Libsheet

A lightweight data analysis C++ library

- https://github.com/YufeiOu/libsheet.git

Liang-Chun Tsai (lt2590)
Ming-Ching Chu (mc4107)
Yufei Ou (yo2265)

Design Document

1. Data structure overview

- Sheet

libsheet store and represent the data as the class Sheet. Every sub part of the original data is also a Sheet instance. A sheet instance has two main component

- std::vector<ColumnHead> columns
- std::unorder_map<stirng, unsign_int> column_map

columns is a vector that stores instance of a special class ColumnHead that is a wrapper of std::vector that is actually holding the data, column by column, in the library. We will discuss more about ColumnHead later.

column_map is the a hash_map that can map the name of the column to a column index. This data structure is mainly for supporting column name indexing in many operations that give user more convenience to process data.

- ColumnHead

ColumnHead is the wrapper of the std::vector that actually stores the data. Since we support three types of data. We have three std::vector in each ColumnHead. Each one supports different type of vector. vector<int> vint, vector<double> vdouble, vector<string> vstring. Among these three vector only one vector will actually holding data for a column depends on the type of the column. ColumnHead stores a variable called flag with a value of 0, 1, 2, each corresponds to the type of the column. Using this flag, we can decide what is the type of this column; as a result, we can decide which vector we should deal with.

ColumnHead also stores a column_name that is used for column name indexing.

Notice we have thought about using boost:any<T> to store the value, as a result we will only need one vector<boost:any> to stores the data for each column. However, we find this design cannot solve the fundamental problem. As the goal of our library is to process the data not just “store” the data. For any operation on the data like sort and filter, we still needs to cast the boost:any type into it’s original type. For example boost::any_cast<int>(). Otherwise, even some basic operator like > cannot work on this data. If we have to cast the type everytime we do operations on the data, it will become a huge burden. Since we have to cast the data can restore it into another new vector. We think it will decrease our performance a lot. On the other hand, in our current design, it may seems we use three std::vector for each ColumnHead is a waste of resource. Actually, only one of them will actually holding data, the other two std:vector is only empty vector so the resource it takes is just the default spaces a std::vector may take while we don’t have to cast the data in further operation.

2. Data loading implementation

libsheet allows the user to initialize an empty instance and load data into it later or it can also be initialize with one row of data. The three related functions is

- Sheet(vector<string>& entry, vector<string>& col_names, const string& NAN_symbol = "");
- void row_append(vector<string> &new_row, const string& NAN_symbol = "");
- void load_data(Sheet& sheet, const string& path, bool header = true, const string& NAN_symbol = "");

load_data allows user to fill an empty sheet with a comma separated txt format file. It will infer the data type from the txt file. The function to infer the data type is implemented as get_type(const string&). Before we input the string into get_type, we will trim the space beside the actual string. What get_type does is it will first see if it can transform it in to a integer, if it does it will be detected as integer type and next it will detect if it can be transformed to a double type. If both is not possible, it will be detected as string type. In the afore-mentioned function 1 and 3 will do this detection. Row append() will also transform the string value into primitive type but it will not do type determination, it will just depends on the corresponding ColumnHead to do type casting. These functions also allows user to specify how to interpret the missing value in original txt file and input string. In these three functions, the parameter NAN_symbol indicates how the Sheet should interpret "missing" in the original data. The default value is "", which means a real missing between two ",". The other possible choices are to define it as NA, nan, NaN, NAN.

3. Condtition selection implementation

As mentioned in library usage document, we provide condition selection of the rows of which the value in specified column meets the condition. The condition should be defined by the select function. Multiple conditions can be combined together using logic operator '&&', '||', '!'.

This function is

Sheet filter(const vector<bool>& vb)

The function takes "condition" defined by user and returned a new copy of the sub-Sheet that contains the rows with specific column fulfill the conditions. The "condition" is defined by vb which works on specific column and it should be the return value of either iselect/dselect/sselect depends on the type of the columns that the condition works on. The return value vector<bool> indicate whether certain row meets the conditions. The filter function will first transform vector<bool> into vector<int>, which is the column that should be returned and call get_row(vector<int>) to get the rows that fulfill the conditions.

Next, we take a further look at the selection function.

```
// The condition works on integer column
vector<bool> iselect(int col, Function fn);
vector<bool> iselect(string col, Function fn);
// The condition works on double column
vector<bool> dselect(int col, Function fn);
vector<bool> dselect(string col, Function fn);
// The condition works on string column
vector<bool> sselect(int col, Function fn);
vector<bool> sselect(string col, Function fn);
```

The condition is defined by Function fn, fn is a function object in C++. It can simply be a function name or using lambda expression to define the function. Either way, the function should only takes 1 input argument that has the same type of data in the column that the condition will work on. The return of this condition function should be boolean value. The reason why we need three different functions is because we have to decide which vector among vint, vdouble, vstring in the ColumnHead should the fn works on. Since the user defined function has a specific input type. It is impossible to use only one selection function and apply this fn on three different types of column vector. Even though we can use switch condition to make the program looks as if it will only execute one type at run time. The compiler cannot let fn has possibility to work on three different types. (This problem cannot be solved by using vector<boost:any>, since you have to still have to cast the value into three different types and the problem remains)

To support multiple conditons. We also overload three logic operator &&, ||, ! of vector<bool>. The overloading is very simple, user can user these operator among multiple selection() functions. The type of the function doesn’t have to be the same since their return value is all vector<bool>.

4. Sort by column implementation

libsheet support sorting on a column. The main idea is to find the resulting row-id sequence, and reorder each column according to that sequence. Firstly, we locate the vector storing data from the column the user specified, and pair each element in the vector with its row id in to a pair<data_type, int>, where the data_type in our structure can be int, double, or string. These pair<data_type, int> sets are stored as a vector. In this step, we skip entries that has a missing value in the column we're looking at, and keep note for their row id for future use. After doing so, all entries having a non-missing value are stored as pair<data_type, int> in the "data_id" vector, and those having missing value are stored in the "nan_indices" vector. Seconly, we sort the vector<pair<data_type, int>> acendingly or descendingly according to user input. Thirdly, we extract the row id in every pairs into another vector called "indices". Once we have the "indices" vector, we call another function reorder() on every columns to rearrange their data order according to indices. reorder() takes in the indices vector and a vector<data_type>. This function goes through the indices vector to get the corresponding element from vector<data_type> and push-back into a new vector. Finally, we assign the new vector to replace the old vector. After applying the reorder() function to every column, all data would be stored in the correct ordering.

5. Missing value handling

Since in actual world, it is very likely that there is some missing in the data, libsheet want to allow the user to still process the data not because it is missing only a tiny portion of data. Thus, libsheet support missing value. For each primitive type that Sheet can store, there is one value that represent "missing" in the data. It is defined in libsheet.h by

```
#define NAN_int INT_MIN
#define NAN_double NAN
#define NAN_string ""
```

If the user wants to change these value it can only be changed from here. As the missing value for integer is stored as INT_MIN, the user should be aware of doing calculation on integer and create INT_MIN value.

As a result, missing value is not actually strictly missing. It is just treated as missing. Thus, in our sorting, filtering, apply function, we will detect these missing value in specified column and simply skip these value that would not generate unexpected behavior after the operation. Thus, a missing value after any operation is still a missing value.

Sheet also allows user to specify how to interpret the missing value in original txt file and input string. There are three functions in Sheet that read data in string format.

```
1. Sheet(vector<string>& entry, vector<string>& col_names,
      const string& NAN_symbol = ""); (see Section 1)
2. void row_append(vector<string> &new_row,
            const string& NAN_symbol = ""); (see Section 6)
```

```
3. void load_data(Sheet& sheet, const string& path, bool header = true,
            const string& NAN_symbol = ""); (see section 1)
```

In these three function, the parameter NAN_symbol indicates how the Sheet should interpret "missing" in the original data. The default value is "", which means a real missing between two ",". The other possible choices is to define it as NA, nan, NaN, NAN.

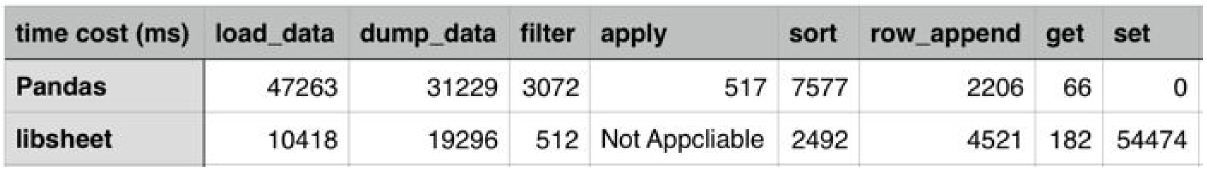
Similarly, the dump function also allows user to specify how to dump the missing value in print function. The missing value will only be dumped as the user specified value but not it’s actual value, say INT_MIN or NAN (double) and ““.

Moreover, Sheet provides a utility function that allows user to decide whether certain value will be treated as "missing" in the Sheet.

```
bool isNAN(int i) {return i == NAN_int;}
bool isNAN(double d) {return isnan(d);}
bool isNAN(string s) { return s == NAN_string;}
```

6. Performance evaluation

The run time performance of libsheet is compared with python library Pandas. The dataset we use is 276MB in size, which has approximately 2 millions rows and 20 columns, and contains integer, double, and string data types. We perform load_data, dump_data, filter, apply, sort, row_append, get and set functions to both libraries and timed the operations. Evaluation result is shown below.



time cost (ms)	load_data	dump_data	filter	apply	sort	row_append	get	set
Pandas	47263	31229	3072	517	7577	2206	66	0
libsheet	10418	19296	512	Not Appcliable	2492	4521	182	54474

Fig. 1 operation time costs in ms.

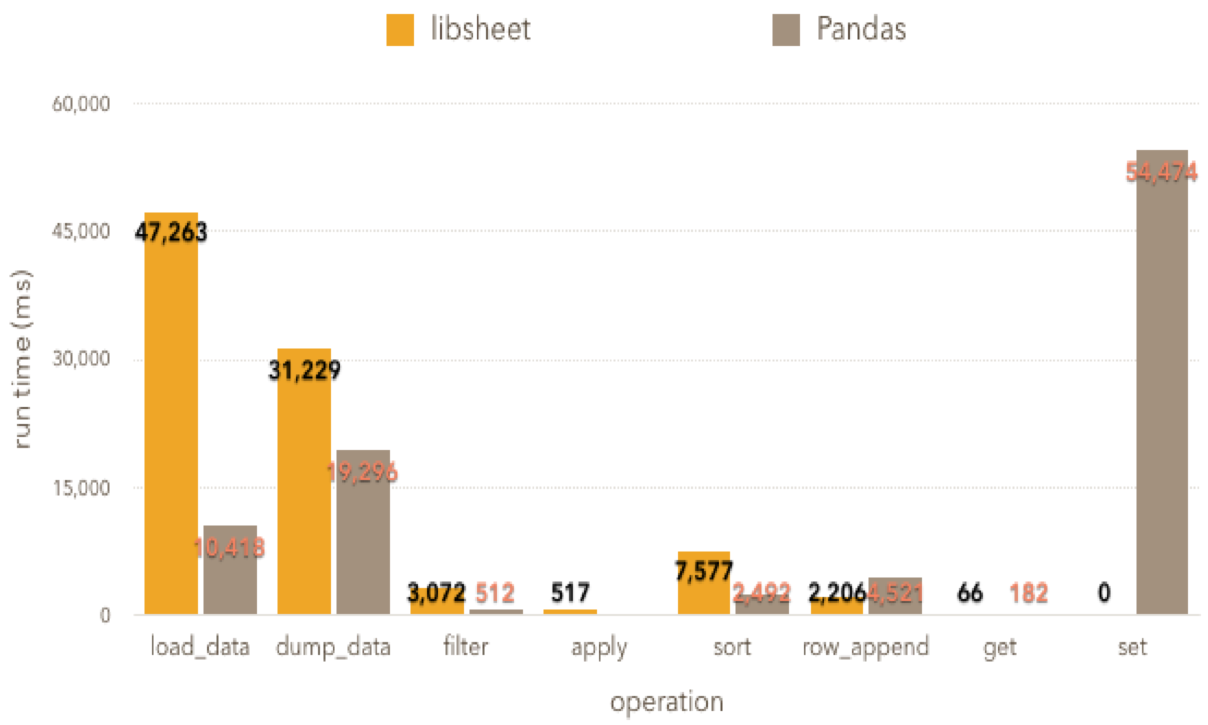


Fig. 2 performance comparison

Those functions that involves file I/O are slower in libsheet than it is in Pandas. The main reason behind this is that we read in the data from file row by row but append data in the sheet column by column and one element at a time. As explained in session 2, the way we implement load_data did not utilize the caching advantage of sequential vector push-backs. Rather, we push-back new element to a column and switch to the next column to push-back the next one. We switch the column to push-back circularly and add one element at a time. Every time we switch on to another column, we need to do preprocessing on raw data and check the column type. These checking process is done to every single element, and it's costly when we have large dataset. Similarly for dump_data, we must check the "flag" in Column_head to know the type, and print out data from the corresponding vector in Column_head.

Aside from I/O functions, filter and sort functions are relatively slow too. That is because once we have the new index ordering sequence, we need to reorder every other columns according to the index sequence. To have better performance, the user should utilize the operators to combine all filtering conditions before they use filter() to get the final resulting sheet.

On the other hand, row_append, get, and set is faster in libsheet. In particular, set functions takes almost no time to complete. It shows that libsheet is very efficient in operations that does not require type checking or column switching.

Lastly, libsheet provide flexible and fast apply() functions, which Pandas does not support.