**Introduction**

Managing large datasets can be difficult due to the issues they present. One frequent problem is moving data around.  For that, the data needs to be in a format that compresses well and at the same time to be fast enough while processing.

The data should also be searchable, and in a format that maintains the relationships between records and fields.

There are many formats to choose from. Comma-separated values (CSV) is a tried-and-true format that works with any language or platform, but it lacks a dictionary or data types, so you’ll need to write your own. Even then, CSV is usually unwieldy with large data sets.

JSON addresses many of CSV’s shortcomings, but it’s not designed for large-scale data storage. The files are large and searching them is still slow.

Fortunately, there are data formats designed for large data sets. Two of the most popular formats are **Apache’s Avro and Parquet**. Both were created as part of the Hadoop project, but they handle data in very different ways. Let’s look at the similarities and tradeoffs.

To perform the Spark Scala code covered in this tutorial you can create a Zeppelin note and use **spark** interpreter (**%spark**).

%spark

sc.version

**Avro Format**

* **What Is Avro**

[Apache Avro](https://avro.apache.org/) is a data serialization system. It’s an open-source data format for serializing and exchanging data. Even though it’s part of the Hadoop project, you can use it in any application, with or without Hadoop libraries. Avro’s serialization works for both data files and messages.

Avro is row-based, so it stores all the fields for each record together. This makes it the best choice for situations where all the fields for a record need to be accessed together.

Avro stores data in a binary format and data definitions in a JSON dictionary. It puts both the data and the schema together in a single file or message, so everything a program needs to process the data is in one place. Unlike similar systems like Protocol Buffers, Avro clients don’t need generated code to read messages. This makes Avro an excellent choice for scripted languages.

One of Avro’s key benefits is robust support for schema changes via a mechanism called schema evolution. This feature gives Avro the ability to gracefully handle missing, new, and altered fields, making Avro data both backward and forward compatible.

When Avro decodes data, it can use two different schemas. One comes from the encoded data and reflects what the publisher encoded, and the other is from the reader and indicates what they expect. Avro will work out the differences, so the reader has a usable result.

Avro supports primitive (boolean, int, long, and string), complex (enumerations, maps, arrays, and user-defined records), and nested data types.

The data format has APIs for Java, Python, Ruby, C, C++, Perl, and PHP. Data serialized by one language can be used by another, and Avro’s C interface means it’s callable from many other languages.

The following figure shows the structure of an AVRO container.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/08/apache-parquet-apache-avro-adatform-tum-png-favpng-jnW7qFbxwZTdMZNxdhUeNpAdw.png)

* **What is Avro Used For?**

The two primary uses for Avro are data **Serialization** and **Remote Procedure Calls** (RPC).

Avro’s speed, flexibility, and compact size make it a popular choice for storing data and for transmitting it over messaging systems like Kafka. In both cases, programs can encode and decode data quickly, while easily managing versioning differences. It’s therefore easier to deploy new code versions with Avro than with primitive formats like CSV or JSON.  You can also use Avro with RPC services like [gRPC](https://grpc.io/" \t "_blank), and use its powerful versioning system with remote procedures.

* **Avro Schema**

Avro purely relies on a schema. When Avro data is read, the schema that is used for writing it is always present. This permits each datum to be written with no per-value overheads, making the serialization small and faster. This method also facilitates using it with dynamic scripting languages, since the data is together with its schema, it is fully self-describing.

When Avro data is stored in a file, its schema is also stored with it, so that files may be processed later by any application. If the application reading the data expects a different schema this can be easily resolved, since both schema’s are present.

When Avro is used in RPC, the client and server exchange schemas in the connection handshake. (This can be optimized so that, for most calls, no schema’s are transmitted.) Since both client and server have the other’s full schema, correspondence between the same-named fields, missing fields, extra fields, etc. can all be easily resolved.

Avro schemas are defined with JSON. This facilitates implementation in languages that already have JSON libraries. Using Avro, we can convert unstructured and semi-structured data into properly structured data using its schema.

You can say that Avro format is actually a combination of a JSON data structure and a schema for validation purposes. So, before we create an Avro file which usually has an extension (*.avro)*, we will be creating its schema having usually the .avsc extension.

{"namespace": "me.adnansiddiqi",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "age", "type": ["int", "null"]},

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

]

}

The JSON structure above represents an AVRO schema structure.

* We will be mentioning a **namespace** first. It is nothing but a string. Usually, it follows the same format which is used by the Java packaging naming convention that is the reverse of your domain name but not necessary.
* After that we mention the type of your schema, here it is of **record**type. There are other types too like **enum**,**arrays**etc.
* After that, we mention the name of the schema which is **User**here.
* The next is **field**item which could be one or more. It has mandatory fields like **name**and **type**and optional fields like **doc**and **alias**. The **doc**field is used to document your field while **alias**is used to give the field a name other than the one mentioned in **name**.

Usually, the schema is saved in a file with the extension **.avsc**.

**Parquet Format**

* **What is Parquet?**

[Apache Parquet](https://parquet.apache.org/) is a free and open-source data format for data storage and retrieval. It’s also a product of the Hadoop project, but it differs from Avro in very important ways.

Avro’s language support differs from Parquet. The core Parquet project only releases [Java jars,](https://parquet.apache.org/blog/2022/05/26/1.12.3/) but, C, C++, and python support is available via the [Arrow](https://arrow.apache.org/docs/cpp/parquet.html) project. There is also a [Python library](https://pypi.org/project/parquet/) for reading Parquet files, and you can process them with [Pandas](https://pandas.pydata.org/).

The biggest difference between Avro and Parquet is that Parquet is a**column-oriented** data format, meaning Parquet stores data by column instead of row. This makes Parquet a good choice when you only need to access specific fields. It also makes reading Parquet files very fast in search situations.

This difference also means that Parquet is not a good choice for network messages, since streaming data and column formats don’t work well together. While it’s possible to use a column-oriented format for streaming data, it often eliminates many of the performance benefits.

Parquet’s schema support is like Avro’s. It supports primitive types like boolean, int, long, and string, and offers robust support for complex and user-defined data types. But schema evolution is expensive with column data, since changes require reprocessing entire data sets, instead of record-by-record in row-oriented data.

The following figure shows the structure of an Parquet container.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/08/parquet.png)

* **What Is Parquet Used For?**

Applications that need rapid access to specific fields in a large dataset use **Parquet**. The format works remarkably well for read-intensive applications and low latency data storage and retrieval.

When you want to aggregate a few columns in a large data set, Parquet is your best bet. Writing files in Parquet is more compute-intensive than Avro, but querying is faster.

* **Parquet Schema**

Apache Parquet schema format is actually a JSON structure. As per the Avro format, schema is need when you create and read a parquet file. So before we create a Parquet file which usually has an extension .parquet, we will be creating its schema (usually having the same extension as Avro schema). The following file is a sample Parquet schema:

message emp\_schema {

optional int32 EmpID;

optional binary LName (UTF8);

optional binary FName (UTF8);

optional double salary;

optional int32 age;

}

* **Compression**

Parquet’s columnar storage allows for better compression, as the data in a column is usually more homogeneous. It supports various compression codecs like Snappy, Gzip, and LZO.

Avro, while supporting similar codecs, doesn’t achieve the same level of compression due to its row-based nature.

**Avro And Parquet with Spark**

Parquet and Avro can be used in any Hadoop ecosystems such as Spark, Hive, Impala, Nifi, etc.

When the data is processed with big  data frameworks such as Hadoop or Spark, the cost of storing the data is more if the data is to be stored in HDFS as HDFS has replication factor, minimum 3 copies of each file will be maintained for fault tolerance. So automatically the storage cost will increase, along with storage, processing cost will also increase as the data comes into CPU, Network IO etc., So  to minimize all these costs, Parquet is one of the choice for developers which efficiently stores the data and thereby increasing the performance.

To work on Parquet files with Spark, we do not need to download any external jars files while you need to download specific jars for Avro. Spark by default has provided support for Parquet files.  Converting a file to Avro or Parquet with Spark is very simple. Just save the file as a Avro / Parquet file.

* **Input Data Preparation**

CSV works well for a small datasets. However, what if you need to increase to millions records? What if each record has nested properties? While CSV files are simple and human-readable, they unfortunately do not scale well. As the file size grows, load times become impractical, and reads cannot be optimized. Their primitive nature allows them to be a great option for smaller data sets, as shown above, but very inconvenient for managing larger sets of data. This is where both Parquet and Avro come in.

In this tutorial we will convert the **flights.csv** (a CSV text file) to Avro and Parquet using Spark. The file is provided as a zip archive. You start by extracting the file from the archive and then placing it on HDFS. Once the file is on HDFS, you will load it into a Spark DataFrame and then convert it into Avro and Parquet.

%sh

# unzip the input archive and place the file on HDFS

cd /home/training/Data/flights

unzip flights.zip

hdfs dfs -mkdir -p /tutorials/spark/formats/csv

hdfs dfs -put flights.csv /tutorials/spark/formats/csv

* **Load CSV into Spark DataFrame**

Spark SQL provides spark.read().csv("file\_name") to read a file or directory of files in CSV format into Spark DataFrame, and dataframe.write().csv("path") to write to a CSV file. Function option() can be used to customize the behavior of reading or writing, such as controlling behavior of the header, delimiter character, character set, and so on.

First extract the archive and then place the flights.csv file on HDFS

%sh

# unzip the input archive and place the file on HDFS

​cd /home/training/Data/flights

unzip flights.zip

hdfs dfs -mkdir -p /tutorials/spark/formats/csv

hdfs dfs -put flights.csv /tutorials/spark/formats/csv

Load the input file into a Spark DataFrame

%spark

// Load the input file into a Spark dataframe

val csvDF = spark.read

.option("header",true)

.option("inferSchema",true)

.csv("/tutorials/spark/formats/csv/flights.csv")

.cache

Verify the file has been loaded

%spark

// Verify the file has been loaded

csvDF.printSchema

csvDF.show(5)

* **Convert CSV to Parquet**

Using parquet() function of DataFrameWriter class, we can write Spark DataFrame to the Parquet file. As mentioned earlier Spark doesn’t need any additional packages or libraries to use Parquet as it by default provides with Spark. easy isn’t it? so we don’t have to worry about version and compatibility issues. In this example, we are writing DataFrame to the HDFS directoy “parquet”.

In order to slightly reduce the file size, you can apply [snappy codec](https://en.wikipedia.org/wiki/Snappy_(compression)) compression. Doing this is trivial as it was just a small change to the write function.

%spark

// Save in Parquet Format

csvDF

.write

.mode("overwrite")

.option("compression", "snappy")

.format("parquet")

.save("/tutorials/spark/formats/parquet/")

* **Convert CSV to Avro**

Since Avro library is external to Spark, it doesn’t provide avro() function on DataFrameWriter , hence we should use DataSource “avro” or “org.apache.spark.sql.avro” to write Spark DataFrame to Avro file.

As per Parquet, you can apply snappy codec compression. You can add this option to the write function.

%spark

// Save in Avro Format

csvDF

.write

.mode("overwrite")

.option("compression", "snappy")

.format("avro")

.save("/tutorials/spark/formats/avro")

Now we have the three datasets formats on HDFS, we can easily compare their sizes.

%sh

# Check the size on HDFS

hdfs dfs -du -h /tutorials/spark/formats

[A black text on a white background

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/08/files-formats-and-sizes.jpg)

The file size decreased from 565 MB to 134.9 MB when storing in Parquet.

* **Read Parquet files**

Similar to write, DataFrameReader provides parquet() function (spark.read.parquet) to read the parquet files and creates a Spark DataFrame. In this example snippet, we are reading data from an apache parquet file we have written before. It is quite simple. Just load the data as a Parquet file.

%spark

// Load Parquet files

val parquetDF = spark

.read

//.schema(schema)// I can add schema here

.parquet("/tutorials/spark/formats/parquet/\*")

.cache

println("PARQUET")

println("Count: " + parquetDF.count)

* **Read Avro files**

Similarly avro() function is not provided in Spark DataFrameReader hence, we should use DataSource format as “avro” or “org.apache.spark.sql.avro” and load() is used to read the Avro file.

%spark

// Load Avro files

val avroDF = spark

.read

.format("avro")

//.option("avroSchema", schemaAvro.toString) // I can add schema here

.load("/tutorials/spark/formats/avro/\*")

.cache

println("AVRO")

println("Count: " + avroDF.count)

Stop Spark context to free resources.

%spark

sc.stop

**Summary**

The choice between Avro and Parquet is a pivotal decision that impacts the efficiency, performance, and flexibility of your data workflows. Parquet is best for analytical workloads. By storing data in columns and implementing compression schemes, it optimizes performance and minimizes storage costs. On the other hand, Avro is a row-based format that offers a versatile solution for data interchange and real-time processing. Choosing which big data file format to use depends on understanding your data’s intricacies and your organization’s unique needs.

I this tutorial we covered both file formats and learned how to read and write these formats with Apache Spark.

Open Zeppelin Note

[Avro and Parquet With Spark](http://localhost:19995/#/notebook/2J9GJCC1Y)