Apache Avro Data Source Guide

Since Spark 2.4 release, Spark SQL provides built-in support for reading and writing Apache Avro data.

Deploying

The `spark-avro` module is external and not included in `spark-submit` or `spark-shell` by default.

As with any Spark applications, `spark-submit` is used to launch your application. `spark-avro_2.12` and its dependencies can be directly added to `spark-submit` using `--packages`, such as,

```
./bin/spark-submit --packages org.apache.spark:spark-avro_2.12:2.4.1 ...
```

For experimenting on `spark-shell`, you can also use `--packages` to add `org.apache.spark:spark-avro_2.12` and its dependencies directly,

```
./bin/spark-shell --packages org.apache.spark:spark-avro_2.12:2.4.1 ...
```

See Application Submission Guide for more details about submitting applications with external dependencies.

Load and Save Functions

Since `spark-avro` module is external, there is no `.avro` API in `DataFrameReader` or `DataFrameWriter`.

To load/save data in Avro format, you need to specify the data source option `format` as `avro`(or `org.apache.spark.sql.avro`).

Scala Java **Python** R

```
df = spark.read.format("avro").load("examples/src/main/resources/users.avro")
df.select("name", "favorite_color").write.format("avro").save("namesAndFavColors.avro")
```

to_avro() and from_avro()

The Avro package provides function `to_avro` to encode a column as binary in Avro format, and `from_avro()` to decode Avro binary data into a column. Both functions transform one column to another column, and the input/output SQL data type can be complex type or primitive type.

Using Avro record as columns are useful when reading from or writing to a streaming source like Kafka. Each Kafka key-value record will be augmented with some metadata, such as the ingestion timestamp into Kafka, the offset in Kafka, etc.

- If the "value" field that contains your data is in Avro, you could use `from_avro()` to extract your data, enrich it, clean it, and then push it downstream to Kafka again or write it out to a file.
- `to_avro()` can be used to turn structs into Avro records. This method is particularly useful when you would like to re-encode multiple columns into a single one when writing data out to Kafka.

Both functions are currently only available in Scala and Java.

Scala Java

```
import org.apache.spark.sql.avro._

// `from_avro` requires Avro schema in JSON string format.
val jsonFormatSchema = new String(Files.readAllBytes(Paths.get("./examples/src/main/resources/u
ser.avsc")))

val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "topic1")
  .load()

// 1. Decode the Avro data into a struct;
// 2. Filter by column `favorite_color`;
// 3. Encode the column `name` in Avro format.
val output = df
  .select(from_avro('value, jsonFormatSchema) as 'user)
  .where("user.favorite_color == \"red\"")
  .select(to_avro($"user.name") as 'value)

val query = output
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "topic2")
  .start()
```

Data Source Option

Data source options of Avro can be set using the `.option` method on `DataFrameReader` or `DataFrameWriter`.

Property Name	Default	Meaning	Scope
`avroSchema`	None	Optional Avro schema provided by an user in JSON format. The date type and naming of record fields should match the input Avro data or Catalyst data, otherwise the read/write action will fail.	read and write
`recordName`	topLevelRecord	Top level record name in write result, which is required in Avro spec.	write
`recordNamespace`	""	Record namespace in write result.	write
`ignoreExtension`	true	The option controls ignoring of files without `.avro` extensions in read. If the option is enabled, all files (with and without `.avro` extension) are loaded.	read
`compression`	snappy	The `compression` option allows to specify a compression codec used in write. Currently supported codecs are `uncompressed`, `snappy`, `deflate`, `bzip2` and `xz`. If the option is not set, the configuration `spark.sql.avro.compression.codec` config is taken into account.	write

Configuration

Configuration of Avro can be done using the `setConf` method on SparkSession or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning
spark.sql.legacy.replaceDatabricksSparkAvro.enabled	true	If it is set to true, the data source provider `com.databricks.spark.avro` is mapped to the built-in but external Avro data source module for backward compatibility.
spark.sql.avro.compression.codec	snappy	Compression codec used in writing of AVRO files. Supported codecs: uncompressed, deflate, snappy, bzip2 and xz. Default codec is snappy.
spark.sql.avro.deflate.level	-1	Compression level for the deflate codec used in writing of AVRO files. Valid value must be in the range of from 1 to 9 inclusive or -1. The default value is -1 which corresponds to 6 level in the current implementation.

Compatibility with Databricks spark-avro

This Avro data source module is originally from and compatible with Databricks's open source repository spark-avro.

By default with the SQL configuration `spark.sql.legacy.replaceDatabricksSparkAvro.enabled` enabled, the data source provider `com.databricks.spark.avro` is mapped to this built-in Avro module. For the Spark tables created with `Provider` property as `com.databricks.spark.avro` in catalog meta store, the mapping is essential to load these tables if you are using this built-in Avro module.

Note in Databricks's spark-avro, implicit classes `AvroDataFrameWriter` and `AvroDataFrameReader` were created for shortcut function `.avro()`. In this built-in but external module, both implicit classes are removed. Please use `.format("avro")` in `DataFrameWriter` or `DataFrameReader` instead, which should be clean and good enough.

If you prefer using your own build of `spark-avro` jar file, you can simply disable the configuration `spark.sql.legacy.replaceDatabricksSparkAvro.enabled`, and use the option `--jars` on deploying your applications. Read the Advanced Dependency Management section in Application Submission Guide for more details.

Supported types for Avro -> Spark SQL conversion

Currently Spark supports reading all primitive types and complex types under records of Avro.

Avro type	Spark SQL type
boolean	BooleanType
int	IntegerType
long	LongType
float	FloatType
double	DoubleType
string	StringType
enum	StringType
fixed	BinaryType
bytes	BinaryType
record	StructType
array	ArrayType
map	MapType
union	See below

In addition to the types listed above, it supports reading `union` types. The following three types are considered basic `union` types:

1. `union(int, long)` will be mapped to LongType.
2. `union(float, double)` will be mapped to DoubleType.
3. `union(something, null)`, where something is any supported Avro type. This will be mapped to the same Spark SQL type as that of something, with nullable set to true. All other union types are considered complex. They will be mapped to StructType where field names are member0, member1, etc., in accordance with members of the union. This is consistent with the behavior when converting between Avro and Parquet.

It also supports reading the following Avro logical types:

Avro logical type	**Avro type**	**Spark SQL type**
date	int	DateType
timestamp-millis	long	TimestampType
timestamp-micros	long	TimestampType
decimal	fixed	DecimalType
decimal	bytes	DecimalType

At the moment, it ignores docs, aliases and other properties present in the Avro file.

Supported types for Spark SQL -> Avro conversion

Spark supports writing of all Spark SQL types into Avro. For most types, the mapping from Spark types to Avro types is straightforward (e.g. IntegerType gets converted to int); however, there are a few special cases which are listed below:

Spark SQL type	**Avro type**	**Avro logical type**
ByteType	int	
ShortType	int	
BinaryType	bytes	
DateType	int	date
TimestampType	long	timestamp-micros
DecimalType	fixed	decimal

You can also specify the whole output Avro schema with the option `avroSchema`, so that Spark SQL types can be converted into other Avro types. The following conversions are not applied by default and require user specified Avro schema:

Spark SQL type	**Avro type**	**Avro logical type**
BinaryType	fixed	
StringType	enum	
TimestampType	long	timestamp-millis
DecimalType	bytes	decimal