Big Data File Formats

What are file formats? What are the common Hadoop file format features? Which format should you be using?

Rahul Bhatia Follow
May 9, 2019 · 7 min read ★

Big Data File Formats

In this blog, I will talk about what file formats actually are, go through some common Hadoop file format features, and give a little advice on which format you should be using. You can also read a few other interesting case studies on how different big data file formats can be handled using Hadoop managed services here.

Why do we need different file formats?

A huge bottleneck for HDFS-enabled applications like MapReduce and Spark is the time it takes to find relevant data in a particular location and the time it takes to write the data back to another location. These issues get complicated with the difficulties managing large datasets, such as evolving schemas, or storage constraints.

When we are processing Big data, the cost required to store such data is more (Hadoop stores data redundantly to achieve fault tolerance). Along with the storage cost, processing the data comes with CPU, Network, IO costs, etc., As the data increases, the cost for processing and storage increases too.

The various Hadoop file formats have evolved as a way to ease these issues across a number of use cases.

Choosing an appropriate file format can have some significant benefits:

1. Faster read times
2. Faster write times
3. Splittable files
4. Schema evolution support
5. Advanced compression support

Some file formats are designed for general use, others are designed for more specific use cases, and some are designed with specific data characteristics in mind. So there really is quite a lot of choice.

AVRO File Format

Avro is a row-based storage format for Hadoop which is widely used as a serialization platform.

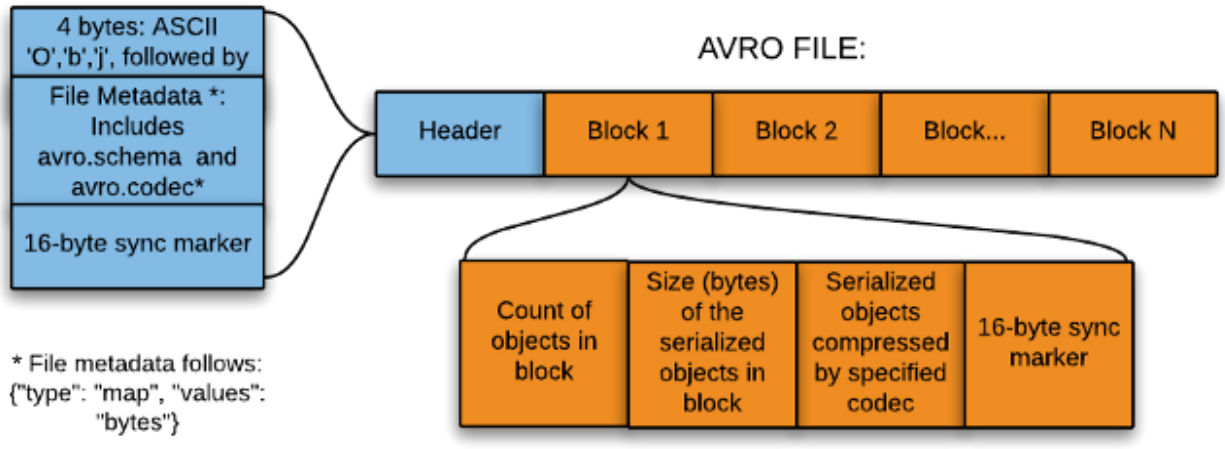
Avro stores the schema in JSON format making it easy to read and interpret by any program.

The data itself is stored in a binary format making it compact and efficient.

Avro is a language-neutral data serialization system. It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).

A key feature of Avro is the robust support for data schemas that changes over time, i.e. schema evolution. Avro handles schema changes like missing fields, added fields and changed fields.

Avro provides rich data structures. For example, you can create a record that contains an array, an enumerated type, and a sub-record.



This format is the ideal candidate for storing data in a data lake landing zone, because:
1. Data from the landing zone is usually read as a whole for further processing by downstream systems (the row-based format is more efficient in this case).
2. Downstream systems can easily retrieve table schemas from files (there is no need to store the schemas separately in an external meta store).
3. Any source schema change is easily handled (schema evolution).

PARQUET File Format

Parquet, an open-source file format for Hadoop stores **nested data structures** in a flat **columnar format**.

Compared to a traditional approach where data is stored in a row-oriented approach, parquet is more efficient in terms of storage and performance.

It is especially good for queries that read particular columns from a "wide" (with many columns) table since only needed columns are read and IO is minimized.

What is a columnar storage format?

In order to understand the Parquet file format in Hadoop better, first, let's see what is a columnar format. In a column-oriented format, the values of each column of the same type in the records are stored together.

For example, if there is a record which comprises of ID, emp Name and Department, then all the values for the ID column will be stored together, values for Name column together and so on. If we take the same record schema as mentioned above having three fields ID (int), NAME (varchar) and Department (varchar), the table will look something like this:

ID	Name	Department
1	emp1	d1
2	emp2	d2

3	emp3	d3

For this table, the data in a row-wise storage format will be stored as follows:

1	emp1	d1	2	emp2	d2	3	emp3	d3

Whereas, the same data in a Column-oriented storage format will look like this:

1	2	3	emp1	emp2	emp3	d1	d2	d3

The columnar storage format is more efficient when you need to query a few columns from a table. It will read only the required columns since they are adjacent thus minimizing IO.

For example, let's say you want only the NAME column. In a row storage format, each record in the dataset has to be loaded, parsed into fields and then the data for Name is extracted. With the column-oriented format, it can directly go to the Name column as all the values for that column are stored together. It doesn't need to go through the whole record.

So, the column-oriented format increases the query performance as less seek time is required to go to the required columns and less IO is required as it needs to read only the columns whose data are required.

One of the unique features of Parquet is that it can store data with nested structures in columnar fashion too. This means that in a Parquet file format, even the nested fields can be read individually without the need to read all the fields in the nested structure. Parquet format uses the record shredding and assembly algorithm for storing nested structures in columnar fashion.

To understand the Parquet file format in Hadoop you should be aware of the following terms-

- **Row group**: A logical horizontal partitioning of the data into rows. A row group consists of a column chunk for each column in the dataset.
- **Column chunk**: A chunk of the data for a particular column. These column chunks live in a particular row group and are guaranteed to be contiguous in the file.
- **Page**: Column chunks are divided up into pages written back to back. The pages share a common header and readers can skip the page they are not interested in.

Here, the Header just contains a magic number “PAR1” (4-byte) that identifies the file as Parquet format file.

Footer contains the following-

- File metadata- The file metadata contains the locations of all the column metadata start locations. Readers are expected to first read the file metadata to find all the column chunks they are interested in. The column chunks should then be read sequentially. It also includes the format version, the schema, and any extra key-value pairs.
- length of file metadata (4-byte)
- magic number “PAR1” (4-byte)

ORC File Format

The *Optimized Row Columnar* (ORC) file format provides a highly efficient way to store data. It was designed to overcome the limitations of other file formats. It ideally stores data compact and enables skipping over irrelevant parts without the need for large, complex, or manually maintained indices. The ORC file format addresses all of these issues.

ORC file format has many advantages such as:

- A single file as the output of each task, which reduces the NameNode's load
- Hive type support including DateTime, decimal, and the complex types (struct, list, map, and union)
- Concurrent reads of the same file using separate RecordReaders
- Ability to split files without scanning for markers
- Estimate an upper bound on heap memory allocation by the Reader/Writer based on the information in the file footer.
- Metadata stored using Protocol Buffers, which allows the addition and removal of fields

ORC stores collections of rows in one file and within the collection the row data is stored in a columnar format.

An ORC file contains groups of row data called **stripes**, along with auxiliary information in a file **footer**. At the end of the file a **postscript** holds compression parameters and the size of the compressed footer.

The default stripe size is **250 MB**. Large stripe sizes enable large, efficient reads from HDFS.

The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum.

Stripe footer contains a directory of stream locations.

Row data is used in table scans.

Index data include min and max values for each column and the row's positions within each column. ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

COMPARISONS BETWEEN DIFFERENT FILE FORMATS

AVRO vs PARQUET

1. AVRO is a row-based storage format whereas PARQUET is a columnar based storage format.
2. PARQUET is much better for analytical querying i.e. reads and querying are much more efficient than writing.
3. Write operations in AVRO are better than in PARQUET.
4. AVRO is much matured than PARQUET when it comes to schema evolution. PARQUET only supports schema append whereas AVRO supports a much-featured schema evolution i.e. adding or modifying columns.
5. PARQUET is ideal for querying a subset of columns in a multi-column table. AVRO is ideal in case of ETL operations where we need to query all the columns.

ORC vs PARQUET

1. PARQUET is more capable of storing nested data.

2. ORC is more capable of Predicate Pushdown.

3. ORC supports ACID properties.

4. ORC is more compression efficient.

Big Data