**Motivation**

Although Pandas has a spot-on interface and it is full of useful functionalities, it lacks performance and scalability. For example, it is hard to decipher intensive intraday data such as Options data or S&P500 constituents tick-by-tick data using Pandas.

Another issue that I have encountered is, often, the research is done using Python, because it has such tools as Pandas, but the execution in production is in C++ for its efficiency, reliability and scalability. Therefore, there is this translation, or sometimes a bridge, between research and executions.

Also, in this day and age, C++ needs a heterogeneous data container. Mainly because of these factors, I implemented the C++ DataFrame.

*This library is still far from complete. It needs much more statistical and logical functionalities. I welcome all contributions from people with expertise, interest, and time to do it. I will add more functionalities from time to time, but currently I don’t have much free time.*

**Code structure**

The DataFrame library is almost a header-only library with one source file exception, *BaseContainer.cc*.

Starting from the root directory;

*DMScu* is a helper module that contains a few objects. One is a stack-based string object and the other is a Linux-only mmap file interface. These objects are used by the DataFrame library and therefore must be compiled beforehand (see build instructions below).

*include* directory contains most of the code. It includes *.h* and *.tcc* files. The later are C++ template code files. The main header file is *DataFrame.h*. It contains the entire DataFrame object and its interface. There are comprehensive comments for each interface call in that file. The rest of the files there will show you how the sausage is made.

*src* directory has the only source file for the library, make-files, and a test program source file. The test source file is *datasci\_tester.cc*. It contains test cases for almost all functionalities of DataFrame. It is not in a very organized structure. I plan to make the test cases more organized.

**Build Instructions**

Using plain make and make-files

Go to the root of the repository, where license file is, and execute *build\_all.sh*. This will build the library and test executables for Linux flavors.

Using cmake

Coming soon for Linux, Windows, Mac, …, thanks to [@justinjk007](https://github.com/justinjk007)

**General**

This library is based on a heterogenous vector. The heterogeneity is achieved by using static STL or STL-like vectors. Since C++ is a strongly typed language, you still have to know your column types per container at compile time. You can add more columns with different types at any time to your container, but when analyzing the data at any given time you must know the column types.

This is an example of how to create a DataFrame, load data, and run an operation on it:

// Defines a DataFrame with unsigned long index type that used std::vector

typedef DataFrame<unsigned long, std::vector> MyDataFrame;

MyDataFrame df;

std::vector<int> intvec = { 1, 2, 3, 4, 5 };

std::vector<double> dblvec = { 1.2345, 2.2345, 3.2345, 4.2345, 5.2345 };

std::vector<double> dblvec2 = { 0.998, 0.3456, 0.056, 0.15678, 0.00345, 0.923, 0.06743, 0.1 };

std::vector<std::string>. strvec = { "Col\_name", "Col\_name", "Col\_name", "Col\_name", "Col\_name" };

std::vector<unsigned long> ulgvec = { 1UL, 2UL, 3UL, 4UL, 5UL, 8UL, 7UL, 6UL }

std::vector<unsigned long> xulgvec = ulgvec;

// This is only one way of loading data into the DataFrame. There are

// many different ways of doing it. Please see *DataFrame.h* and *datasci\_tester.cc*

int rc = df.load\_data(std::move(ulgvec),

std::make\_pair("int\_col", intvec),

std::make\_pair("dbl\_col", dblvec),

std::make\_pair("dbl\_col\_2", dblvec2),

std::make\_pair("str\_col", strvec),

std::make\_pair("ul\_col", xulgvec));

// Sort the Frame by index

df.sort<MyDataFrame::TimeStamp, int, double, std::string>();

//Sort the Frame by column “dbl\_col\_2”

df.sort<double, int, double, std::string>("dbl\_col\_2");

// A functor to calculate mean, variance, skew, kurtosis, defined in *DFVisitors.h* file

StatsVisitor<double> stats\_visitor;

// Calculate the stats on column “dbl\_col”

df.visit<double>("dbl\_col", stats\_visitor);

|  |
| --- |
|  |
|  |
|  |

For more examples see file *datasci\_testr.cc*

**Methods, Types, and more**

*template<typename TS, template<typename DT, class... types> class DS>*

*class DataFrame*

DataFrame is a class that has:

An index column of type TS (timestamp, although it doesn’t have to be time)

It uses type DS as Data Storage. For example, std::vector

*template<typename T>*

*DS<T> &create\_column(const char \*name);*

It creates an empty column named “name”

T: Type of the column

Returns a reference to the vector for that column

*template<typename … Ts>*

*size\_type &load\_data(TSVec &&indices, Ts … args);*

This is the most generalized load function. It creates and loads an

index and a variable number of columns. The index vector and all

column vectors are "moved" to DataFrame.

Ts: The list of types for columns in args

indices: A vector of indices (timestamps) of type TimeStamp;

args: A variable list of arguments consisting of

std::pair(<const char \*name, DS<T> &&data>).

Each pair, represents a column data and its name

Returns number of items loaded

template<typename ITR>

size\_type load\_index(const ITR &begin, const ITR &end);

It copies the data from iterators begin to end into the index column

ITR: Type of the iterator

Returns number of items loaded

template<typename ITR>

size\_type load\_index(const ITR &begin, const ITR &end);

It copies the data from iterators begin to end into the index column

ITR: Type of the iterator

Returns number of items loaded

size\_type load\_index(TSVec &&idx);

It moves the idx vector into the index column.

Returns number of items loaded

template<typename T, typename ITR>

size\_type load\_column(const char \*name,

const ITR &begin,

const ITR &end,

bool pad\_with\_nan = true);

It copies the data from iterators begin to end to the named column.

If column does not exist, it will be created. If the column exist,

it will be over written.

T: Type of data being copied

ITR: Type of the iterator

name: Name of the column

pad\_with\_nan: If true, it pads the data column with nan if it is shorter than the index column.

Returns number of items loaded

template<typename T>

size\_type

load\_column(const char \*name, DS<T> &&data, bool pad\_with\_nan = true);

It moves the data to the named column in DataFrame.

If column does not exist, it will be created. If the column exist,

it will be over written.

T: Type of data being moved

name: Name of the column

pad\_with\_nan: If true, it pads the data column with nan,

if it is shorter than the index column.

Returns number of items loaded

size\_type append\_index(const TimeStamp &val);

It appends val to the end of the index column.

Returns number of items loaded

template<typename T>

size\_type append\_column(const char \*name,

const T &val,

bool pad\_with\_nan = true);

It appends val to the end of the named data column.

If data column doesn't exist, it throws an exception.

T: Type of the named data column

name: Name of the column

pad\_with\_nan: If true, it pads the data column with nan,

if it is shorter than the index column.

Returns number of items loaded

template<typename ITR>

size\_type append\_index(const ITR &begin, const ITR &end);

It appends the range begin to end to the end of the index column

ITR: Type of the iterator

Returns number of items loaded

template<typename T, typename ITR>

size\_type append\_column(const char \*name,

const ITR &begin,

const ITR &end,

bool pad\_with\_nan = true);

It appends the range begin to end to the end of the named data column.

If data column doesn't exist, it throws an exception.

T: Type of the named data column

ITR: Type of the iterator

name: Name of the column

pad\_with\_nan: If true, it pads the data column with nan,

if it is shorter than the index column.

Returns number of items loaded

template<typename ... types>

void make\_consistent ();

Make all data columns the same length as the index.

If any data column is shorter than the index column, it will be padded by nan.

This is also called by sort(), before sorting

template<typename T, typename ... types>

void sort(const char \*by\_name = nullptr);

Sort the DataFrame by the named column. By default, it sorts by index (i.e. by\_name == nullptr).

Sort first calls make\_consistent() that may add nan values to data columns.

nan values make sorting nondeterministic.

T: Type of the by\_name column. You always of the specify this type,

even if it is being sorted to the default index

types: List all the types of all data columns.

A type should be specified in the list only once.

template<typename T, typename ... types>

std::future<void> sort\_async (const char \*by\_name = nullptr);

Same as sort() above, but executed asynchronously

template<typename F, typename T, typename ... types>

DataFrame groupby (F &&func,

const char \*gb\_col\_name = nullptr,

bool already\_sorted = false) const;

Groupby copies the DataFrame into a temp DataFrame and sorts the temp df by gb\_col\_name before performing groupby. If gb\_col\_name is null, it groups by index.

F: type functor to be applied to columns to group by

T: type of the groupby column. In case if index, it is type of index

types: List of the types of all data columns.

A type should be specified in the list only once.

func: The functor to do the groupby. Specs for the functor is

in a separate doc.

already\_sorted: If the DataFrame is already sorted by gb\_col\_name,

this will save the expensive sort operation

std::future<DataFrame>

groupby\_async (F &&func,

const char \*gb\_col\_name = nullptr,

bool already\_sorted = false) const;

Same as groupby() above, but executed asynchronously