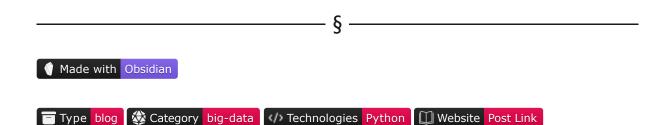
6 Big Data File Formats Compared, Pt. 1



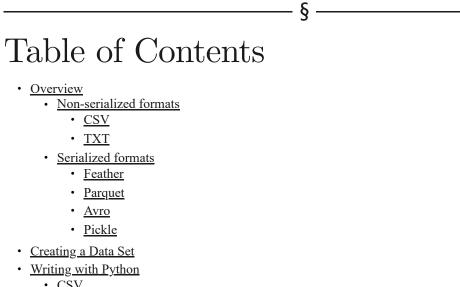
A Big Data file format is designed to store high volumes of variable data optimally. This can be achieved using different formats, such as columnar or row-based.

Columnar formats store data by clustering entries by column, whereas row-based formats store data by clustering entries by row. Both formats are widely used in Big Data and present advantages & disadvantages among each other.

We can also further classify formats as text files or binary files. A binary file is designed to be read by computers; we cannot open a binary file and read its content simply using a text editor. In contrast, a text file can be directly opened with a text editor.

In this 3-article series, we will discuss six popular Big Data file formats, explain what they are for, go over writing & reading examples, and make some performance comparisons.

We'll be using Python scripts which can be found in the Blog Article Repo.



- CSV
 - <u>Using numpy.tofile()</u>
 - <u>Using numpy.savetext()</u>
 - Using pandas.DataFrame.to csv()
- TXT
 - <u>Using numpy.savetext()</u>
 - Using pandas.DataFrame.to csv()

- <u>Feather</u>
 - <u>Using pandas.DataFrame.to_feather()</u>
- Parquet
 - <u>Using pandas.DataFrame.to_parquet() without partitioning</u>
 - <u>Using pandas.DataFrame.to_parquet() with a single partition</u>
 - <u>Using pandas.DataFrame.to_parquet() with multiple partitions</u>
- <u>Avro</u>
 - Using fastavro Python file handler
- <u>Pickle</u>
 - <u>Using .pickle.dump() to write as an open file</u>
 - <u>Using .pickle.dumps() to write as a byte string</u>
- <u>Conclusions</u>
- <u>References</u>
- <u>Copyright</u>

Overview

1. Non-serialized formats

As opposed to the serialized formats, **non-serialized** formats do not convert the object into a stream of bytes. We will explain serialization formats in more detail further on. The most common non-serialization formats are CSV & TXT files.

§

$1.1 \ \mathrm{CSV}$

Comma-separated values (*CSV*) is a delimited text file format that typically uses a comma to separate values, and although more delimiters can be used, it's not standard practice. It is the most popular format for storing & reading tabular data since it's fast, easy to write & read, supported by practically all programs & libraries involving data processing, and forces a flat & simple schema.

As popular as it is, CSV also has some disadvantages, such as large file sizes, slow parsing time, poor support from Apache Spark, missing data handling, limited encoding formats, special handling required with nested data, basic data support only, lack of support of special characters, no defined schema, and the use of commas as delimiters; if our data entries have commas, we will have to enclose the entry in quotes. Otherwise, they will be treated as delimiters.

These disadvantages make CSV files suboptimal when working with big data.

A typical CSV file will have a .csv extension and will look like the example below:

```
Name,Age,Occupation,Country,State,City
Joe,20,Student,United States,Kansas,Kansas City
Chloe,37,Detective,United States,California,Los Angeles
Dan,39,Detective,United States,California,Los Angeles
...
```

Some considerations:

- The header is denoted as the first row of our document.
- Each entry is followed by a comma but without blank spaces.
- Entries can have blank spaces and will be treated accordingly when parsing.
- Even though we can use text and numeric values, a CSV file will not store information regarding data types.

$1.2 \mathrm{TXT}$

Text document file (*TXT*) is a plain-text format structured as a sequence of lines of text. It is also a prevalent format used for storing & reading tabular data because of its simplicity & versatility; a TXT file can be formatted as delimited, free form, fixed width, jagged right, and so on.

A typical TXT file will have a .txt extension and will look like the example below (depending on the delimiter used, it will vary. In this example, we use tab delimiters which is the convention):

```
City
Name
       Age
             Occupation
                          Country
                                    State
Joe
      20
           Student United States Kansas
                                              Kansas Citv
Chloe
             Detective
                         United States
                                        California
                                                      Los Angeles
Dan
      39
          Detective
                      United States
                                       California
                                                    Los Angeles
```

2. Serialized formats

Here is where things get more interesting; we introduce a concept called serialization.

This refers to the process of converting a data object into a series of bytes that save the state of the object in an easily transmittable form. The inverse process, called **deserialization**, consists of reverting the object to its original form.

Serialization has multiple advantages when it comes to Big Data handling & processing:

- It facilitates the transportation of data across networks and avoids compatibility issues.
- It enables us to save the exact current state of the object, transfer it, and replicate it in a new location.
- The serialization process can be customized depending on the specific needs, and the serialized data can also be encrypted.

There are multiple data-serialization formats available. In this section, we'll mention five examples widely used to process data.

2.1 Feather

Feather is a portable, lightweight, columnar, serialized file format based on Apache arrow. It uses the <u>Arrow</u> <u>IPC format</u> internally to store & organize data. Feather supports two compression libraries, the default being LZ4 (*included in the pyarrow library*), and is available for Python & R programming languages.

In general, .feather files are mostly recommended for short-term storage since stability between binary versions is not guaranteed.

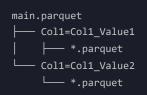
A typical Feather file will have a .feather extension. Since it is a serialized format, we cannot see the contents of a .feather file by simply using a text editor.

2.2 Parquet

Parquet is an open source, columnar, serialized data file format created by Apache, and designed for efficient data storage & retrieval. As with .feather files, it supports data compression and encoding schemes, and is optimized for handling data in bulk.

Unlike .feather files, .parquet files are suited for long-term storage since the binary versions are much more stable and constitute the gold standard for large data set columnar format storage.

A typical Parquet file will have a .parquet extension. Since it is a serialized format, we cannot see the contents of a .parquet file by simply using a text editor. Still, we can see the folder structure (*partition scheme*) created if we use partitioning. A single-partitioned .parquet system would look similar to the example below:



2.3 Avro

Avro is a widely used, row-based, serialized storage format for Hadoop. It uses serialization for the actual data and the JSON format to store the data schema, making it easily readable by other platforms.

The main difference between Avro and Feather & Parquet formats is that Avro uses a row-based structure, whereas the last two use a column-based (*columnar*) format.

A typical Avro file will have a .avro extension. Since it is a serialized format, we cannot see the contents of a .avro file by simply using a text editor.

2.4 Pickle

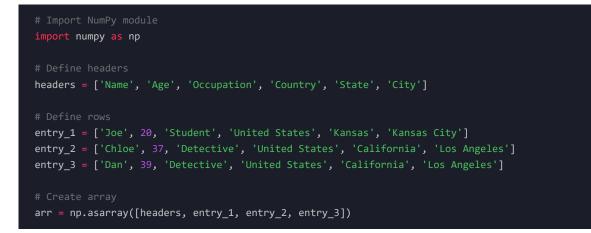
Pickle is a language-specific, serialized file format used to store Python objects. Its usage is sometimes discouraged since it is not a universal file format, and other languages might present difficulties parsing it. On the other hand, it allows us to write virtually any Python object to disk preserving its structure: tuples, lists, arrays, dictionaries, nested objects, class instances & DataFrames, among others, are supported.

A typical Pickle file will have a .pickle extension, though other extensions such as .pck , .pcl , and .db are also supported. Since it is a serialized format, we cannot see the contents of a .pickle file by simply using a text editor.

Creating a Data Set

For this section, we will create an array containing strings & numbers using the NumPy module. Keep in mind that we will be using this same object throughout the entire section:

_____ § ____



Name	Age	Occupation	Country	State	City
Joe	20	Student	United States	Kansas	Kansas City
Chloe	37	Detective	United States	California	Los Angeles
Dan	39	Detective	United States	California	Los Angeles

We will also create an outputs folder, where we'll store all written files:

Code

mkdir outputs

Once we have our data set as a numpy.ndarray object arr and out outputs folder ready, we can proceed with the writing.

§

Writing with Python

1. CSV

There are five primary methods for writing a CSV file using Python, although we'll only be covering three:

1.1 Using numpy.tofile()

This method takes a numpy.ndarray object as input and writes a .csv file in return. Additionally, it is very simple and accepts only two additional parameters:

- fid : An open file object or a string containing a filename.
- sep : Separator between array items for text output. If empty, a binary file is written, equivalent to file.write(a.tobytes()).
- str : Format string for text file output.

This method is not recommended because it lacks flexibility, and even though we generated a two-dimensional array, the output is a single-lined .csv file:

Code

```
# Export data to csv using numpy.tofile() method
arr.tofile('outputs/01_dataset_method_1.csv', sep = ',')
```

OUTPUT

```
'Name', 'Age', 'Occupation', 'Country', 'State', 'City', 'Joe', '20', 'Student',...
```

1.2 Using numpy.savetext()

This method takes a numpy.ndarray object as input and writes a .csv file in return. It accepts a total of eight parameters. We will stick with the most relevant:

- fname : Filename or file handle.
- x : Data to be saved to a text file.
- fmt : A single format (%10.5f), a sequence of formats, or a multi-format string.
- delimiter : String or character separating columns.
- newline : String or character separating lines.
- header : String that will be written at the beginning of the file.

In contrast to the previous method, numpy.savetext() is more versatile and lets us output a multi-line .csv file using the newline parameter.

The catch to this method is being careful in the fmt parameter we specify. If we leave it empty or specify the incorrect format, we'll probably get a TypeError in return.

Also, we need to be careful & remember which newline parameter we use. We'll stick to a newline \n for this example:

Code

```
# Export data to csv using numpy.savetext() method
np.savetxt('outputs/02_dataset_method_2.csv', arr, fmt = '%s', delimiter = ',', newline = '\n')
```

OUTPUT

					- • ·
Name	Age	Occupation	Country	State	City
Joe	20	Student	United States	Kansas	Kansas City
Chloe	37	Detective	United States	California	Los Angeles
Dan	39	Detective	United States	California	Los Angeles

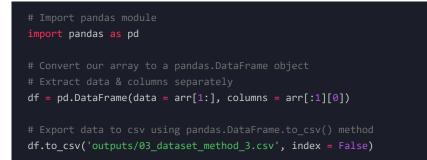
1.3 Using pandas.DataFrame.to_csv()

By far the most common technique when working with tabular data, but we leave it at the end since this method requires a different object as input.

The pandas.DataFrame.to_csv() method accepts a pandas.DataFrame object and writes a .csv file in return. It accepts a total of 21 parameters. We will stick with the most relevant:

- path_or_buf : String or path object.
- sep : String of length 1. Field delimiter for the output file.
- header : Write out the column names.
- index : Write row names (index).

Code



OUTPUT

Name	e Age	Occupation	Country	State	City
Joe	20	Student	United States	Kansas	Kansas City
Chlo	e 37	Detective	United States	California	Los Angeles
Dan	39	Detective	United States	California	Los Angeles

2. TXT

There are two main methods for writing a .txt file using Python, both of which we've already seen:

2.1 Using numpy.savetext()

The syntax is the same as with a .csv file; we will only change some parameters:

- We will change the fname extension.
- We will remove the current delimiter (,) and substitute it with a tab delimiter ('\t'). If we do not specify the delimiter parameter, our .txt file will be written with a single space delimiter. This is bad practice and will probably lead to problems when parsing the file if we have entries with single spaces included.

Everything else can stay as is.

We will use the numpy.ndarray object arr we created in the previous section:

Code

```
# Export data to csv using numpy.savetext() method
np.savetxt('outputs/04_dataset_method_1.txt', arr, fmt = '%s', delimiter = '\t', newline = '\n')
```

OUTPUT

Name	Age	Occupation	Country	State	City
Joe	20	Student	United States	Kansas	Kansas City
Chloe	37	Detective	United States	California	Los Angeles
Dan	39	Detective	United States	California	Los Angeles

2.2 Using pandas.DataFrame.to_csv()

This method, again, is the preferred one since it has a fair amount of parameters we can fine-tune. The syntax is the same as with a .csv file; we will only change some parameters:

• If we omit the sep parameter, our file will be written as comma-separated. As stated before, the convention for .txt files is to use tab delimiters, so we'll change that.

We will use the pandas.DataFrame object df we created in the previous section:

\mathbf{C} ODE

Export data to csv using pandas.DataFrame.to_csv() method
df.to_csv('outputs/05_dataset_method_3.csv', index = False)

OUTPUT

Age	Occupation	Country	State	City
20	Student	United States	Kansas	Kansas City
37	Detective	United States	California	Los Angeles
39	Detective	United States	California	Los Angeles
	20 37	20 Student 37 Detective	20StudentUnited States37DetectiveUnited States	20 Student United States Kansas 37 Detective United States California

3. Feather

We can use the pandas.DataFrame.to_feather() method. To use this method, we will need to install an additional library called pyarrow :

pip install pyarrow

$3.1 \ Using$ pandas.DataFrame.to_feather()

This method accepts a pandas.DataFrame object and writes a feather file in return. It accepts a total of 5 parameters, including the additional kwargs :

- path : String or path object.
- kwargs : Additional keywords passed:
 - compression
 - compression_level
 - chunksize
 - version

We will use the pandas.DataFrame object df we created in the previous section:

```
# Export data to feather using pandas.DataFrame.to_feather() method
df.to_feather('outputs/06_dataset_method_1.feather')
```

Since .feather files are binary, we won't be able to see the actual contents of a .feather file directly using a text editor.

4. Parquet

We can use the pandas.DataFrame.to_parquet() method. Same as with the pandas.DataFrame.to_feather() module, to use this method, we will need to install an additional module called pyarrow.

Since we already have it, we'll go straight to writing.

4.1 Using pandas.DataFrame.to_parquet() without partitioning

This method accepts a pandas.DataFrame object and writes a parquet file in return. It accepts a total of 5 parameters, including the additional kwargs :

- path : String or path object.
- engine : Parquet library to use. If 'auto', the option io.parquet.engine is used.
- compression : Name of the compression to use.
- index : If True, include the DataFrame's index(es) in the file output.
- partition_cols : Column names by which to partition the dataset.
- kwargs : Additional keywords passed.

We will use the pandas.DataFrame object df we created in the previous section.

As mentioned earlier, we can write <code>.parquet</code> files partitioned or without partitioning. If we want to write an unpartitioned <code>.parquet</code> file, we can do so by leaving the <code>partition_cols</code> parameter unspecified:

Code

```
# Using pandas.DataFrame.to_parquet() without partitioning
df.to_parquet('outputs/07_dataset_method_1.parquet')
```

OUTPUT

Since .parquet files are binary, we won't be able to see the actual contents of a .parquet file directly by using a text editor.

4.2 Using pandas.DataFrame.to_parquet() with a single partition

In contrast, if we want to write a partitioned .parquet file, we can specify the column names by which to partition the dataset using the partition_cols parameter. In this example, we will partition only by State :

```
# Using pandas.DataFrame.to_parquet() with partitioning
df.to_parquet('outputs/08_dataset_method_2.parquet', partition_cols = 'State')
```

As before, .parquet files are binary, but we can take a look at the different partitions:



4.3 Using pandas.DataFrame.to_parquet() with multiple partitions

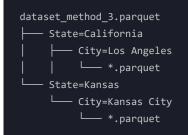
We can also pass a list of columns if we want an output partitioned multiple times. Each partition will be nested inside its parent partition:

Code

Using pandas.DataFrame.to_parquet() with partitioning df.to_parquet('outputs/09_dataset_method_3.parquet', partition_cols = ['State', 'City'])

OUTPUT

As before, .parquet files are binary, but we can take a look at the different partitions:



5. Avro

.avro files are less straightforward when working with pandas since there is no default support. Also, as mentioned earlier, we need to define a schema to write .avro files.

There are two main libraries for manipulating .avro files in Python. We will stick with the latter since the first one is significantly slower:

- avro
- fastavro

5.1 Using _{fastavro} Python file handler

Before anything else, we will need to install the required library:

Code

pip install fastavro

We can then import the required modules from the fastavro library:

Code

Import fastavro modules
from fastavro import writer, parse_schema

We must also cast the Age column of our DataFrame df to int. Otherwise, when defining our schema and writing our file, we will get a TypeError :

Code

```
# Cast age to int type
df['Age'] = df['Age'].astype('int')
# Verify casting
df.dtypes
```

OUTPUT

Name	object
Age	int32
Occupation	object
Country	object
State	object
City	object
dtype: objec	t

The next step will consist of defining our schema in a JSON-like format and then parsing it using the parse_schema() method:

```
# Define the schema
schema = {
    'type': 'record',
    'name': 'dataset',
    'namespace': 'dataset',
    'doc': 'This schema consists of 1 int type and 7 string types',
    'fields': [
        {'name': 'Name', 'type': 'string'},
        {'name': 'Age', 'type': 'int'},
        {'name': 'Occupation', 'type': 'string'},
        {'name': 'Country', 'type': 'string'},
        {'name': 'City', 'type': 'string'}
    ]
# Parse the schema
parsed_schema = parse_schema(schema)
```

- The type parameter is set to record . A record is a complex Avro type that supports the following parameters:
 - The name parameter provides the name of the record .
 - The namespace parameter provides a name qualification.
 - The doc parameter provides documentation to the user of our schema.
 - The fields parameter provides the data types for each column.

The parse_schema() method returns a dictionary consisting of 6 entries:

OUTPUT

fastavro_parsed	bool
named_schemas	dict
doc	str
fields	list
name	str
type	str

- __fastavro_parsed confirms we have a fastavro parsed schema.
- __named_schemas contains doc , fields , name & type .
- doc contains our brief documentation.
- fields contains a dictionary with column data type associations.
- name contains our record name.
- type denotes that this is a record .

Once we have generated our schema, we can then convert our pandas.DataFrame object to a list of records:

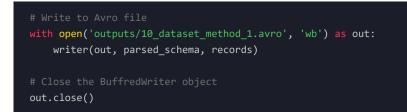
```
# Convert pd.DataFrame to records (list of dictionaries)
records = df.to_dict('records')
```

The df.to_dict() method returns a Python list containing three dictionaries (*one per row*), each with columnentry pairs.

OUTPUT

```
dict {'Name': 'Joe', 'Age': 20, 'Occupation': 'Student'...}
dict {'Name': 'Chloe', 'Age': 37, 'Occupation': 'Detective'...}
dict {'Name': 'Dan', 'Age': 39, 'Occupation': 'Detective'...}
```

Finally, we can write our list of records to a .avro file using the Python file handler:



The wb parameter denotes we're writing a binary file.

6. Pickle

We can use the built-in pickle library for writing .pickle files. This library provides two different serialization methods:

- pickle.dump() : The open file version.
- pickle.dumps() : The byte string version.

6.1 Using .pickle.dump() to write as an open file

We will start by importing the pickle library:

Code

Import pickle library
import pickle

Next, we will create a Python file handle by specifying our target filename.pickle :

```
# Open a file to store the data
file = open('outputs/11_dataset_method_1.pickle', 'wb')
```

Finally, we will use the pickle.dump() method to convert our previously generated list of dictionaries named records, to a pickle open file:

 \mathbf{C} ODE

```
# Write open file to disk
pickle.dump(records, file)
```

.avro files are also binary, although a pickle.dump() object is different in structure from a pickle.dumps() object.

6.2 Using .pickle.dumps() to write as a byte string

This method differs slightly from the previous one. As a first step, we will convert our previously generated list of dictionaries named records, to a pickle byte string:

Code

```
# Define a pickle object
my_pickled_object = pickle.dumps(records)
# Check the data type
type(my_pickled_object)
```

OUTPUT

bytes

We will next write our string of bytes as a .pickle file in memory using the Python file handler:

Code

```
# Write byte string to disk
with open('outputs/12_dataset_method_2.pickle','wb') as out:
    out.write(my_pickled_object)
# Close the BufferedWriter object
out.close()
```

OUTPUT

The end result is a .pickle file containing a line of binary characters, or byte string.

Conclusions

We've reviewed six file formats that can be used to write different data types. Each file format serves a different purpose, from the very simple to the more complex.

- § –

Now that we know some general theories behind serialization, deserialization, the different file formats used, and how to write them using Python, it's time to move on to reading these files and comparing them.

- § -

References

- · Geeks for Geeks, Working with csv files in Python
- Stack Exchange, Why do we keep using CSV?
- Python Documentation, CSV File Reading and Writing
- · Ireland's Open Data Portal, Choosing the right format for open data
- <u>Towards Data Science, Big Data File Formats Explained</u>
- Apache Arrow, Tabular File Formats
- Apache Avro, Schema Record
- Informatica, Avro Data Types
- Berkeley, Python Numerical Methods Pickle Files
- Towards Data Science, CSV Files for Storage? Absolutely Not. Use Apache Avro Instead

– § -

Copyright

Pablo Aguirre, Creative Commons Attribution 4.0 International, All Rights Reserved.