Apache Avro# 1.7.5 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`).

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>1.7.5</version>
</dependency>
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-mapred</artifactId>
  <version>1.7.5</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>1.7.5</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 Apache Avro# Releases page. The relevant Avro jars for this guide are *avro-1.7.5.jar* and *avro-mapred-1.7.5.jar*, as well as *avro-tools-1.7.5.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. 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 `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 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.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-1.7.5.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 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 `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 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 `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`.

5 Learning more

It's possible to mix `AvroMapper`s and `AvroReducer`s with non-Avro `Mapper`s and `Reducer`s. See the `org.apache.avro.mapred` documentation for more details. There is also a `org.apache.avro.mapreduce` package for use with the new MapReduce API (`org.apache.hadoop.mapreduce`). It's also possible 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.