The Best Format to Save Pandas Data

A small comparison of various ways to serialize a pandas data frame to the persistent storage

Ilia Zaitsev Follow
Mar 14, 2019 · 6 min read

When working on data analytical projects, I usually use `Jupyter` notebooks and a great `pandas` library to process and move my data around. It is a very straightforward process for moderate-sized datasets which you can store as plain-text files without too much overhead.

However, when the number of observations in your dataset is high, the process of saving and loading data back into the memory becomes slower, and now each kernel's restart steals your time and forces you to wait until the data reloads. So eventually, the CSV files or any other plain-text formats lose their attractiveness.

We can do better. There are plenty of binary formats to store the data on disk, and many of them `pandas` supports. How can we know which one is better for our purposes? Well, we can try a few of them and compare! That's what I decided to do in this post: go through several methods to save `pandas.DataFrame` onto disk and see which one is better in terms of I/O speed, consumed memory and disk space. In this post, I'm going to show the results of my little benchmark.

Photo by Patrick Lindenberg on Unsplash

Formats to Compare

We're going to consider the following formats to store our data.

- Plain-text CSV — a good old friend of a data scientist
- Pickle — a Python's way to serialize things
- MessagePack — it's like JSON but fast and small
- HDF5 —a file format designed to store and organize large amounts of data
- Feather — a fast, lightweight, and easy-to-use binary file format for storing data frames
- Parquet — an Apache Hadoop's columnar storage format

All of them are very widely used and (except MessagePack maybe) very often encountered when you're doing some data analytical stuff.

Chosen Metrics

Pursuing the goal of finding the best buffer format to store the data between notebook sessions, I chose the following metrics for comparison.

- `size_mb` — the size of the file (in Mb) with the serialized data frame
- `save_time` — an amount of time required to save a data frame onto a disk

- `load_time` — an amount of time needed to load the previously dumped data frame into memory
- `save_ram_delta_mb` — the maximal memory consumption growth during a data frame saving process
- `load_ram_delta_mb` — the maximal memory consumption growth during a data frame loading process

Note that the last two metrics become very important when we use the efficiently compressed binary data formats, like Parquet. They could help us to estimate the amount of RAM required to load the serialized data, *in addition* to the data size itself. We'll talk about this question in more details in the next sections.

The Benchmark

I decided to use a synthetic dataset for my tests to have better control over the serialized data structure and properties. Also, I use two different approaches in my benchmark: (a) keeping generated categorical variables as strings and (b) converting them into `pandas.Categorical` data type before performing any I/O.

The function `generate_dataset` shows how I was generating the datasets in my benchmark.

```
def generate_dataset(n_rows, num_count, cat_count, max_nan=0.1, max_cat_size=100):
    """Randomly generate datasets with numerical and categorical features.

    The numerical features are taken from the normal distribution X ~ N(0, 1).
    The categorical features are generated as random uuid4 strings with
    cardinality C where 2 <= C <= max_cat_size.

    Also, a max_nan proportion of both numerical and categorical features is replaces
    with NaN values.
    """
    dataset, types = {}, {}

    def generate_categories():
        from uuid import uuid4
        category_size = np.random.randint(2, max_cat_size)
        return [str(uuid4()) for _ in range(category_size)]

```

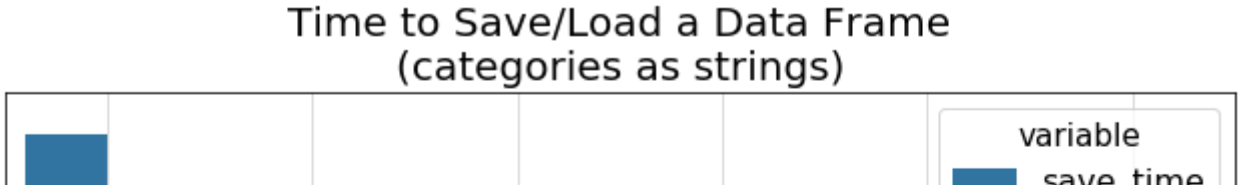
```
    for col in range(num_count):
        name = f'n{col}'
        values = np.random.normal(0, 1, n_rows)
        nan_cnt = np.random.randint(1, int(max_nan*n_rows))
        index = np.random.choice(n_rows, nan_cnt, replace=False)
        values[index] = np.nan
        dataset[name] = values
        types[name] = 'float32'

    for col in range(cat_count):
        name = f'c{col}'
        cats = generate_categories()
        values = np.array(np.random.choice(cats, n_rows, replace=True), dtype=object)
        nan_cnt = np.random.randint(1, int(max_nan*n_rows))
        index = np.random.choice(n_rows, nan_cnt, replace=False)
        values[index] = np.nan
        dataset[name] = values
        types[name] = 'object'

    return pd.DataFrame(dataset), types
```

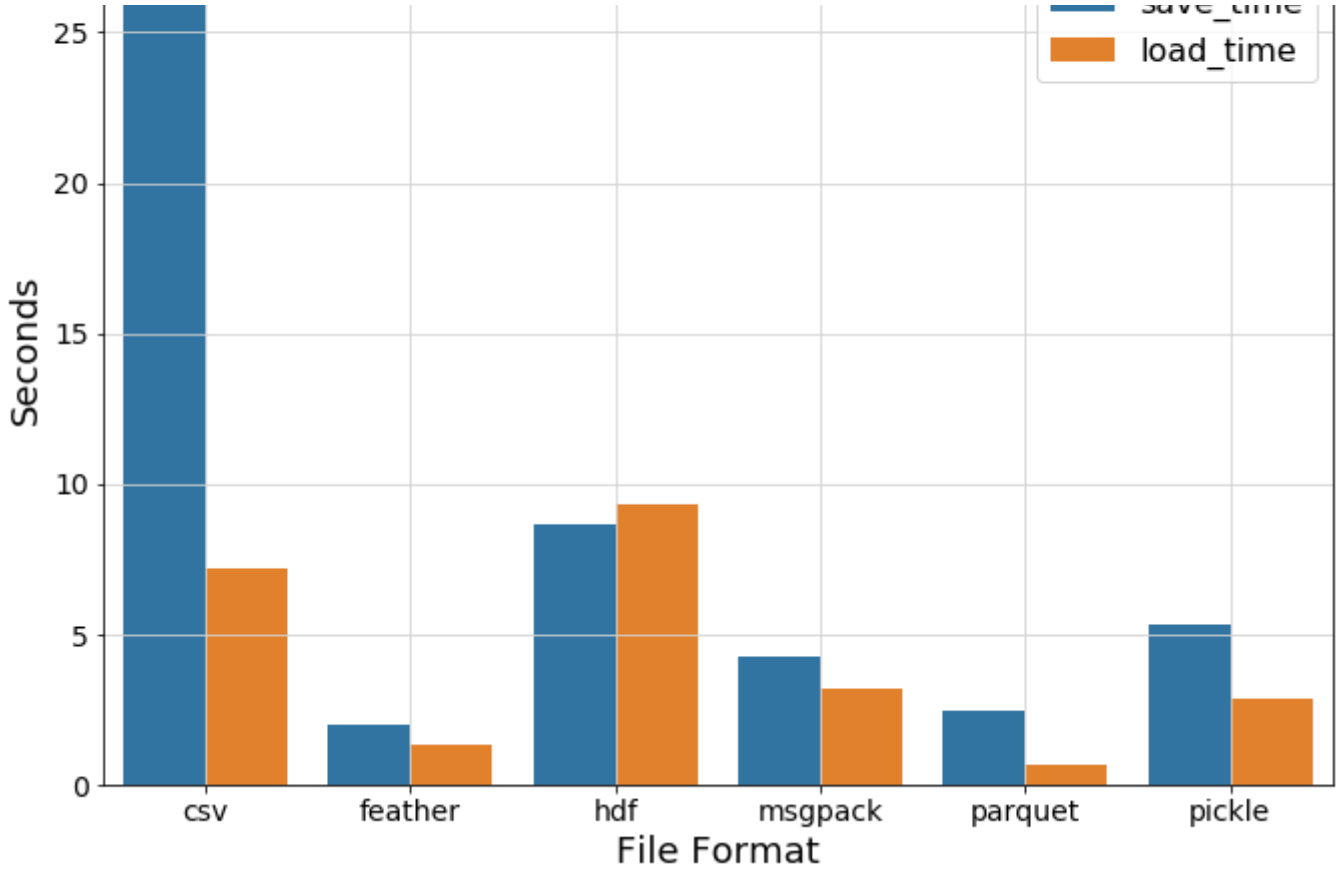
gendata.py hosted with ❤️ by GitHub view raw

The performance of CSV file saving and loading serves as a baseline. The five randomly generated datasets with million observations were dumped into CSV and read back into memory to get mean metrics. Each binary format was tested against 20 randomly generated datasets with the same number of rows. The datasets consist of 15 numerical and 15 categorical features. You can find the full source code with the benchmarking function and required in this repository.

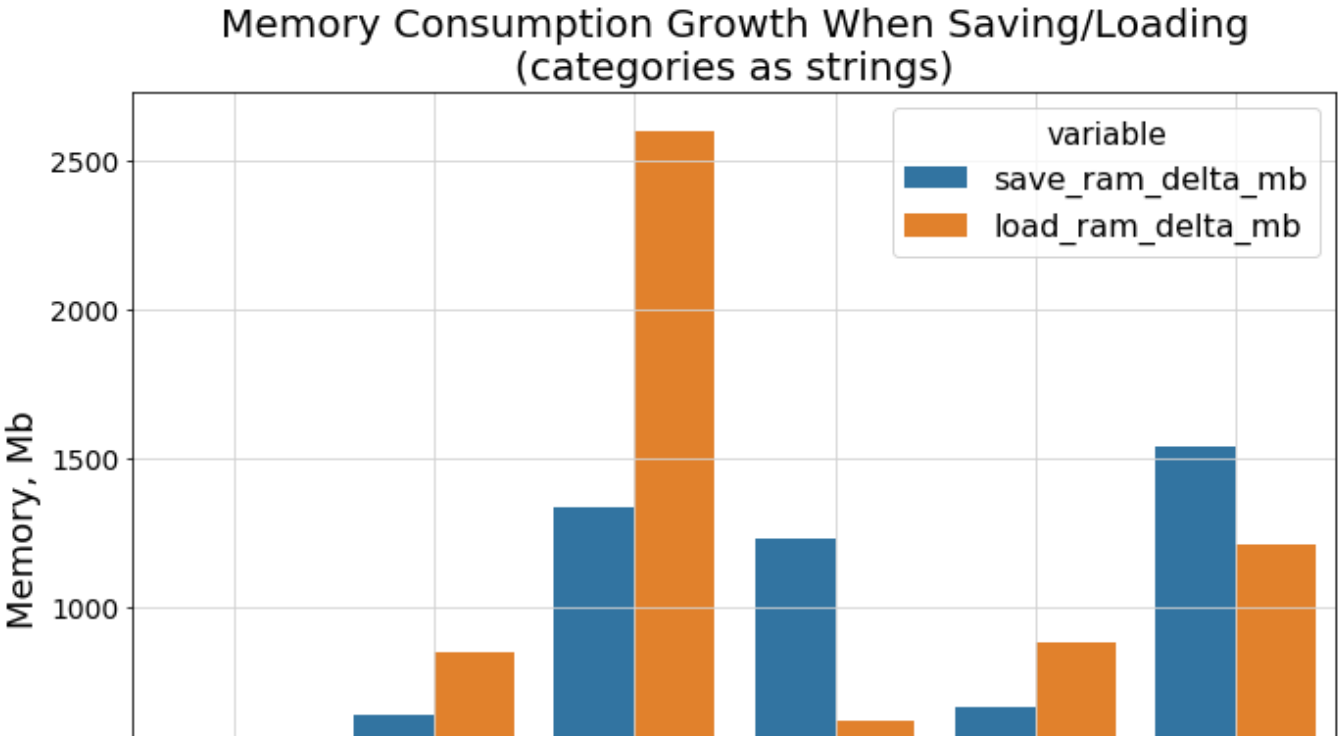
(a) Categorical Features as Strings

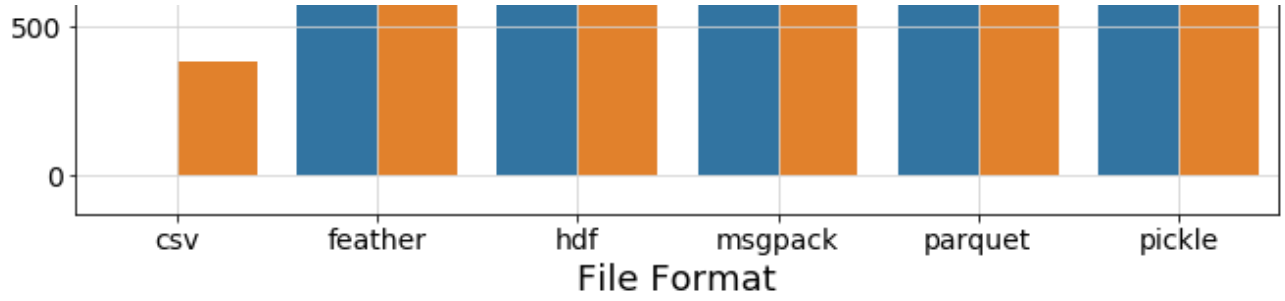
The following picture shows averaged I/O times for each data format. An interesting observation here is that `hdf` shows even slower loading speed that the `csv` one while other binary formats perform noticeably better. The two most impressive are `feather` and `parquet`.



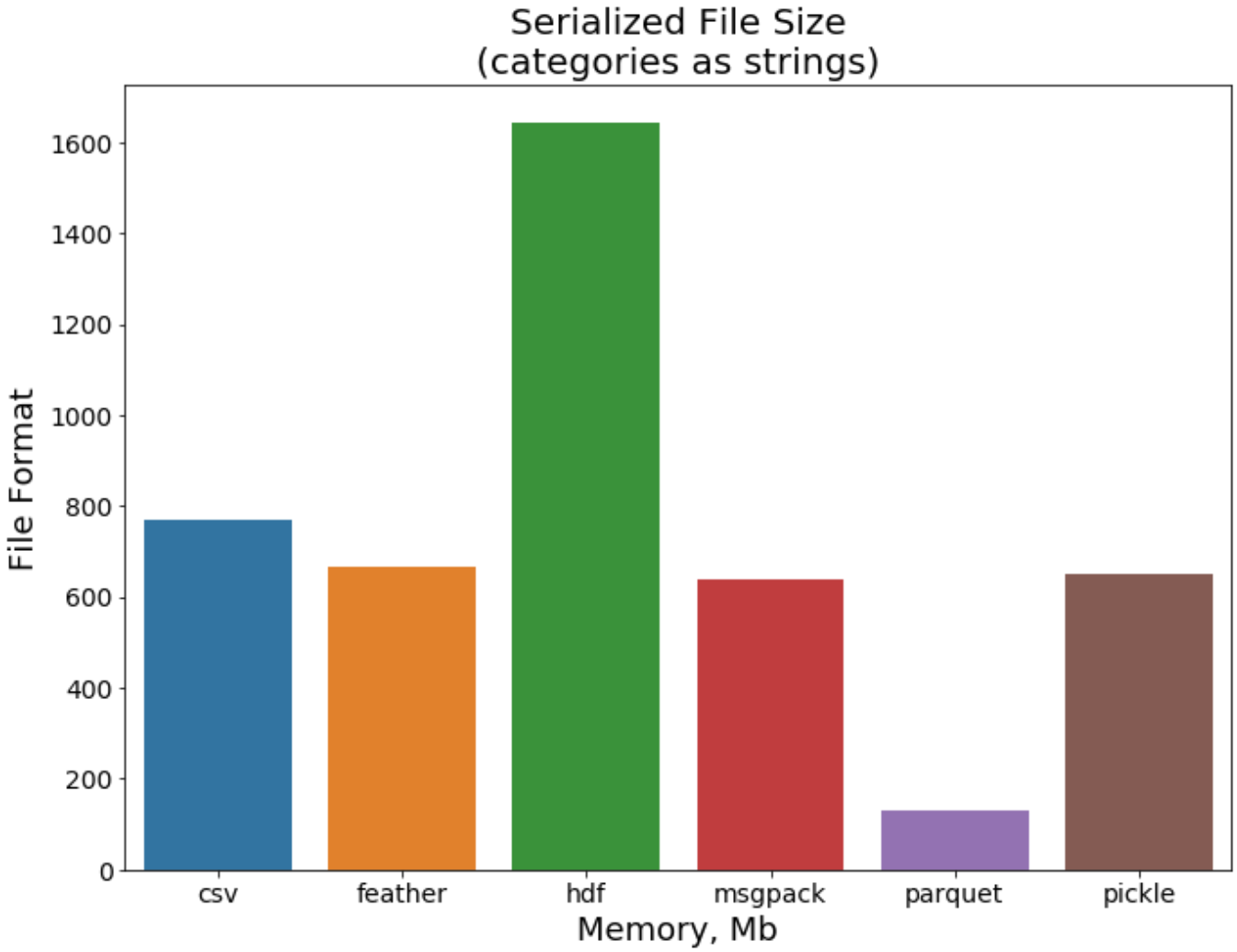


What about memory overhead while saving the data and reading it from a disk? The next picture shows us that `hdf` is again performing not that good. And sure enough, the `csv` doesn't require too much additional memory to save/load plain text strings while `feather` and `parquet` go pretty close to each other.



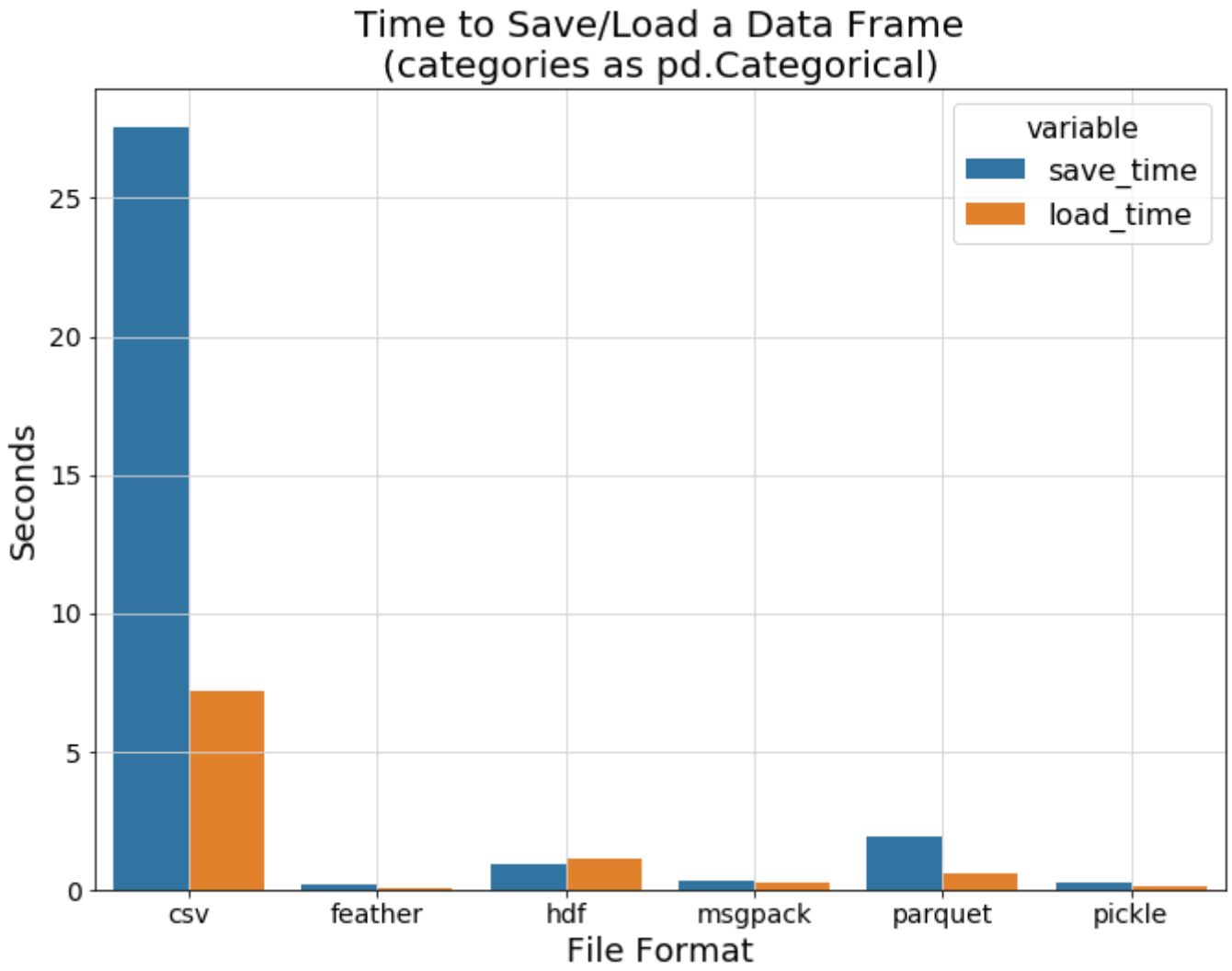


Finally, let's look at the file sizes. This time `parquet` shows an impressive result which is not surprising taking into account that this format was developed to store large volumes of data efficiently.

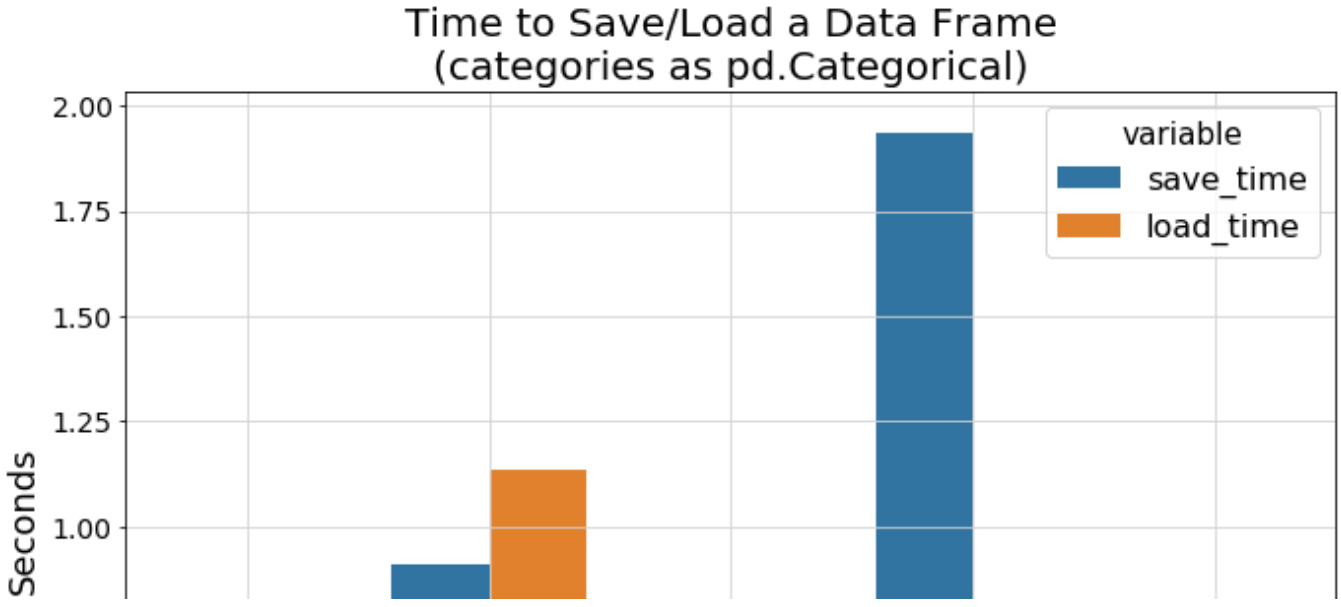


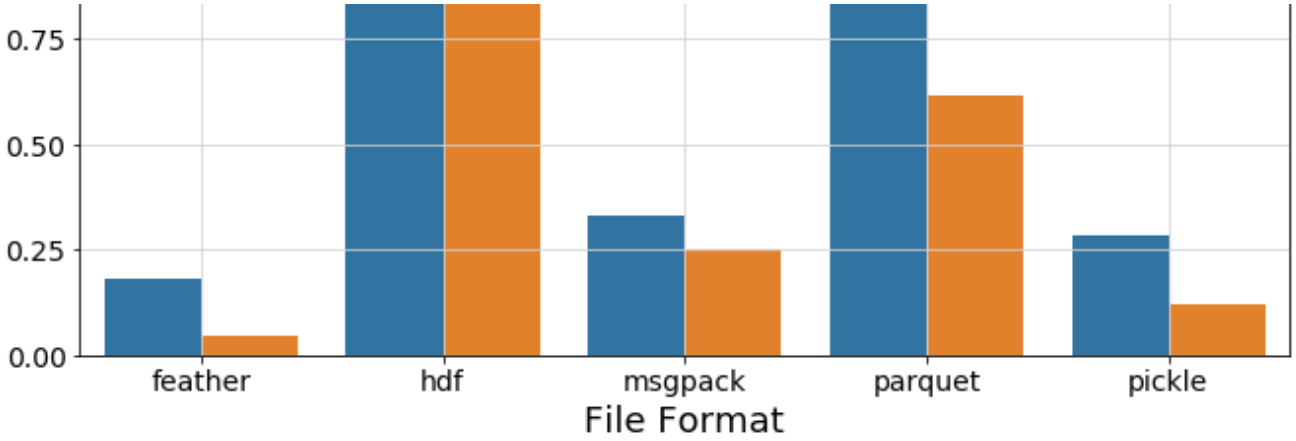
(b) Categorical Features Converted

In the previous section, we don't make any attempt to store our categorical features efficiently instead of using the plain strings. Let's fix this omission! This time we use a dedicated `pandas.Categorical` type instead of plain strings.



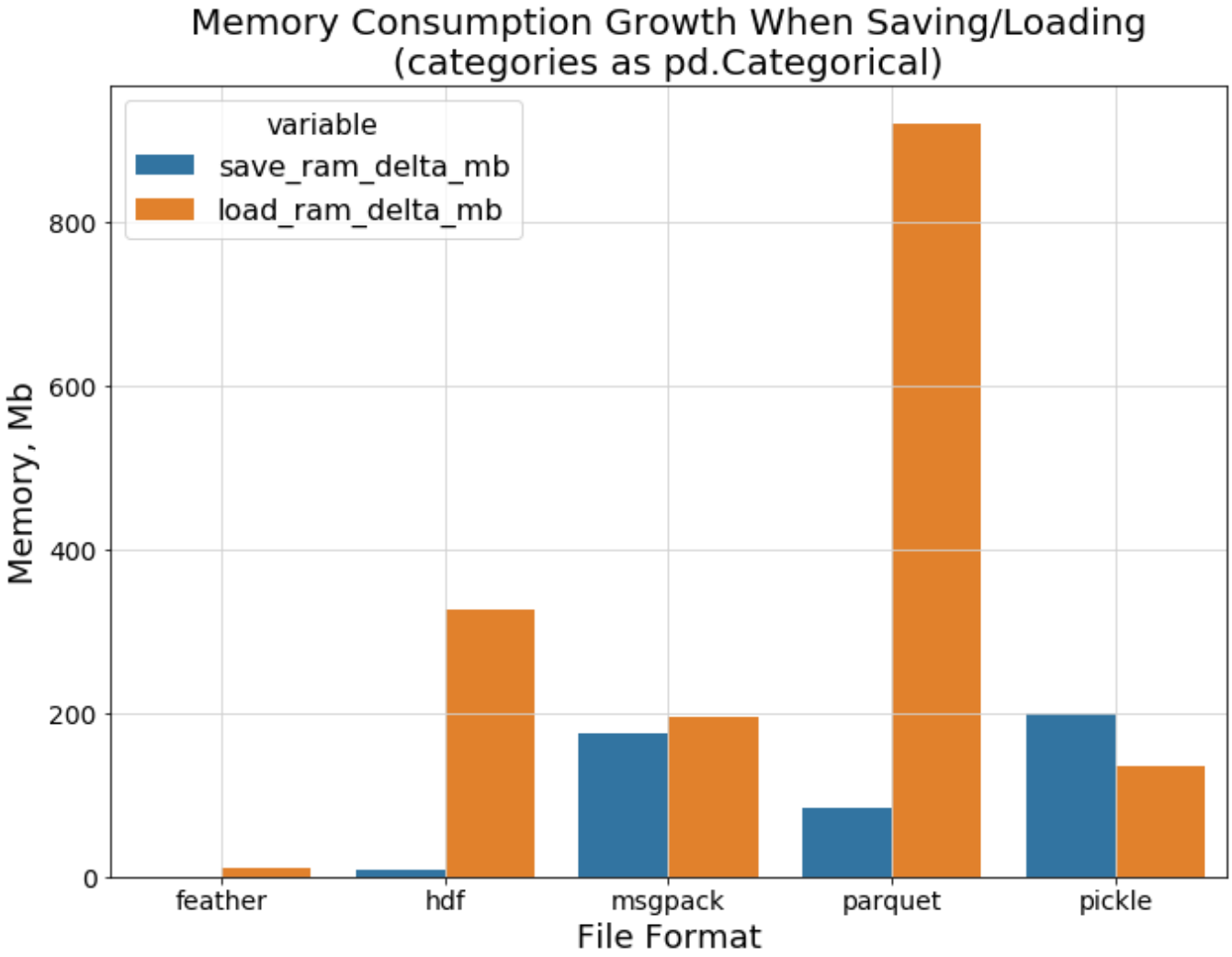
See how it looks now compared to the plain text `csv`! Now all binary formats show their real power. The baseline is far behind so let's remove it to see the differences between various binary formats more clearly.





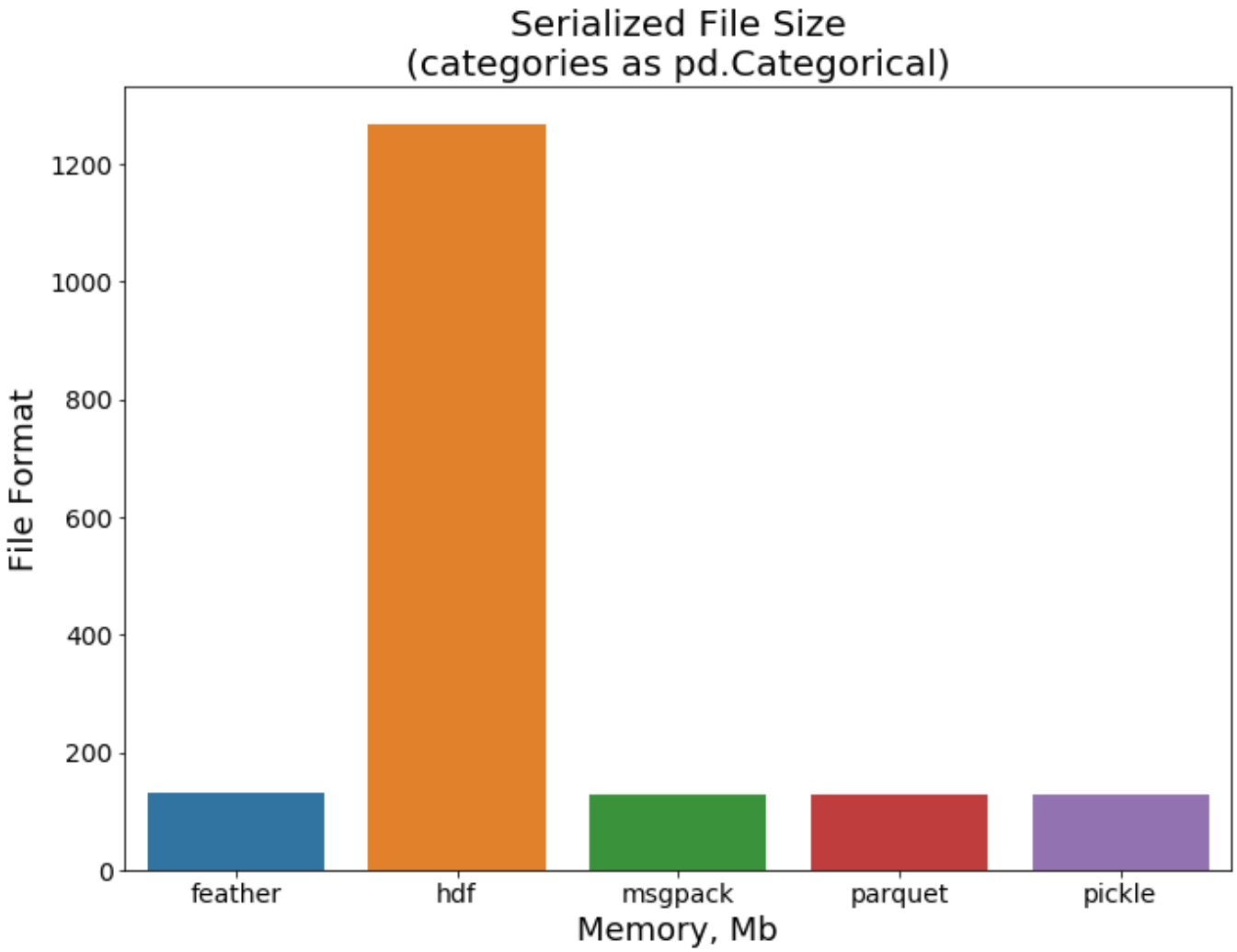
The `feather` and `pickle` show the best I/O speed while `hdf` still shows noticeable overhead.

Now it is time to compare memory consumption during data process loading. The following bar diagram shows an important fact about parquet format we've mentioned before.



As soon as it takes a little space on the disk, it requires an extra amount of resources to un-compress the data back into a data frame. It is possible that you'll not be able to load the file into the memory even if it requires a moderate volume on the persistent storage disk.

The final plot shows file sizes for the formats. All the formats show good results, except `hdf` that still requires much more space than others.



Conclusion

As our little test shows, it seems that `feather` format is an ideal candidate to store the data between Jupyter sessions. It shows high I/O speed, doesn't take too much memory on the disk and doesn't need any unpacking when loaded back into RAM.

Sure enough, this comparison doesn't imply that you should use this format in each possible case. For example, the `feather` format is not expected to be used as a long-term

file storage. Also, it doesn't take into account all possible situations when other formats could show their best. However, it seems to be an excellent choice for the purpose stated at the beginning of this post.

. . .

Are you interested in Python language? Can't live without Machine Learning? Have read everything else on the Internet?

Then probably you would be interested in my blog where I am talking about various programming topics and provide links to textbooks and guides I've found interesting.

Data Science Pandas Jupyter Notebook Python