<User>, NullWritable, Text, IntWritable> {

    @Override
    public void map(AvroKey<User> key, NullWritable value, Context context)
        throws IOException, InterruptedException {

      CharSequence color = key.datum().getFavoriteColor();
      if (color == null) {
        color = "none";
      }
      context.write(new Text(color.toString()), new IntWritable(1));
    }
  }
```

5. Reducer - org.apache.hadoop.mapred API

Analogously to `AvroMapper`, an `AvroReducer` defines a reducer function that takes the key/value types output by an `AvroMapper` (or any mapper that outputs `Pair`s) 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 `Pair`s output by the mapper are split into `AvroKey`s and `AvroValue`s, 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`.

6. Reduce - org.apache.hadoop.mapreduce API

As before we not detail every difference between the APIs. As with the Mapper change ColorCountReducer is now a subclass of Reducer and AvroKey and AvroValue are emitted. Additionally, the AvroJob method calls were slightly changed.

```
  public static class ColorCountReducer extends
      Reducer<Text, IntWritable, AvroKey<CharSequence>, AvroValue<Integer>>
{

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

      int sum = 0;
      for (IntWritable value : values) {
        sum += value.get();
      }
      context.write(new AvroKey<CharSequence>(key.toString()), new
AvroValue<Integer>(sum));
    }
  }
```

7. Learning more

The mapred API allows users to mix Avro `AvroMapper`s and `AvroReducer`s with non-Avro `Mapper`s and `Reducer`s and the mapreduce API allows users input Avro and output non-Avro or vice versa.

The mapred package has API `org.apache.avro.mapred` documentation as does the `org.apache.avro.mapreduce` package. MapReduce API (`org.apache.hadoop.mapreduce`). Similarily to the mapreduce package, it's possible with the mapred API to implement your own `Mapper`s and `Reducer`s 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 the old MapReduce API. See the MapReduceAvroWordCount application, found under *examples/mr-example/src/main/java/example/MapReduceAvroWordCount.java* in the Avro documentation, for an example of implementing a `Reducer` that outputs Avro data using the new MapReduce API.