**Avro File Format**

Avro is a widely used row based storage format.

The easiest way to work with Avro data files in Spark applications is by using the DataFrame API in scala. The spark-avro library includes avro methods in SQLContext for reading and writing Avro files:

**Scala Example with Function:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

// The Avro records are converted to Spark types, filtered, and

// then written back out as Avro records

val df = sqlContext.read.avro("input dir")

df.filter("age > 5").write.avro("output dir")

You can also specify "com.databricks.spark.avro" in the format method:

**Scala Example with Format:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.format("com.databricks.spark.avro").load("input dir")

df.filter("age > 5").write.format("com.databricks.spark.avro").save("output dir")

**Writing Deflate Compressed Records:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

// configuration to use deflate compression

sqlContext.setConf("spark.sql.avro.compression.codec", "deflate")

sqlContext.setConf("spark.sql.avro.deflate.level", "5")

val df = sqlContext.read.avro("input dir")

// writes out compressed Avro records

df.write.avro("output dir")

**Writing Partitioned Data:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

val df = Seq(

(2012, 8, "Batman", 9.8),

(2012, 8, "Hero", 8.7),

(2012, 7, "Robot", 5.5),

(2011, 7, "Git", 2.0)).toDF("year", "month", "title", "rating")

df.write.partitionBy("year", "month").avro("output dir")

This code outputs a directory structure like this:

-rw-r--r-- 3 hdfs supergroup 0 2015-11-03 14:58 /tmp/output/\_SUCCESS

drwxr-xr-x - hdfs supergroup 0 2015-11-03 14:58 /tmp/output/year=2011

drwxr-xr-x - hdfs supergroup 0 2015-11-03 14:58 /tmp/output/year=2011/month=7

-rw-r--r-- 3 hdfs supergroup 229 2015-11-03 14:58 /tmp/output/year=2011/month=7/part-r-00001-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro

drwxr-xr-x - hdfs supergroup 0 2015-11-03 14:58 /tmp/output/year=2012

drwxr-xr-x - hdfs supergroup 0 2015-11-03 14:58 /tmp/output/year=2012/month=7

-rw-r--r-- 3 hdfs supergroup 231 2015-11-03 14:58 /tmp/output/year=2012/month=7/part-r-00001-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro

drwxr-xr-x - hdfs supergroup 0 2015-11-03 14:58 /tmp/output/year=2012/month=8

-rw-r--r-- 3 hdfs supergroup 246 2015-11-03 14:58 /tmp/output/year=2012/month=8/part-r-00000-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro

**Reading Partitioned Data:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.avro("input dir")

df.printSchema()

df.filter("year = 2011").collect().foreach(println)

This code automatically detects the partitioned data and joins it all, so it is treated the same as unpartitioned data. This also queries only the directory required, to decrease disk I/O.

root

|-- title: string (nullable = true)

|-- rating: double (nullable = true)

|-- year: integer (nullable = true)

|-- month: integer (nullable = true)

[Git,2.0,2011,7]

**Specifying a Record Name:**

import com.databricks.spark.avro.\_

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.avro("input dir")

val name = "AvroTest"

val namespace = "com.cloudera.spark"

val parameters = Map("recordName" -> name, "recordNamespace" -> namespace)

df.write.options(parameters).avro("output dir")

**Loading into custom Avro schema**:(only available for spark 2.0 because of SparkSession)

import org.apache.avro.Schema

import org.apache.spark.sql.SparkSession

val schema = new Schema.Parser().parse(new File("user.avsc"))

val spark = SparkSession.builder().master("local").getOrCreate()

spark

.read

.format("com.databricks.spark.avro")

.option("avroSchema", schema.toString)

.load("src/test/resources/episodes.avro").show()

**Avro to Parquet**

val df = sqlContext.load("/home/michael/data.avro", "com.databricks.spark.avro")

df.save("/home/michael/data.parquet", "parquet")

**Json to Parquet**

val df = sqlContext.read.format("json").load("examples/src/main/resources/people.json")

df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

**CSV to Avro**

**Using Map/reduce in Hive:**

To convert csv data to Avro data using Hive we need to follow the steps below:

1. Create a Hive table stored as textfile and specify your csv delimiter also.
2. Load csv file to above table using "load data" command.
3. Create another Hive table using AvroSerDe.
4. Insert data from former table to new Avro Hive table using "insert overwrite" command.

To demonstrate this I will use use the data below (student.csv):

**0,38,91  
0,65,28  
0,78,16  
1,34,96  
1,78,14  
1,11,43**

Now execute below queries in Hive:

--1. Create a Hive table stored as textfile

USE test;

CREATE TABLE csv\_table (

student\_id INT,

subject\_id INT,

marks INT)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

STORED AS TEXTFILE;

--2. Load csv\_table with student.csv data

LOAD DATA LOCAL INPATH "/path/to/student.csv" OVERWRITE INTO TABLE test.csv\_table;

--3. Create another Hive table using AvroSerDe

CREATE TABLE avro\_table

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

TBLPROPERTIES (

'avro.schema.literal'='{

"namespace": "com.rishav.avro",

"name": "student\_marks",

"type": "record",

"fields": [ { "name":"student\_id","type":"int"}, { "name":"subject\_id","type":"int"}, { "name":"marks","type":"int"}]

}');

--4. Load avro\_table with data from csv\_table

INSERT OVERWRITE TABLE avro\_table SELECT student\_id, subject\_id, marks FROM csv\_table;

Now you can get data in Avro format from Hive warehouse folder. To dump this file to local file system use below command:

hadoop fs -cat /path/to/warehouse/test.db/avro\_table/\* > student.avro

If you want to get json data from this avro file you can use avro tools command:

java -jar avro-tools-1.7.5.jar tojson student.avro > student.json

So we can easily convert csv to avro and csv to json also by just writing 4 HQLs.

**Using the Avro data model in Parquet**

This recipe showcases how we can retain the older and flexible Avro schema in our code but still Use the Parquet format during storage. The Spark MR project (yes, the one that has the Parquet tools we saw in the previous recipe) has converters for almost all the popular data formats. These model converters take your format and convert it into Parquet format before causing it to persist.

We’ll use the Avro data model and serialize the data in a Parquet file. The recipe involves the following steps:

1. Create the Avro Model.

2. Generate Avro objects using the sbt avro plugin.

3. Construct the RDD of your generated object (StudentAvro) from Students.csv.

4. Save the RDD[StudentAvro] in a Parquet file.

5. Read the file back for verification.

6. Use Parquet-tools to verify.

**Creation of the Avro model:**

let's code the four fields— id, name, phone, and email—in the schema:

{"namespace": "studentavro.avro",

"type": "record",

"name": "StudentAvro",

"fields": [

{"name": "id", "type": ["string", "null"]},

{"name": "name", "type": ["string", "null"]},

{"name": "phone", "type": ["string", "null"]},

{"name": "email", "type": ["string", "null"]}

]

}

The namespace and name attributes in the JSON translate into our package name and class name in our world, respectively. So, our generated class will have a fully qualified name as studentavro.avro.StudentAvro. The "record" (of the type attribute) is one of the complex types in Avro (http://avro.apache.org/docs/1.7.6/spec.html#schema\_ complex). Let me rephrase this again. A record roughly translates to classes in Java/Scala. It is at the topmost level in the schema hierarchy. A record can have multiple fields encapsulated inside it, and these fields can be primitives (https://avro.apache.org/docs/1.7.7/spec.html#schema\_primitive) or other complex types. The last bit about the type having an array of types is interesting ("type": ["string", null"]). It just means that the field can be more than one type. In Avro terms, it is called a union.

Let’s save this file with an extension of .avsc. I have saved it as student.avsc in the src/main/avro directory.

**Generation of Avro objects using the sbt-avro plugin:**

Generate a class from the schema:

we'll be using an sbt-avro plugin (https://github.com/cavorite/sbt-avro) to generate a Java class from the schema.

1) Let’s add the plugin to project/plugins.sbt:

resolvers += "sbt-plugin-releases" at "http://repo.scala-sbt.org/scalasbt/sbt-plugin-releases"

addSbtPlugin("com.cavorite" % "sbt-avro" % "0.3.2")

2) Add the default settings of the plugin to our build.sbt:

fork := true

seq( sbtavro.SbtAvro.avroSettings : \_\*)

3) Let’s generate the Java class now. We can do this by calling sbt avro:generate(on source root in CMD) and then you can see the generated Java file at target/scala-2.10/src\_managed/main/compiled\_avro/studentavro/avro/StudentAvro.java.





4) Add the below library dependencies:

libraryDependencies ++= Seq(

"org.apache.spark" %% "spark-core" % sparkVersion,

"org.apache.spark" %% "spark-sql" % sparkVersion,

"org.apache.spark" %% "spark-mllib" % sparkVersion,

"org.apache.spark" %% "spark-hive" % sparkVersion,

"org.apache.avro" % "avro" % "1.7.7",

"org.apache.parquet" % "parquet-avro" % "1.8.1",

"com.twitter" %% "chill-avro" % "0.6.0"

)

5) Perform the sbt compile on Source root directory in CMD. to compile the class so that the rest of the project picks up the generated Java file.

**Constructing an RDD of our generated object from Students.csv**

convertCSVToStudents function to generate an RDD of the StudentAvro object. Also, since this isn't a Scala class and the generated Java object comes up with a builder inside it, we use the builder to construct the class fluently, as shown below:

**val** *conf* = **new** SparkConf().setAppName(**"AvroModelToParquet"**).setMaster(**"local[2]"**)  
  
**val** *sc* = **new** SparkContext(*conf*)  
**val** *sqlContext* = **new** SQLContext(*sc*)  
*sqlContext*.setConf(**"spark.sql.parquet.binaryAsString"**, **"true"**)  
  
  
*//passing csv columns to avro schema in StudentAvro.java***val** *rddOfStudents* = *convertCSVToStudents*(**"StudentData.csv"**, *sc*)  
**def** convertCSVToStudents(filePath: String, sc: SparkContext): RDD[StudentAvro] = {  
 **val** rddOfStudents: RDD[StudentAvro]=sc.textFile(filePath).flatMap(eachLine => {  
 **val** data = eachLine.split(**"\\|"**)  
 **if** (data(0) == **"id"**) None  
 **else** *Some*(StudentAvro.*newBuilder*()  
 .setId(data(0))  
 .setName(data(1))  
 .setPhone(data(2))  
 .setEmail(data(3)).build())  
 })  
 rddOfStudents  
}

**Saving RDD[StudentAvro] in a Parquet file**

In this step backwards. We fall back to RDD[StudentAvro] in this example instead of a DataFrame because DataFrames can be constructed only from an RDD of case classes (or classes that extend Product, as we saw earlier in this chapter) or from RDD[org.apache.spark.sql.Row]. If you prefer to use DataFrames, you can read the CSV as an array of values, and use RowFactory.create for each array of values. Once an RDD[Row] is available, we can use sqlContext.createDataFrame to convert it to a DataFrame

1) Save the RDD as a Hadoop SequenceFile, we can use saveAsNewAPIHadoopFile. A sequence file is simply a text file that holds key-value pairs. To represent a pair (key-value) in Spark, we use PairRDD and saveAsNewAPIHadoopFile is available only for PairRDDs.

2) Spark uses Java serialization by default to serialize the RDD to be distributed across the cluster. However, the Avro model doesn't implement the serializable interface, and hence it won't be able to leverage Java serialization. That's no reason for worry, however, because Spark provides another 10x performant serialization mechanism called Kryo. The only downside is that we need to explicitly register our serialization candidates

*//can be saved has a json file  
rddOfStudents*.saveAsTextFile(**"deleteDir"**)  
  
  
  
*//saving the rdd as a seqeuence file***val** *pairRddOfStudentsWithNullKey* = *rddOfStudents*.map(each => (**null**, each))  
*conf*.set(**"spark.kryo.registrator"**, *classOf*[StudentAvroRegistrator].getName)  
*conf*.set(**"spark.serializer"**, **"org.apache.spark.serializer.KryoSerializer"**)  
**val** *job* = **new** Job()  
ParquetOutputFormat.*setWriteSupportClass*(*job*, *classOf*[AvroWriteSupport[StudentAvro]])  
AvroParquetOutputFormat.*setSchema*(*job*, StudentAvro.*getClassSchema*)  
*pairRddOfStudentsWithNullKey*.saveAsNewAPIHadoopFile(**"studentAvroPq"**,  
 *classOf*[Void],  
 *classOf*[StudentAvro],  
 *classOf*[AvroParquetOutputFormat[StudentAvro]],  
 *job*.getConfiguration())

3) Using the "spark.serializer" configuration that we intend to use KryoSerializer, and that our registrator is StudentAvroRegistrator. As you may expect, what the Registrator does is register our StudentAvro class as a candidate for Kryo serialization. The twitter-chill project (https://github.com/twitter/chill) provides a nice extension to delegate the Kryo serializer to use the Avro serialization. Keep the below code in a separate class on the same ParquetAvroSchemaMain object.

**class** StudentAvroRegistrator **extends** KryoRegistrator {  
 **override def** registerClasses(kryo: Kryo) {  
 kryo.register(*classOf*[StudentAvro], AvroSerializer.*SpecificRecordBinarySerializer*[StudentAvro])  
 }  
}

**Reading the file back for verification**

We are using newAPIHadoopFile instead of the previously used the parquetFile method. This is because we are reading from a Hadoop sequence file

*//Reading the file b ack for verification*ParquetInputFormat.*setReadSupportClass*(*job*, *classOf*[AvroWriteSupport[StudentAvro]])  
**val** *readStudentsPair* = *sc*.newAPIHadoopFile(**"studentAvroPq"**, *classOf*[AvroParquetInputFormat[StudentAvro]], *classOf*[Void], *classOf*[StudentAvro], *job*.getConfiguration())  
**val** *justStudentRDD*: RDD[StudentAvro] = *readStudentsPair*.map(\_.\_2)  
**val** studentsAsString = *justStudentRDD*.collect().take(5).mkString(**"\n"**)  
*println*(studentsAsString)  
  
*//or can also use***val** *apqDFrame* = *sqlContext*.read.parquet(**"studentAvroPq/\*.parquet"**)  
*apqDFrame*.printSchema()  
*apqDFrame*.show()

**Text to Compressed file**

**To write all output files in compressed files format:**

Use, when you initialize your SparkConf:

conf.set("spark.hadoop.mapred.output.compress", "true")

conf.set("spark.hadoop.mapred.output.compression.codec", "true")

conf.set("spark.hadoop.mapred.output.compression.codec", "org.apache.hadoop.io.compress.GzipCodec")

conf.set("spark.hadoop.mapred.output.compression.type", "BLOCK")

**To compress individual files:**

import org.apache.hadoop.io.compress.GzipCodec

RDD\_Name.saveAsTextFile("/user/sg952655/ReducePCCArray/", classOf[GzipCodec])

Or can use

df\_rdd.saveAsTextFile(filename,compressionCodecClass="org.apache.hadoop.io.compress.GzipCodec")