gtable Documentation

Release 1.1.0

Guillem Borrell Nogueras

Oct 21, 2018

Contents:

Gtable (source) is a container for tabular or tabular-like data designed with speed in mind. It is very similar to pandas, and it relies on many of its capabilities. It tries to improve one some aspects of using pandas data types `pandas.Series` and `pandas.DataFrame` as containers for simple computations, but it is not a replacement for them.

- It tries to reduce the overhead for column access, creation, and concatenation.
- It supports sparse data with bitmap indexing.
- It truly handles NaNs, making a difference between a NaN and a NA in its internal representation.
- It provides fast transformations (filling NA values, filtering, joining...)

It also relies heavily on numpy. You can consider gtable as a thin layer over numpy arrays.

CHAPTER 1

Tutorial

CHAPTER 2

Motivation

One important issue that beomes evident if you is use the Pandas Dataframe very often is that, while it's a terrific class for data analysis, it's that it's not a very good container for data. This notebook is a short explanation on why one may want to reduce an hypothetical intensive use of the Pandas Dataframe and to explore some other solutions. None of this is a criticism about Pandas. It is a game changer, and I am a strong advocate for its use. It's just that we hit one of its limitations. Some simple operations just have too much overhead.

To be more precise, let's start by importing Pandas

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'a': np.arange(1E6), 'b': np.arange(1E6)})
```

We have just created a relatively large dataframe with some dummy data, enough to prove my initial point. Let's see how much time it takes to add the two columns and to insert the result into the third one.

```
%%timeit
df.c = df.a + df.b
```

```
3.01 ms ± 20.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Is that fast or slow? Well, let's try to make the very same computation in a slightly different manner

```
a = df.a.values
b = df.b.values
```

```
%%timeit
c = a + b
```

```
2.86 ms ± 12.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

If we compare how fast it is to a simple sum of two numpy arrays, it is pretty fast. But we are adding two relatively large arrays. Let's try the exact same thing with smaller arrays.

```
df = pd.DataFrame({'a': np.arange(100), 'b': np.arange(100)})
```

```
%%timeit
df.c = df.a + df.b
```

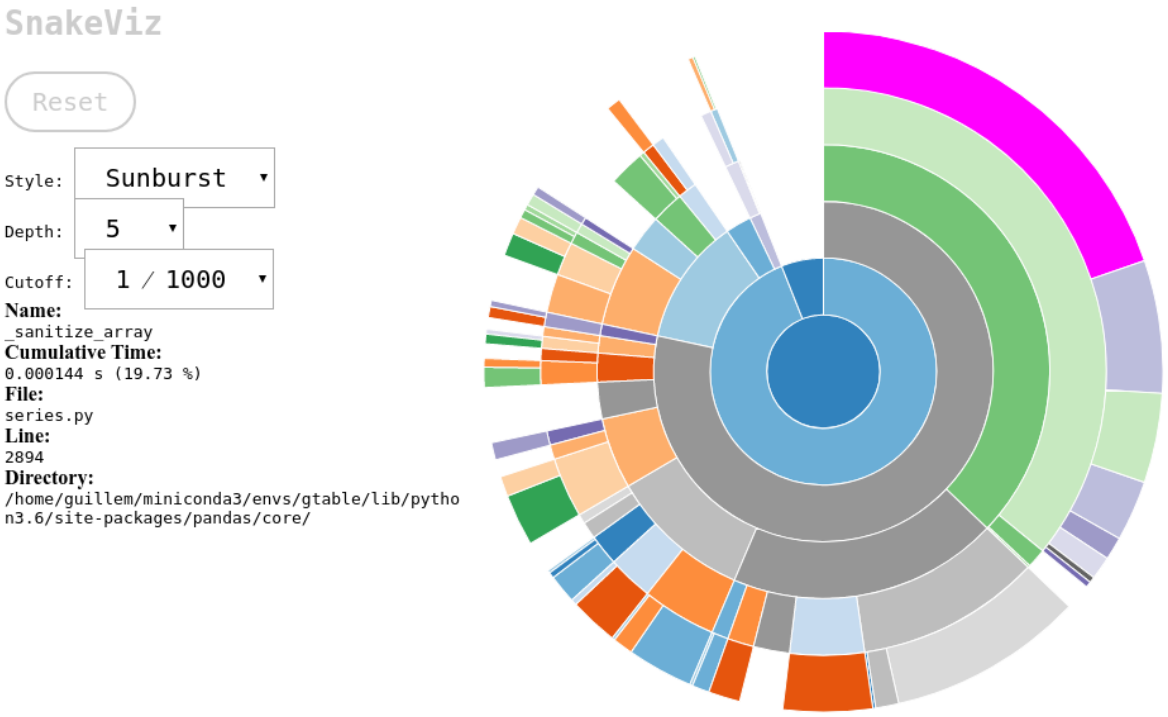
```
95.5 µs ± 114 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
a = df.a.values
b = df.b.values
```

```
%%timeit
c = a + b
```

```
599 ns ± 3.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Now things have changed quite a lot. Just adding two arrays takes two orders of magnitude less than adding from the Pandas Dataframe. But this comparison is not fare at all. Those 145µs are not spent waiting. Pandas does lots of things with the value of the Series resulting from the sum before it inserts it to the dataframe. If we profile the execution of that simple sum, we'll see that almost a fifth of the time is spent on a function called `_sanitize_array`.



The most important characteristic of Pandas is that it always does what it is supposed to do with data regardless of how dirty, heterogeneous, sparse (you name it) your data is. And it does an amazing job with that. But the price we have to pay are those two orders of magnitude in time.

That is exactly what impacted the performance of our last project. The Dataframe is a very convenient container because it always does something that makes sense, therefore you have to code very little. For instance, take the `join` method of a dataframe. It does just what it has to do, and it is definitely not trivial. Unfortunately, that overhead is too much for some use cases.

We are in the typical situation where abstractions are not for free. The higher the level, the slower the computation. This is a kind of a *second law of Thermodynamics* applied to numerical computing. And there are abstractions that are tremendously useful. A Dataframe is not a dictionary of arrays. It can be indexed by row and by column, and it can operate as a whole, and on any imaginable portion of it. It can sort, group, joing, merge... You name it. But if you want to compute the payment schedule of all the securities of an entire bank, you may need thousands of processors to have it done in less than six hours.

This is where I started thinking. There must be something in between. Something that is fast, but it's not just a dictionary of numpy arrays. And I started designing gtable

```
from gtable import Table

tb = Table({'a': np.arange(1E6), 'b': np.arange(1E6)})
```

```
%%timeit
tb.c = tb.a + tb.b
```

```
3.16 ms ± 11.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

You can see that for large arrays, the computation time shadows the overhead. Let's see how well it does with smaller arrays

```
tb = Table({'a': np.arange(100), 'b': np.arange(100)})
```

```
%%timeit
tb.c = tb.a + tb.b
```

```
8.51 µs ± 437 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

We have improved by a factor of 10, which is crucial if that's the difference between running in one or ten servers. We can still improve the computation by a little bit more if we fallback into some kind of *I know what I am doing* mode, and we want to reuse memory to avoid allocations:

```
%%timeit
tb['a'] = tb['a'] + tb['b']
```

```
2.17 µs ± 117 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Now the performance of arithmetic operations with gtable is closer to operate with plain arrays to the overhead-driven performance of Pandas. You can seriously break the table if you really don't know what you are doing. But for obvious reasons, having this kind of performance tricks is sometimes neessary.

Of course, these speedups come at a cost: features. Gtable is in its infancy. It is a small module that one can hack easily. It is pure python, and I have not started to seriously tune its performance. But the idea of having something inbetween a Dataframe and a dictionary of arrays with support for sparse information is appealing to say the list.

CHAPTER 3

Gtable and Pandas

Let's start by creating a table

```
from gtable import Table
import numpy as np
import pandas as pd
```

```
t = Table()
```

```
t.a = np.random.rand(10)
t.b = pd.date_range('2000-01-01', freq='M', periods=10)
t.c = np.array([1,2])
t.add_column('d', np.array([1, 2]), align='bottom')
```

You can create a column by assignment to an attribute. You can also use the `add_column` method if the default alignment is not the one you want. The usual representation of the table gives information about the actual length of each column and its type.

```
t
```

```
<Table[ a[10] <float64>, b[10] <object>, c[2] <int64>, d[2] <int64> ] object at␣
↪0x7f4f0eae68d0>
```

You can translate the table to a Pandas dataframe by just calling the `to_pandas` method, and leverage the great notebook visualization of the Dataframe

```
df = t.to_pandas()
df
```

Now that we have the same data stored as a Table and as a Dataframe, let's see some of the differences between them. The first one is that while the DataFrame has an index (an integer just keeps the order in this case), the Table is just a table trivially indexed by the order of the records

```
df.index.values
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Another important difference how data is stored in each container.

```
df.c.values
```

```
array([  1.,   2.,  nan,  nan,  nan,  nan,  nan,  nan,  nan,  nan])
```

```
t.c.values
```

```
array([1, 2])
```

While Pandas relies on NaN to store empty values, the Table uses a bitmap index to differentiate between a missing element and a NaN

```
t.index
```

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]], dtype=uint8)
```

The mechanism for tracking NAs is the bitmap index. Of course, a bitmap index has pros and cons. One of the interesting pros is that computations with sparse data are significantly faster, while keeping data indexed.

```
df.c.values
```

```
array([  1.,   2.,  nan,  nan,  nan,  nan,  nan,  nan,  nan,  nan])
```

```
t.c.values
```

```
array([1, 2])
```

The main benefit of the Table class is that both assignment and computation with sparse data is significantly faster. It operates with less data, and it does not have to deal with the index

```
%%timeit
2*t['c']
```

```
1.63 µs ± 200 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%%timeit
2*df['c']
```

```
73.6 µs ± 5.77 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The amount of features of the Dataframe dwarfs the ones present in the Table. But that does not mean that the Table is completely feature-less, or that the features are slow. Table allows to filter the data in a similar fashon to the Dataframe with slightly better performance.

```
%%timeit
df[df.c>0]
```

```
474 µs ± 89.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
df[df.c>0]
```

```
%%timeit
t.filter(t.c > 0)
```

```
131 µs ± 2.15 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
t.filter(t.c > 0).to_pandas()
```

See that, as Table sees that there have not been results for the fourth column, the generated dataframe omits that column.

One of the consequences of the Table's mechanism of indexing is that data cannot be accessed through the index, and there is no such thing as the Dataframe's iloc. If we extract the data of the column and we assign a value to one of its items, we may get the result we want.

```
t['c'][1] = 3
t.filter(t.c > 0).to_pandas()
```

But we cannot assign an element that does not exist

```
#t['c'][9]
```

Since the data of that column only has two elements

```
t['c']
```

```
array([1, 3])
```

Up to this point we have created the Dataframe from the table, but we can make the conversion the other way round

```
t1 = Table.from_pandas(df)
t1
```

```
<Table[ idx[10] <int64>, a[10] <float64>, b[10] <datetime64[ns]>, c[10] <float64>,␣
↪d[10] <float64> ] object at 0x7f4ee2ae1c18>
```

See that some datatypes have changed, and the sparsity of the table is lost, since Pandas cannot distinguish between NA and NaN. Note also that another column has been added with the index information. If we already know that all NaN are in fact NA, we can recover the sparse structure with

```
t1.dropnan()
```

```
t1
```

```
<Table[ idx[10] <int64>, a[10] <float64>, b[10] <datetime64[ns]>, c[2] <float64>,␣
↪d[2] <float64> ] object at 0x7f4ee2ae1c18>
```

We can recover the types casting the columns, that are numpy arrays. To restore the original columns we can also delete the index

```
t1['c'] = t1['c'].astype(np.int)
t1['d'] = t1['d'].astype(np.int)
t1.del_column('idx')
```

```
t1
```

```
<Table[ a[10] <float64>, b[10] <datetime64[ns]>, c[2] <int64>, d[2] <int64> ]␣
↪object at 0x7f4ee2ae1c18>
```

CHAPTER 4

Indexed or unindexed columns

One important feature of the Table container is the ability to compute arithmetic operations with and without taking the index into account. Using the indexed or the unindexed operation is left as a choice for the user. This notebook tries to explain the differences, and the consequences of using each option. We'll start by creating a sparse Table.

```
from gtable import Table
t = Table()
t.add_column('a', [1,2,3,4,5,6])
t.add_column('b', [1,2,3], align="bottom")
```

If we access the column by attribute, we'll get a type called `Column`, that includes information about the index

```
t.a
```

```
<Column[ int64 ] object at 0x7f5348736c50>
```

Accessing by key is a shortcut to the data stored within the `Table`, and has no information about how the table is indexed.

```
t['a']
```

```
array([1, 2, 3, 4, 5, 6])
```

The column `a` is not a particularly good example, but `b` is. The data stored in the latter column has only three elements. Where those elements are actually placed within the table is stored in the index.

```
t['b']
```

```
array([1, 2, 3])
```

The easiest and safest way to operate with columns is to take the index into account

```
c = t.a + t.b
```

```
c.values
```

```
array([ 2.,  3.,  4.])
```

See that, since the `b` column only had three elements, the result of the addition with the `a` column has only three elements. There are no NaNs or NAs. However, using the index to perform arithmetic operations has some cost, particularly in the case of large dense columns. Assume that we want to scale the `a` column by the last element of `b`. We can do that either accessing the full column or by accessing the raw data

```
c = t.a * t.b[-1]
```

```
c.values
```

```
array([ 3,  6,  9, 12, 15, 18])
```

Using columns is more convenient, since in many cases arithmetic operations do what they are supposed to do, but they have an important caveat: performance:

```
%%timeit
t.a * t.b[-1]
```

```
26.2 µs ± 224 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%%timeit
t['a'] * t['b'][-1]
```

```
6.19 µs ± 88.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

But since the data of each column has a different length, using the raw data or the colum will have different outcomes

```
t.a + t.b
```

```
<Column[ float64 ] object at 0x7f531fc411d0>
```

```
t['a'] + t['b']
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-12-2fb36be086ed> in <module>()
----> 1 t['a'] + t['b']


ValueError: operands could not be broadcast together with shapes (6,) (3,)
```

A caveat of columns is that they are designed to perform fast operations using the column as a whole, and in consequence, accessing individual item of a column is O(N).

Another important difference is that we can create new columns by attribute, but not by index

```
t.c = t.a + t.b
```

```
t
```

```
<Table[ a[6] <int64>, b[3] <int64>, c[3] <float64> ] object at 0x7f5348736fd0>
```

```
t['d'] = t['a']
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-16-46490dc9bd03> in <module>()
----> 1 t['d'] = t['a']


~/projects/gtable/gtable/table.py in __setitem__(self, key, value)
    158
    159     def __setitem__(self, key, value):
--> 160         self.data[self.keys.index(key)] = value
    161
    162     def __delitem__(self, key):


ValueError: 'd' is not in list
```

CHAPTER 5

API

class gtable.**Table**(*data={}*)

Table is a class for fast columnar storage using a bitmap index for sparse storage

add_column(*k*, *v*, *dtype=None*, *index=None*, *align='top'*)

Column concatenation.

copy()

Returns a copy of the table

crop(*key*)

Purge the records where the column key is empty

del_column(*k*)

Column deletion

dropnan(*clip=False*)

Drop the NaNs and leave missing values instead

fill_column(*key*, *fillvalue*)

Fill N/A elements in the given columns with fillvalue

Parameters

- **key** – String, list or tuple with the column names to be filled.
- **fillvalue** – Scalar to fill the N/A elements

Returns

fillna_column(*key*, *reverse=False*, *fillvalue=None*)

Fillna on a column inplace

Parameters

- **key** – string or list
- **reverse** –
- **fillvalue** –

Returns

filter(*predicate*)

Filter table using a column specification or predicate

first_record(*fill=False*)
Returns the first record of the table

static from_chunks(*chunks*)
Create a table from table chunks

Parameters **chunks** –

Returns

classmethod from_pandas(*dataframe*)
Create a table from a pandas dataframe

get(*key*, *copy=False*)
Gets a column or a table with columns

last_record(*fill=False*)
Returns the last record of the table

merge(*table*, *column*)
Merge two tables using two dense and sorted columns

records(*fill=False*)
Generator that returns a dictionary for each row of the table

reduce_by_key(*column*, *check_sorted=False*)
Reduce by key

Parameters

- **column** –
- **check_sorted** –

Returns

rename_column(*old_name*, *new_name*)
Rename a column of the table

Parameters

- **old_name** –
- **new_name** –

Returns

required_column(*key*, *dtype*)
Enforce the required column with a dtype

Parameters

- **key** –
- **dtype** –

Returns

required_columns(**args*)
Enforce the required columns. Create empty columns if necessary.

Parameters **args** –

Returns

sieve(*idx*)
Filter table using a one-dimensional array of boolean values

sort_by(*column*)
Sorts by values of a column

stack(*table*)
Vertical (Table) concatenation.

to_dict()
Translate the table to a dict {key -> array_of_values}

to_pandas(*fill=False*)
Translate the table to a pandas dataframe

class gtable.**Column**(*values*, *index*)
Indexed column view of the table

astype(*dtype*)
Changes the (numpy) datatype of the values

contains(*item*)
Returns a column with the value of the column present in item.

Parameters **item** –

Returns

copy()
Return a copy of the column

date_range(*fr='1970-01-01'*, *to='2262-01-01'*, *include_fr=True*, *include_to=True*)
Filter a column by date range.

Parameters

- **fr** –
- **to** –
- **include_fr** –
- **include_to** –

Returns

dtype
Returns the datatype of the column

fill(*fillvalue*)
Fills the N/A values of the column with the fillvalue :param fillvalue: :return:

fillna(*reverse=False*, *fillvalue=None*)
Fills the non available value sequentially with the previous available position. Operates inplace.

is_empty()
True if the column is empty

Returns

is_sorted()
True if the column is sorted :return:

mask(*mask*)
Apply mask on data to the data and the index out of place :param mask: :return:

reindex(*index*)
Reindex according to a global index

Parameters **index** –

Returns

reorder(*order*)
Reorder the column inplace :param order: :return:

CHAPTER 6

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

You can install gtable with a simple `pip install gtable`.

Gtable is an open-source project released under a BSD 3-Clause license. You can find a copy of the license in this document

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index