[Design] Built-in AVRO Data Source In Spark 2.4

Gengliang Wang

July 11, 2018

1 Overview

Apache Avro (https://avro.apache.org) is a popular data serialization format. It is widely used in the Spark and Hadoop ecosystem, especially for Kafka-based data pipelines. Using the external package, Spark SQL can read and write the avro data. Making spark-Avro built-in can provide a better experience for first-time users of Spark SQL and structured streaming. We expect the built-in Avro data source can further improve the adoption of structured streaming. The proposal is to inline code from spark-avro package (https://github.com/databricks/spark-avro). The target release is Spark 2.4.

In terms of actual implementations, the plan is to create a pull request that does the minimal cleanup, and then leverage open source contributors to implement more functionalities once the initial code is merged.

2 User-facing APIs and Options

The user-facing APIs for Avro data source should be pretty straightforward to implement. In DataFrameReader/DataFrameWriter, users can directly use **.format("avro")** to specify the file format as Avro. We will also provide implicit avro function, e.g. , *AvroDataFrameWriter(df).avro(...)*

3 Parsing Options

We should support the following options:

- **avroSchema** user specified avro schema

```
val avroSchema = scala.io.Source.fromFile("test.avsc").mkString
spark.read.option("avroSchema", avroSchema).avro("test.avro")
```

- **avro.mapred.ignore.inputs.without.extension** when set to true files without ending ".avro" naming are ignored in read path. Default value is true.
- **recordName** Top level record name in write result, which is required in Avro spec . Default value is "topLevelRecord".
- **recordNamespace** record namespace in write result. Default value is "". See Avro spec for details.
- **compression** compression codec to use when saving to file,which can be "uncompressed"/"snappy"/"deflate". Default value is "snappy".
- **deflateLevel** compression level if compression codec is "deflate" in write path. The default level is -1.

4 Credits

Aaron Davidson, Amanj Sherwany, Arun Allamsetty, Cheng Lian, David Bieber, Gengliang Wang, Guillaume Simard, Hao Xia, Imran Rashid, JD , Jacky Shen, Jacob Salomonsen, James Aley, Jan Prach, Jon Bender, Joseph Batchik, Josh Rosen, Ken Sedgwick, Kevin Duraj, Koert Kuipers, Liang-Chi Hsieh, Mark Grover, Michael Armbrust, Nihed MBAREK, OopsOutOfMemory, Patrick Wendell, Phil Wills, Reynold Xin, Sean Zhong, Silvio Fiorito, Steven Aerts, Todd Gibson, Viacheslav Rodionov, Volodymyr Lyubinets, Xiangrui Meng, Yitong Zhou, gsolasab, leahmcguire

5 TODOs after the initial import PR

1. Add a new function **from_avro** for parsing a binary column of avro format and converting it into its corresponding catalyst value. Add a new function **to_avro** for converting a column into binary of avro format with the specified schema.
2. Upgrade avro version from 1.7.7 to 1.8: support reading logical types - Decimal, Timestamp with different precisions, Time with different precisions, Duration.
3. Create the related SQLConf with documentation, e.g, spark.sql.avro.compression.codec and spark.sql.avro.deflate.level

4. Improve unit tests
    (a) Use Spark test utilities
    (b) Refactor the read/write benchmark.