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

Please see README file. Thanks to @justinjk007, you should be able to build this in Linux, Windows, Mac, and more

Example

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*

Types and Methods

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

using size_type = typename DS<DataVec>::size_type;
size_type is the size type

using TimeStamp = TS;
TimeStamp is the type of the index column

using TSVec = DS<TS>;
TSVec is the type of the vector containing the index column

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

template<typename F, typename ... types>
DataFrame bucketize (F &&func, const TimeStamp &bucket_interval) const;
It bucketizes the data and index into bucket_interval's, based on index values and calls the functor for each bucket. The result of each bucket will be stored in a new DataFrame with same shape and returned. Every data bucket is guaranteed to be as wide as bucket_interval. This mean some data items at the end may not be included in the new bucketized DataFrame. The index of each bucket will be the last index in the original DataFrame that is less than bucket_interval away from the previous bucket

NOTE: The DataFrame must already be sorted by index.

F: type functor to be applied to columns to bucketize
types: List of the types of all data columns.
    A type should be specified in the list only once.
bucket_interval: Bucket interval is in the index's single value unit.
    For example, if index is in minutes, bucket_interval
    will be in the unit of minutes and so on.
already_sorted: If the DataFrame is already sorted by index,
    this will save the expensive sort operation

template<typename F, typename ... types>
std::future<DataFrame>
bucketize_async (F &&func, const TimeStamp &bucket_interval) const;
Same as bucketize() above, but executed asynchronously

template<typename F, typename ... types>
void self_bucketize (F &&func, const TimeStamp &bucket_interval);
This is exactly the same as bucketize() above. The only difference is it stores the result in itself and returns void. So, after the return the original data is lost and replaced with bucketized data

NOTE: The DataFrame must already be sorted by index.

template<typename S, typename ... types>
bool write (S &o, bool values_only = false) const;
It outputs the content of DataFrame into the stream o as text in the following format:

```
INDEX:<Comma delimited list of values>
<Column1 name>:<Column1 type>:<Comma delimited list of values>
<Column2 name>:<Column2 type>:<Comma delimited list of values>
```

S: Output stream type
types: List all the types of all data columns.
    A type should be specified in the list only once.
o: Reference to an streamable object (e.g. cout)
values_only: If true, the name and type of each column is not written

template<typename S, typename ... Ts>
std::future<bool> write_async (S &o, bool values_only = false) const;
Same as write() above, but executed asynchronously

bool read (const char *file_name);
It inputs the contents of a text file into itself (i.e. DataFrame). The format of the file must be:

    INDEX:<Comma delimited list of values>
    <Column1 name>:<Column1 type>:<Comma delimited list of values>
    <Column2 name>:<Column2 type>:<Comma delimited list of values>

All empty lines or lines starting with # will be skipped.
file_name: Complete path to the file

std::future<bool> read_async (const char *file_name);
Same as read() above, but executed asynchronously

template<typename T>
DS<T> &get_column (const char *name);
It returns a reference to the container of named data column
T: Data type of the named column

template<typename T>
const DS<T> &get_column (const char *name) const;
It returns a const reference to the container of named data column
T: Data type of the named column

template<typename ... types>
DataFrame get_data_by_idx (TS begin, TS end) const;
It returns a DataFrame (including the index and data columns) containing the data from index begin to index end this function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.
types: List all the types of all data columns.
    A type should be specified in the list only once.

template<typename ... types>
DataFrame get_data_by_loc (size_type begin, size_type end) const;
It returns a DataFrame (including the index and data columns) containing the data from location begin to location end this function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.
types: List all the types of all data columns.
    A type should be specified in the list only once.

const TSVec &get_index () const { return (timestamps_); }
It returns a const reference to the index container

```
TSVec &get_index ()  { return (timestamps_); }
```
It returns a reference to the index container

```
template<typename ... Ts>
void multi_visit (Ts ... args) const;
```
This is the most generalized visit function. It visits multiple columns with the corresponding function objects sequentially. Each function object is passed every single value of the given column along with its name and the corresponding index value. All functions objects must have this signature

```
        bool (const TimeStamp &i, const char *name, const T &col_value)
```
If the function object returns false, the DataFrame will not go on that column.
Ts: The list of types for columns in args
args: A variable list of arguments consisting of
```
      std::pair(<const char *name,
                &std::function<bool (const TimeStamp &,
                                     const char *, const T &)>).
```
Each pair represents a column name and the functor to run on it.
NOTE: The second member of pair is a _pointer_ to the function or functor object

```
template<typename T, typename V>
V &visit (const char *name, V &visitor) const;
```
It passes the values of each index and each named column to the functor visitor sequentially from beginning to end
T: Type of the named column
V: Type of the visitor functor
name: Name of the data column

```
template<typename T1, typename T2, typename V>
V &&visit (const char *name1, const char *name2, V &&visitor) const;
```
It passes the values of each index and the two named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column
T2: Type of the second named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column

```
template<typename T1, typename T2, typename T3, typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
           V &&visitor) const;
```
It passes the values of each index and the three named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column

T2: Type of the second named column
T3: Type of the third named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column

```
template<typename T1, typename T2, typename T3, typename T4, typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
           const char *name4,
           V &&visitor) const;
```

It passes the values of each index and the four named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column
T2: Type of the second named column
T3: Type of the third named column
T4: Type of the forth named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column
name4: Name of the forth data column

```
template<typename T1, typename T2, typename T3, typename T4, typename T5,
         typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
           const char *name4,
           const char *name5,
           V &&visitor) const;
```

It passes the values of each index and the five named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column
T2: Type of the second named column
T3: Type of the third named column
T4: Type of the forth named column
T5: Type of the fifth named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column
name4: Name of the forth data column
name5: Name of the fifth data column

template<typename ... types>
bool is_equal (const DataFrame &rhs) const;

It compares self with rhs. If both have the same indices, same number of columns, same names for each column, and all columns are equal, then it returns true. Otherwise it returns false

types: List all the types of all data columns.
    A type should be specified in the list only once.

template<typename ... types>
DataFrame &modify_by_idx (DataFrame &rhs, bool already_sorted = false);

It iterates over all indices in rhs and modifies all the data columns in self that correspond to the given index value. If not already_sorted, both rhs and self will be sorted by index. It returns a reference to self

types: List all the types of all data columns.
    A type should be specified in the list only once.

already_sorted: If the self and rhs are already sorted by index,
    this will save the expensive sort operations