libsheet

A lightweight data analysis C++ library

By: Liang-Chun Tsai, Ming-Ching Chu, and Yufei Ou

Contents

1. Motivation
2. A quick look at how to use it
3. Architecture design
4. Main data structure and operations
5. Experimental result
6. Challenges and future work

Motivation

Why we bother replace Excel Spreadsheet by a C++ library?

1. Sometimes, open, sort, manipulate in Excel is cumbersome, especially when we only need to load part of data we want (into other program for further use).
2. It’s very slow to open large file in Excel, what if we only need a little part of those data?
3. Did anyone use Excel in Linux?

Motivation(cont.)

- Why not using other programming language?
  Ex. Python : pandas.DataFrame
  R : data.frame

  It’s hard to integrate the data processing using other language to C++. Our library provide a simple data analysis pipeline for C++.

- Why not using database?

  Database is not in memory and it needs a third party software (not just C++)
  Database doesn’t allow user-defined condition selection

What is libsheet?

A lightweight C++ library that supports in memory **selection**, **projection**, **sorting** and **modification** to an Excel-spreadsheet-like data structure, with a bunch of auxiliary functions (such as easily load and print).

Highlights of libsheet include:

1. Multi-type container in C++
2. Support NaN (missing value)
3. Flexibility in filtering and modification of data, support lambda expression and multiple conditions
4. Support column name indexing
5. Stable & Faster & Scalability (More type and more operations)

Demo

1. Easy load and print!
2. Easy select!
3. Support relatively complex query!
4. Easy manipulation!
5. Communicate with user program without cumbersome operation!
6. And many other useful operations such as get sub-sheet, set the value of a certain place, and so on.

Architecture

libsheet.h - header file, template function definition

libsheet.cpp - function definition, class definition

test.cpp - unit testing

timer.cpp - performance measurement

demo.cpp - for your pleasure

Basic data structure

- Everything is a class - Sheet (Return type for all operation)
- Sheet
  - vector<ColumnHead> columns
  - unordered_map<string, unsigned int> column_map : To store the column name index
  - Lots of operation : get(), set(), sort_by_column(), filter(), apply(), select(), append(), erase()
- ColumnHead
  - string column_name
  - int flag : To store the type of this column, we now support **int, double** and **string**
  - vector<int> vint;
  - vector<double> vdouble;
  - vector<string> vstring;
  - **Only one of the vector will actually store data for the column**

Container operations

- Sheet **get**(int / vector<int>, int / string / vector<string> / vector<int>)
  - **Support column name index**
- Sheet **get_row**(int / vector<int>), **get_col**(int / string / vector<int> / vector<string>)
- void **row_append**(vector<string> / Sheet)
- void **col_append**(vector<int> / vector<double> / vector<string> / Sheet)
- void **row_erase**(...), **col_erase**(...)
- int **col_len**(int), **row_len**(int)
- void **set**(...)
- vector<int> **get_ivec**()
- vector<double> **get_dvec**()
- vector<string> **get_svec**()

Advanced Operations

- void **sort_by_column**(): we can sort by any column either ascending or descending
  - sheet.sort_by_column(*column_number*);
- Sheet **filter**(vector<bool>)
  - sheet.filter(*column_of_bool_picked_by_select*);
- vector<bool> **iselect**(column, Function Object)
  - User define its own functor that takes 1 argument of desired type and work on 1 column
- vector<bool> operator overload : **&&, ||, !**
  - Support multiple conditions for filtering
- void iapply: we can “apply” a function object (can be lambda exp) to a column
  - sheet.apply(*column_number*, *lambda_expression*);

IO

- **load_data**(Sheet&, “file_path”, header=true, NAN_symbol=””)
  - Build sheet from file
- **print**(bool header = true, const string& nan_symbol = "NAN")
  - std::cout
- **print**(const char* file_path, bool header = true, const string& nan_symbol = "NAN")
  - Save to file

Missing Value Handling

Why it matters?

- It’s a real world problem.
- Essential for join.

We define default NaN values:

- Int: INT.MIN
- Double: NAN
- String: “”

And user can explicitly assign NaN value while building the sheet.

A lesson of using template

- Return Type is template

```
template <typename T>
T Sheet::get(int i, const string& col) {
    /*
    Get the column head ch and get the flag
    */
    switch(flag) {
        case 0:
            return ch.vint[i]
            break;
        case 1:
            return ch.vdouble[i]
            break;
        case 2:
            return ch.vstring[i]
            break;
        default:
            ...
    }
}
```

- Argument Type is template

```
template <typename T>
Sheet get(T col_id, int row_id){
    Sheet sh;
    if (typeid(T) == typeid(int)) {
        /* get element by column index */
    }
    else {
        /* get element by column name */
    }
    return sh;
}
```

A lesson of using template(cont.)

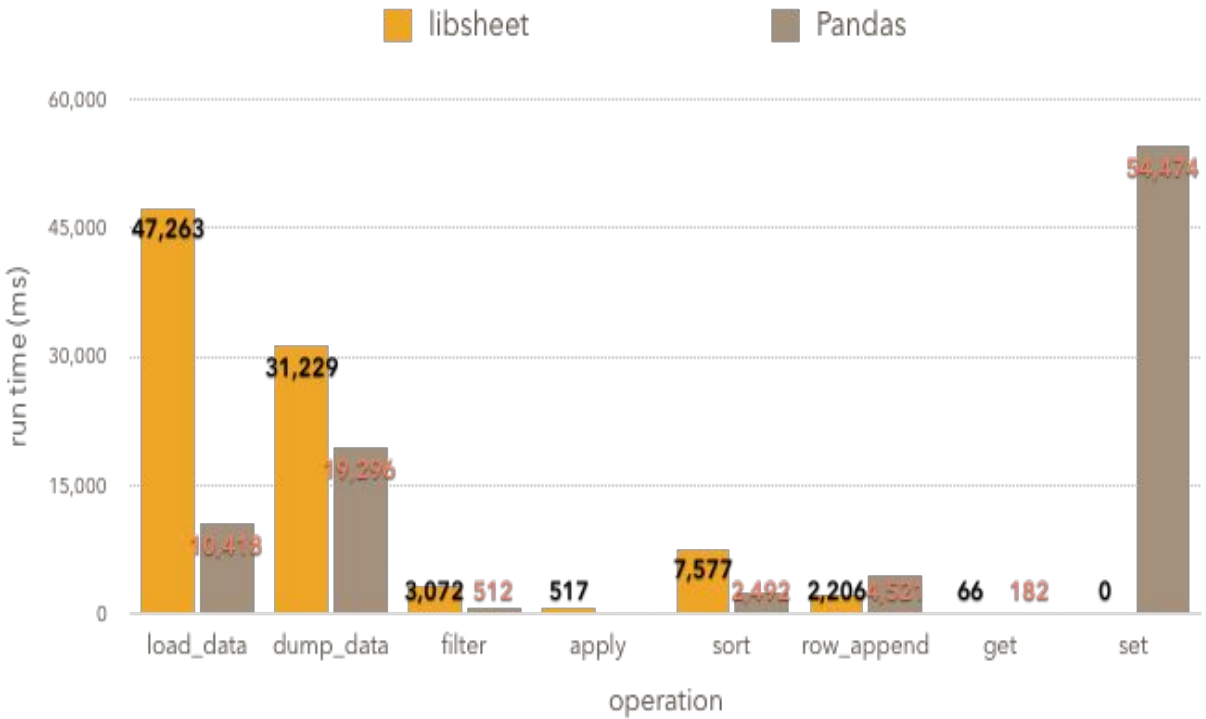
- Function object as an argument

```
template <typename Function>
void apply(int col, Function fn) {
    /*
    Get the column head ch and the type
    */
    switch(flag) {
        case 0:
            for (auto& r : ch.vint[i]) fn(r);
            break;
        case 1:
            for (auto& r : ch.vdouble[i]) fn(r);
            break;
        case 2:
            for (auto& r : ch.vstring[i]) fn(r);
            break;
        default:
            ...
    }
}
```

Testing plan: Black box unit testing

- At first, we wrote the fewest code that could be tested:
  - Sheet class structure
  - copy constructor
  - load_data
  - print
- Vertical development process: every time we created a new part, we test it!
  - get & set module
  - select & apply module
  - append & erase module
- When everything works well, we deal with NaN handler!
  - Start from load_data, repeat testing everything again!

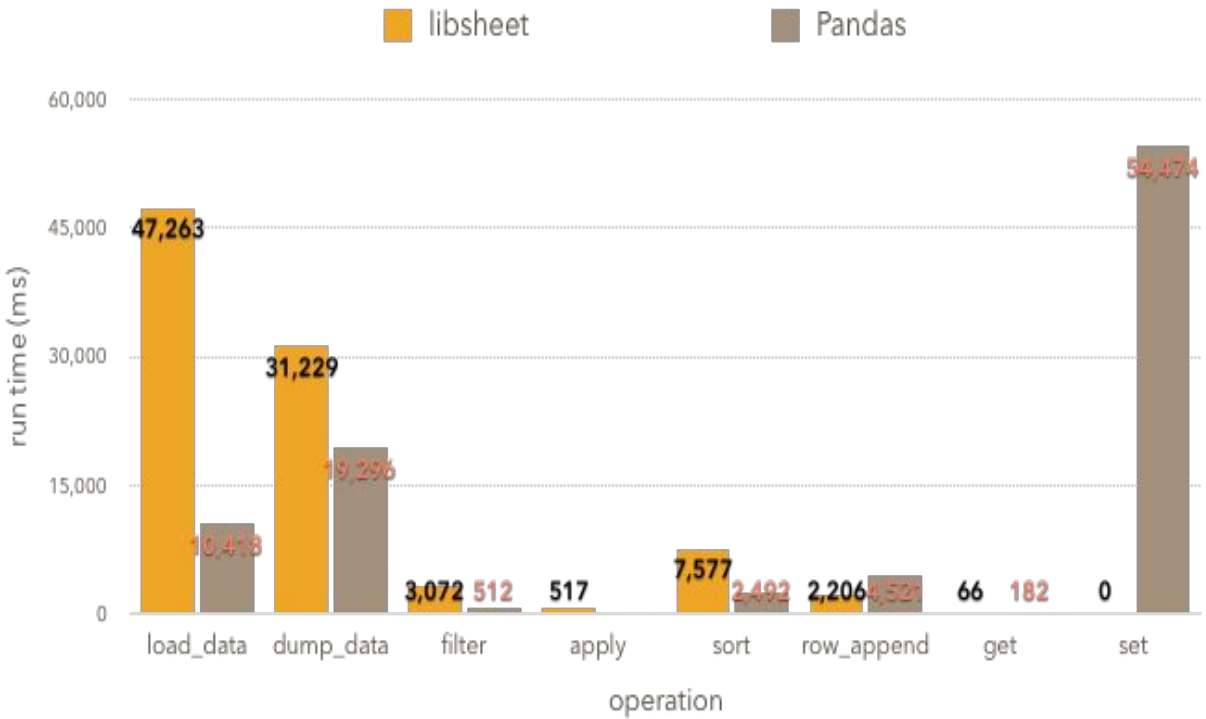
Evaluation: Timed result



1. Slow in I/O:
   Did not utilize vector caching.

2. Slow in Sort():
   Do a lot of “reorder according to indices”, which involve copying.

Evaluation: Timed result



1. Fast in row_append(): Utilize vector caching.

2. Very fast get() /set()

Conclusion and future work

We successfully implement 0.8 edition of libsheet library which has:

1. Multi-type container in C++
2. Support NaN (missing value)
3. Flexibility in filtering and modification of data, support lambda expression and multiple conditions
4. Support column name indexing
5. Stable & Faster & Scalability (More type and more operations)

Future Work:

1. Join function
2. Summary function: mean, median, max, min, histogram, count
3. Condition selection involves multiple columns
4. More data type: Categorical, Date time


Q&A

Acknowledgements

Special thanks to:

- Professor Bjarne Stroustrup's extraordinary lectures and instructions
- TA David's kind help and suggestions to this project.
- NWC library's beautiful desk and computer.

This is a group photo with Bjarne and David we plan to take after this class...