# JavaDataFrame: File Data Tabular Processing

- The reviews will value both correctness and quality of the code.
- All classes should be documented (javadoc).
- Please create unit tests to validate your code (JUnit).

## Background

The goal is to design and develop a DataFrame library in Java using Design Patterns. DataFrames are used to process tabular and columnar data in programming languages like R or Python (using Pandas). You can also find previous attempts to implement this concept in Java like tablesaw, joinery and morpheus (see article).

The simplest example of a DataFrame is to import a CSV file, and filter data using rows, columns, or specific queries over data (maximum, minimum, average, values higher than a value).

DataFrames are tables of data where rows are items and columns are labeled traits. Each row is identified by an item id (usually an integer) and each column is identified by a label name (usually string). Columns and rows can thus be specified by using these identifiers, instead of the table integer coordinates.

## PART 1 – DATA FILES + FACTORY

We want to enable the program to load data files of different types (extensions). Each loaded file will be a DataFrame. A DataFrame should be an interface that allows interacting with the data regardless of the kind of file it's read from. Create different implementations for each kind of file and use the **factory pattern** to create them.

Types of files:

- CSV file: first line contains the column labels
- JSON file: dictionary with items (see gson)
- TXT file: the Java class should have the necessary logic to load a determinate data format (e.g., space separated, semicolons, quotes, etc.)

You can find many example files on the Internet (here or here). For instance, you can use this file (103MiB zipped) for testing the features that require a big file. You can also trim it or split it in different files/DataFrames to fit it to memory and test the features of the next parts.

Once you are able to import different formats using Factories, provide a simple API to access the data in the DataFrame. Include these operations:

- **at**: return the value of a single item (row) and column label (name).
  Example: If the DataFrame in `df` is the following table (here the first row and first column are the identifiers, not part of the table data):

```
      A    B    C
0     0    2    3
1     0    4    1
2    10   20   30
```

  Accessing `df.at(0, "B")` would return `2`.
- **iat**: access a single value for a row and column by integer position
  Example: in the previous DataFrame, accessing `df.iat(1,2)` would return `1`. These are the integer coordinates of the data table.

- **columns**: return number of labels (example: the previous df has 3 columns)
- **size**: return number of items (rows) (example: the previous df has 3 items)
- **sort**: return the values of a column in the DataFrame following a certain order
    - `df.sort(column, comparator)`
  - Example: on the previous DataFrame, `df.sort("C", intAscending)` returns {1, 3, 30}.
- **query**: return all elements where a label value fulfills a certain condition. The Boolean expression or condition is provided by the user, so this method should accept any condition by parameter. Examples:
    - item.user = "pedro"
    - item.age > 99
  - In the previous DataFrame, filtering elements where "C>2" would show:

```
    A   B   C
0   0   2   3
2  10  20  30
```

- Make the DataFrame object **Iterable**

## PART 2 – COMPOSITE

We will use the **composite pattern** to create composite DataFrames that contain several files in a directory, or even files in subdirectories. We could for example use directory names to categorize data: EU/Spain, EU/France, …

A composite DataFrame has the same operations as normal DataFrames. In the composite, nodes are directories and leaves, files. Nodes and leaves also act as DataFrames. DataFrame operations can be executed at any level of the composite.

## PART 4 – STREAMS AND MAPREDUCE

Try to use **Java Streams**, and functional programming when possible.

We want the system to handle several DataFrames at the same time (collection of DataFrames).

We will implement a MapReduce computing design pattern with two clear phases: a parallel Map phase to apply an operation over a collection of DataFrames, and a Reduce phase that will collect and join the results of the different Map workers.

The MapReduce pattern must work over either simple DataFrames (a single file) or composite DataFrames (a top directory). We will have a list of DataFrames loaded in the system and apply the MapReduce to that collection, one Map worker per DataFrame. Use Streams, if possible, in the Map phase, to apply operations in parallel over collections of DataFrames.

Example: compute the average of all users in a collection of DataFrames. The map obtains the data from all DataFrames and the reduce uses these data to compute the average.

[Optional] A complex operation over multiple DataFrames is a join operation, that combines results from different files in a single result. For example, obtain all rows where user=pedro in a group of DataFrames. We can leverage the MapReduce code to execute generic join operations.

## PART 5 – VISITOR

Add the **visitor pattern** to the previous composite structure. The visitor must allow to run new operations on all DataFrames in the composite (recursively through the directories).

Code the following operations (visitors):

- **maximum**: return the item with the maximum value in a determinate label
- **minimum**: return the item with the minimum value in a determinate label
- **average**: return the average value in a label of all items in the DataFrame
- **sum**: return the sum of all values in a label

## PART 6 – OBSERVER AND DYNAMIC PROXY

Use the **dynamic proxy** to create an Interceptor of the DataFrame.

Using the **observer pattern**, enable the subscription of observers to the interceptor.

- Create a LogObserver that logs all operations executed in a DataFrame.
- Create a QueryObserver that only logs operations that satisfy a filter (user=pedro).

## PART 7 – UNIT TESTING AND DOCUMENTATION

Provide a GitHub repository including basic documentation in English, examples, java docs, APIs, and unit tests. Examples and tests should show how all features implemented work.