Improving Astropy Table performance

Contact Information:

Name: Anany Shrey Jain

University : Birla Institute of Technology and Science, Pilani, Goa.

Short Bio: Undergraduate Student of B.E Mechanical Engineering.

Email-Id : f20170920g@goa.bits-pilani.ac.in

GitHub username : ananyashreyjain

(https://github.com/ananyashreyjain)

Time-Zone : UTC + 5:30 .

Personal Details:

I am Anany Shrey Jain, pursuing my first year of Undergraduate from Birla Institute of Technology, Goa . Technology has always intrigued me, though formally I was first introduced with Java in 10th standard. From then I have learned python,C++ and C, but python is what I prefer the most when it comes to coding or programming.I have been working in python for more than half a year now and in the time being I have explored many libraries and worked on many projects. Ever since I was introduced to Python, I liked it far better than any other language. The reason is that one can think more about converting ideas to working code rather than wrestling around with the nuances of the language.

Pull Requests:

I stumbled upon Astropy while searching for astronomy related projects on GitHub, sometime around 20th November 2017.I have attempted to fix issues and understand Astropy codebase since then

1. https://github.com/astropy/astropy/pull/7103 : The fast C reader used to read astropy Tables cannot handle unicode so this fix allows pure-Python reader to be used in case of unicode.
2. https://github.com/astropy/astropy/pull/7024 : This PR focuses on fixing a bug as well as adding a feature.
   Table with columns of strings couldn't be written to HDF5 , this PR focuses on fixing it. This PR also adds an option to convert the Table columns to unicode or to leave them as bytes when read from HDF5 file.
3. https://github.com/astropy/astropy/pull/7033 : Adds a function that converts any quantity to human readable form. (For example: 100000000000 m to 100 Gm.).

Project Proposal:

Introduction

The astropy table sub-package has a core Table class that is used to store and manipulate tabular data within astropy. This class was written with an emphasis on functionality, convenience for astronomers, and code clarity. With the astropy table package now fairly mature and with a strong set of regression tests in place, it is time to focus on performance for basic operations like table creation and slicing. For a simple operation like slicing, astropy Table is currently about a factor of 10 slower than Pandas. This project will focus on identifying performance bottlenecks, writing performance tests for astropy-benchmarks, and then developing code to improve the performance.

Table Creation:

Initializing a Table object and then extending it by adding columns to it is a very slow process. For even a small dataset it is takes a lot of time . For example: In the following benchmarking code a Table object is created and then 1000 columns having same data are added one by one to it.

```
def time_range(self):
    self.t=Table()
    self.c=np.arange(0,10)
      for i in range(0,1000):
            self.t[str(i)]=self.c
```

- Time taken: 7.02 secs.

One the other hand if the columns are stored in a list and then a Table is created from that list it takes a lot less time.

```
def time_range(self):
    self.c=[np.arange(0,10) for col in range(0,1000)]
    self.n=(str(i) for i in range(0,1000))
    self.t=Table(self.c,names=self.n)
```

- Time taken: 67.96 ms.

It is quite evident that using `add_columns()` to extend the Table is slowing down the process of creation of Table using first method.

Table Slicing:

In slicing a Table each column of the data table is sliced one after the other and then each sliced column is added to a new table . So as the number of columns increases the time of execution increases rapidly . For example: In the following benchmarking code 1000 columns and only 10 rows are stored in the table which is then sliced to half of its length.

```python
def time_range(self):
    self.c=[np.arange(0,10) for col in range(0,1000)]
    self.n=(str(i) for i in range(0,1000))
    self.t=Table(self.c,names=self.n)[0:5]
```

- Time taken (without slicing): 69.03 ms
- Time taken(with slicing): 125.02 ms

Increasing the number of rows in the Table does not affect the time of slicing much . For example even if the code mentioned above has a Table with 1000 rows and 1000 columns and then the Table is sliced to half of its length , there was just 3 ms increase in time of execution of code.

So it is quite evident that slicing each column one by one is not that efficient method of doing the job.

Deliverables:

I. Examine ASV Benchmarks of Astropy related to Tables:

Reading and understanding of basic documentation of ASV already done . Some of the Astropy's ASV Benchmarks have also been examined.

II. Profile parts of code related to Table creation and slicing.

Already profiled many parts of code . Some of the code snippets are posted above.

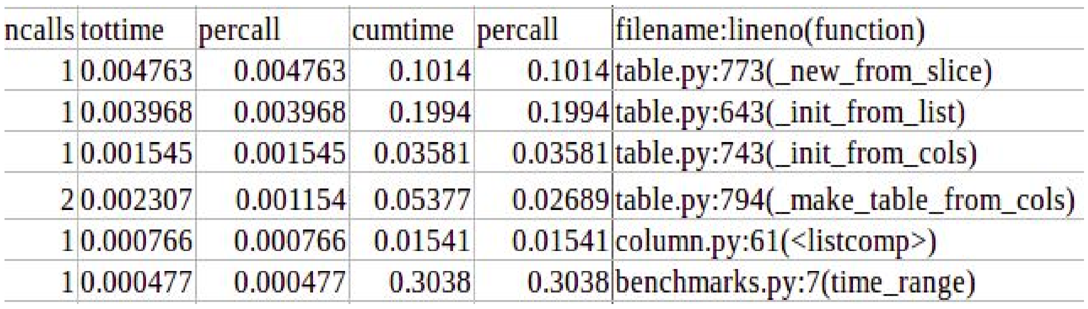
III. Identify all possible performance bottlenecks related to basic operations of Tables.

Some bottlenecks may be like the function `_new_from_slice()` which takes the maximum time in execution during a slicing operation.

A. <u>Benchmarking code:</u>

```python
    def time_range(self):
        self.c=[np.arange(0,1000) for col in range(0,1000)]
        self.n=(str(i) for i in range(0,1000))
        self.t=Table(self.c,names=self.n)[0:500]
```

B. <u>Profiling Results:</u>



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.004763	0.004763	0.1014	0.1014	table.py:773(_new_from_slice)
1	0.003968	0.003968	0.1994	0.1994	table.py:643(_init_from_list)
1	0.001545	0.001545	0.03581	0.03581	table.py:743(_init_from_cols)
2	0.002307	0.001154	0.05377	0.02689	table.py:794(_make_table_from_cols)
1	0.000766	0.000766	0.01541	0.01541	column.py:61(<listcomp>)
1	0.000477	0.000477	0.3038	0.3038	benchmarks.py:7(time_range)

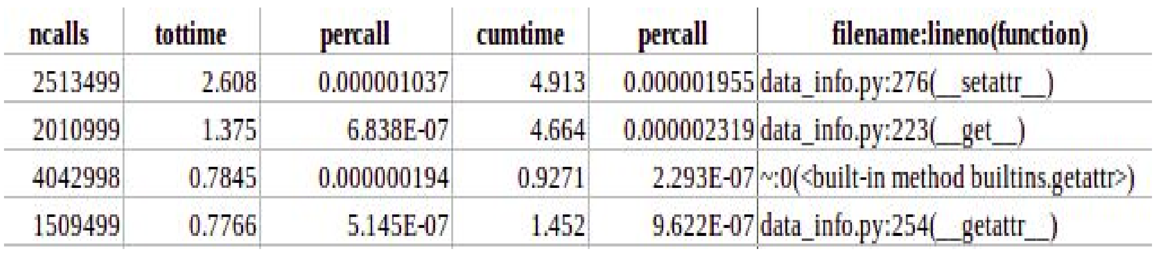
IV. Work out plans to fix the bottlenecks.

It will not only include fixing the bottlenecks that are slowing down the whole pipeline but also searching for parts of code that are needlessly executed and can be excluded in many cases . For example during the execution of following code:

```python
        def time_range(self):
            self.t=Table()
            self.c=np.arange(0,10)
            for i in range(0,1000):
                self.t[str(i)]=self.c
```

`__setattr__` is repeatedly called for all the columns in the Table even though it is not required for the already existing columns in the Table . This unnecessary increase in the number of method calls adversely affects the time of execution of code . In Fact as the size of the Table increases each column is added at comparatively slower pace than the previous one . This causes exponential increase in time of creation of Tables with increase in number of columns in the Table.

Profiling Results:



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2513499	2.608	0.000001037	4.913	0.000001955	data_info.py:276(__setattr__)
2010999	1.375	6.838E-07	4.664	0.000002319	data_info.py:223(__get__)
4042998	0.7845	0.000000194	0.9271	2.293E-07	~:0(<built-in method builtins.getattr>)
1509499	0.7766	5.145E-07	1.452	9.622E-07	data_info.py:254(__getattr__)

Number of calls : 2513499.
Total time : 2.608 seconds.
(number of columns : 1000).

On the other hand a similar code for creation of tables using pandas took much less time to get executed .

Benchmarking code:

```python
    def time_range(self):
        self.c = np.arange(0, 10)
        self.t = pd.DataFrame.from_records([])
        for i in range(0, 1000):
            self.t[str(i)] = self.c
```

Time taken: 0.446 secs

In case of pandas there were comparatively very few method calls and that is the reason why the above code takes a lot less time to get executed.
So either the execution time of such methods or the number of method calls needs to be reduced .

V. Implement the changes to fix the bottlenecks.

Once the plans are made to remove the bottlenecks , implementation of those plans could begin.

VI. Work on improving other basic operations of Tables.

Once both Table creation and Table slicing are optimized some other elementary Table function can also be optimized.

Timeline:

Time Period	**Plan**
23 April - 14 May Week 1 - Week 3	- Read the documentation of Astropy Tables. - Learn more about ASV Benchmarking. - Examine ASV Benchmarks of Astropy related to Tables - Discuss with mentor the ideas to search and fix possible bottlenecks.
14 May - 24 May Week 3 - Week 4.5	- Profile parts of code related to Table creation. - Search for possible bottlenecks in the pipeline of Table creation .
25 May - 10 June Week 4.5 - Week 6.5	- Workout plans to fix the bottlenecks. - Implement the plans to fix the bottlenecks. - Write ASV benchmarks for Table creation using different methods.
11 June - 15 June (5 day of Evaluations) Week 6.5 - Week 7.5	- Solve any bugs encountered in above steps. - Ask for feedback from mentors and the Astropy community regarding the protocol. - Complete any pending work.
16 June - 26 June Week 7.5 - Week 9	- Profile parts of code related to Table slicing. - Search for possible bottlenecks in the pipeline of Table slicing.
27 June - 8 July Week 9 - Week 10.5	- Workout plans to fix the bottlenecks. - Implement the plans to fix the bottlenecks. - Write ASV benchmarks for Table slicing using different ways.

9 July - 13 July (5 day of Evaluations) Week 10.5 - Week 11	- Solve any bugs encountered in above steps. - Complete any pending work.
14 July - 20 July Week 11 - Week 12	- Select Table operations that need boost in performance. - Profile parts of code related to those operations. - Study ASV Benchmarks related to those operations
20 July - 5 August Week 12 - Week 14	- Find bottlenecks in the code and devise plans to fix them. - Implement the plans so as to fix the bottlenecks. - Write ASV benchmarks if required .
6 August - 14 August Week 14 - Week 15	- Fix any bugs that arose in above steps. - Clean up the code (make it PEP8 compliant). - Complete any leftover work.
14 August - 21 August	- Mentors submit final student evaluations.
22 August	- Final results of Google Summer of Code 2018 are announced.

Schedule Information:

I do not plan to work elsewhere, nor do I plan to go on a vacation during this period.I will be having my second semester final exams during the period 1st May - 14th May. Most of work scheduled during this time is already done . My academic load would be minimal after that . Nevertheless, I would be able to devote as much time as needed for GSoC.