Large data handling (Week 6)

Diego Martínez Echevarría

January 28, 2022

In this week I have worked with a large database, which has forced me to use data-handling software that can handle heavy data.

The code and this report can be seen at my Github repo[1].

1 The data source

I have been working with a 2GB csv file, generically called "large_db.csv". This database is a rough modification of the Yelp Dataset at Kaggle

I have deleted some rows just not to waste a lot of space in my computer (original file was around 4GB). For the purpose of this week's practice, I have also modified the names of the columns so they are "not nice". This means that I have included whitespace, uppercase letters and non-ascii characters, since that's the type of stuff we'll handle in this practice.

2 The data handling

For light weight databases, pandas would be the to-go-library for data handling in python. For large files, however, this is not the best solution.

For the first, pandas completely loads the file information in memory. A lot of real world applications deal with databases over 16GB, which means that pandas is just not able to work with them, if your machine hasn't got that memory.

But even if the database can be loaded into memory, it might not be the best option for a large database. Some operations can be done efficiently, without loading all the information; modifying column names or getting the data type of a column is an example.

[1] Not including the inbound or outbound databases, which are too heavy for Github

For that reason, I have used **Dask** in this project. Dask is a python module that gives a similar functionality as pandas, but delaying the heavy computation. Dask also implements many of pandas' classes and methods, which makes it easy to use.

As an example, the following is the code used to create the dataframe object in dask:

```
from dask import dataframe as dd
df = dd.read_csv(filepath)
```

The creation of this object is automatic, since Dask doesn't load the actual values inside of the database. Instead it loads some relevant information (column names, datatypes, ...) and postpones the computation of the rest.

Usually, results that need computation have a "compute" method that actually loads and actually does the work.

3 The config file

One of the tasks for this week is to create a yaml configuration file that describes the database. The following text are its contents:

```
current_file_name: large_db.csv
desired_file_name: large_db.gz
current_delimiter: ","
desired_delimiter: "|"
current_compression: infer
desired_compression: gzip
row_count: 1314000
columns:
  - user_id
  - name
  - review_count
  - yelping_since
  [...]
```

(The rest of column names are omitted)

I have written two sets of parameters: those that describe the current state of the database ("current_file_name", "current_delimeter", "current_compression") and those they describe the desired state after handling it ("desired_file_name", "desired_delimeter", "desired_compression").

Besides that I have also included the keys "row_count" (number of rows present in the database) and "columns" (the names of the columns in the database).

The names of the columns are already processed names. They don't contain uppercase, non-ascii or whitespace characters. This are the names that we expect the column names of the data to turn into, after they have been formatted.

4 The formatting

As a part of the "utils.py" file, I have written a formatter function for the columns in the dataset. This is the code of the it:

```
def format_column_names(df: pd.DataFrame) -> None:
    df.columns = list(map(lambda x: x.strip(), list(df.columns)))
    df.columns = df.columns.str.replace('[^\w]', '_', regex=True)

    df.columns = list(map(remove_non_ascii_characters, list(df.columns)))

    df.columns = df.columns.str.lower()

    df.columns = list(
        map(lambda x: remove_duplicate_char(x, "_"), list(df.columns)))
```

This does the following:

- Turn whitespace into underscores
- Remove non-ascii characters
- Lowercase everything
- Remove repeated underscores

5 The validation

Also as a part of the validation set I have written the "validate_columns", "validate_row_count" and "validate_all".

The first function checks that the amount of columns in the config file and in the database are the same, and if it is it checks that they have the same names. If they are correct, the function returns 1, and 0 (+ a message) if they don't. This is the code for it:

```
def validate_columns(dataframe: pd.DataFrame, config: dict) -> 0 | 1:
    expected_columns = set(config["columns"])
    provided_columns = set(dataframe.columns)

    if len(expected_columns) != len(provided_columns):
        print("Count of provided and expected columns differed.")
        return 0

    if expected_columns != provided_columns:
        print("Names of the provided and expected columns differed.")
        expected_not_provided = expected_columns.difference(provided_columns)
        provided_not_expected = provided_columns.difference(expected_columns)
        print(f"Expected columns not present: {expected_not_provided}")
        print(f"Provided columns not expected: {provided_not_expected}")
        return 0

    return 1
```

The other two functions are relatively more simple. With the same behavior as this function (return 1/0 if match/not match), "validate_row_count" checks that the expected row count is present at the database. Then "validate_all" is just a wrapper that calls both of the previous functions.

6 The compressing

After formatting and validating the database we are asked to save it. In addition we are asked to use the pipe character ("|") as a separator and to compress the result into a file of extension "gz" (i.e. to use the "gzip" compression algorithm).

This we can do through arguments in the Dask Dataframe's method "to_csv", which is an implementation of pandas Dataframe's function of the same method.

This is the call to that method, which I have split into several lines for better comprehension:

```
df.to_csv(df_config["desired_file_name"],
          sep=df_config["desired_delimiter"],
          compression=df_config["desired_compression"],
          index=False,
          single_file=True)
```

Where "df" is the Dask dataframe where we have loaded the database. As it can be seen, except for a couple of flags ("index", "single_file"), all of the pa-

rameters are declared using the configuration file, here loaded as "df_config", a dictionary.

7 Executing everything

At the repo one may find the "main.py" function. There I have included the main script of this assignment. This script first loads the configuration file and the database, then I validate the database before and after formatting it and finally I compress the database.

This is the output that it produces as it follows those steps:

```
BEFORE FORMATTING:
------------------
Names of the provided and expected columns differed.
Expected columns not present: {'name', 'user_id', [...]}
Provided columns not expected: {'Name', 'User id', [...]}

AFTER FORMATTING
------------------
Finished validating

Compressing file...
Finished compressing file.
```

NOTE: For greater clarity, I am only showing 2 out of the 22 different columns in the database/configuration file.

As you can see, the expected columns (columns in the yaml file) have the right format. Because of this they are detected as different columns as the provided ones (columns in the database), which include whitespace and uppercase letters.

After formatting, however, the validation doesn't print anything, which means it passed without problems. The compression of the file also doesn't give any problem, creating the expected gzip file with pipe delimiters.