Working Efficiently with Big Data in Text Formats

David Mertz, Ph.D.
Continuum Analytics
dmertz@continuum.io

Big Data in Text Formats

Who am I?

Senior trainer for Continuum Analytics, since I started there in May 2015. Sometimes still a software developer there also.

… the company that makes the Anaconda Python distribution, Bokeh, Numba, Dask, Blaze, and many other FLOSS technologies.

… co-founder, Travis Oliphant created NumPy, for example.

… Jeff Reback is lead developer of Pandas.

… Lots more amazing folks to work with.

Big Data in Text Formats

Things I've done:

Director of the Python Software Foundation for 6 years; chair of PSF Outreach & Education Committee; chair of PSF Trademarks Committee; chair of PSF/NumFOCUS Scientific Python Working Group; chair of PSF Python-Cuba WG.

Wrote the IBM developerWorks column Charming Python; the Addison-Wesley book Text Processing in Python; some short books on Python for O'Reilly; and various related articles.

For 8 years worked for a research lab, D. E. Shaw Research, who built the world's fastest supercomputer, Anton, for doing molecular dynamics.

Big Data in Text Formats

IOPro

IOPro loads NumPy arrays (and Pandas DataFrames) directly from files, SQL databases, and NoSQL stores–including ones with millions of rows–without creating millions of temporary, intermediate Python objects, or requiring expensive array resizing operations. It provides a drop-in replacement for the NumPy functions `loadtxt()` *and* `genfromtxt()`*, but dramatically improves performance and reduces memory overhead.*

Getting IOPro Now

The usual procedure for installing IOPro works. I.e.

```
conda install iopro
```

Until Continuum determines the open source licensing terms for TextAdapter, IOPro nags minimally during the trial period.

We'll sort out the details soon, and we hope this becomes meaningfully a community project (as many of our past products have: Numba, Dask, conda, etc.)

Big Data in Text Formats

TextAdapter

`iopro.text_adapter()` *provides a fast interface to textual data. This may be CSV (or other delimiters), fixed-width, JSON, optionally compressed. Within the interface, extended slicing is supported, and slices produce NumPy arrays. Data values and types can be manipulated within the adapter itself; this is similar in concept to virtual columns in a relational database. The location of textual data can be either on local filesystems or in S3 buckets.*

TextAdapter Indexing

The problem with reading delimited textual data sources is that, since records are of varying sizes the only way to find a given record is to do a linear scan.

If we read from near the start of the data, there is not so much of a problem. If we read data near the end, we have very slow performance; moreover, the larger the data source is, the worse the performance.

Big Data in Text Formats

TextAdapter in Action (1)

```
from keys import aws_access_key, aws_secret_key, bucket
from conversions import datatypes

key_name = 'PlanAttributes.csv' # 77k lines, 95MB
adapter = iopro.s3_text_adapter(
                        aws_access_key, aws_secret_key,
                        bucket, key_name,
                        index_name='PlanAttributes.index')
adapter.set_field_types(datatypes)
arr = adapter[70000:70005]
```

Big Data in Text Formats

TextAdapter Speeding Up S3 (1)

```
% time ./read_plans.py; time ./read_plans.py
S3 Read PlanAttributes.csv
real    1m2.877s
user    0m3.551s
sys     0m2.085s

S3 Read PlanAttributes.csv
real    0m15.678s
user    0m1.160s
sys     0m0.462s
```

TextAdapter Speeding Up S3 (2)

A 4x speedup on network-limited reads isn't bad. S3 delivers its requests in blocks, so even when using an index, we need to read an entire block then seek within it.

Still, the larger the data source, the bigger the speedup. Our test was only 95MB and 77k lines.

We will see that on local filesystems, we should be able to do direct seeks instead, and get bigger improvements.

Big Data in Text Formats

TextAdapter in Action (2)

```
# 13M lines, 2GB raw, 110MB compressed
filename = 'Rate.csv.gz'
adapter = iopro.text_adapter(filename,
                             compression='gzip',
                             parser='csv',
                             index_name='Rates.index')
row = int(1e7)
arr = adapter[row:row+5]
```

Big Data in Text Formats

TextAdapter Speeding Up Local Access (1)

```
% time ./read_rates.py; time ./read_rates.py
Local Read Rate.csv.gz
real    0m16.992s
user    0m16.748s
sys     0m0.222s
```

```
Local Read Rate.csv.gz
real    0m0.943s
user    0m0.809s
sys     0m0.133s
```

TextAdapter Speeding Up Local Access (2)

With the index created, we can actually access records near the end *faster* than the raw rate of gunzip'ing it. Moreover, the code shown did more than the raw decompression—it created a NumPy array of the rows of interest, with data typing of fields.

```
% time gunzip -c Rate.csv.gz | wc -l
 12694446
real    0m2.031s
```

Specifying Data Types (1)

We cheated a little in the above examples. The type inference mechanism in *iopro* isn't (yet) as smart as it should be. Moreover, we unfortunately currently raise an exception where type inference fails. I glossed over this line earlier:

```
adapter.set_field_types(datatypes)
```

We can manually specify data types of fields if we wish. This can be desirable to give bit width even where the basic type is inferred correctly.

Specifying Data Types (2)

In order to smooth over problems, I gave a large number of the 176 fields in `PlanAttributes.csv` the *Object* data type. Any field could also be a byte array of sufficient length (but "sufficient" is a property of the full dataset).

```
datatype = {0:'f4', 5:'O', 6:'O', 43:'O', 44:'O', …}
```

While type 'O' always succeeds, it provides the least useful information. A specific numeric type is much more useful.

Big Data in Text Formats

Specifying Data Types and Selecting Columns

Let's see what we can do with `.set_field_types()`.

```
# Rate.csv.gz
>>> adapter.size
12694445
>>> %time adapter[['IssuerId',
...                'BusinessYear','Couple']][int(9e6)]
CPU times: user 78.7 ms, sys: 26.2 ms, total: 105 ms
Wall time: 107 ms
array([('2016', 68398, 0)], dtype=[('BusinessYear', 'O'),
  ('IssuerId', '<u8'), ('Couple', '<u8')])
```

Big Data in Text Formats

Specifying Data Types (3)

We read the 9 millionth row rather quickly using the index. However, data types are not optimal (nor correct for other rows).

```
>>> adapter.set_field_types({'BusinessYear':'u2', 16:'f4'})
>>> adapter[['IssuerId','BusinessYear','Couple']][2]
array([(2014, 21989, 73.9000015258789)],
      dtype=[('BusinessYear', '<u2'), ('IssuerId', '<u8'),
             ('Couple', '<f4')])
```

A year is efficiently stored as a 2-byte unsigned int; column 16 (*Couple*) should actually be a floating point number. We could use fewer bits for *IssuerId* too.

Converting Data Types (1)

Sometimes simply declaring a data type is enough. But in other cases, what is in a field takes some massaging to get a simple type. I noticed a few patterns sample values of the fields I had to cast to *Object* in `PlanAttributes.csv`.

```
Field 43 = "Yes"
Field 45 = "$16.67"
Field 48 = "100%"
Field 56 = "$0.00"
Field 62 = "Yes"
Field 63 = "70.92%"
```


Big Data in Text Formats

Converting Data Types (2)

Convert fields to simple scalars using *converter functions*.

```
>>> def yes_to_bool(s):
...     return s.lower() == "Yes"
>>> adapter.set_converter('DentalOnlyPlan', yes_to_bool)
>>> adapter.set_converter(63, lambda s:
                  float(s.decode('utf-8').replace('%',''))/100)
>>> adapter.set_field_types({43:'b', 63:'f2'})
>>> adapter[[43,'IssuerActuarialValue']][70001]
array([(0, 0.604003)], dtype=[('DentalOnlyPlan', 'i1'),
                              ('IssuerActuarialValue', '<f2')]
```

Converting Data Types (3)

Parsing data based on regular expressions rather than delimiters can often be a more efficient way to decompose the parts of records. Of course, that doesn't allow a fully general computed value (but it does allow selecting fields more precisely).

```
% cat transactions.csv  # Real world data!
$2.56, 50%, September 20 1978
$1.23, 23%, April 5 1981
```

Converting Data Types (3)

We only want some of the content in a line in this case, and know patterns for where the fields lie. Only the strings in regex groups are included as fields.

```
>>> regex_string = '\$(\d)\.(\d{2}),\s*'\
...                 '([0-9]+)\%,\s*([A-Za-z]+)'
>>> adapter = iopro.text_adapter('transactions.csv',
...                              parser='regex',
...                              regex_string=regex_string,
...                              field_names=False,
...                              infer_types=False)
```


Converting Data Types (4)

Setting the regex just selects the fields, we explicitly refrained from attempting type inference of the fields because we want to fine-tune the bit-lengths.

```
>>> adapter.set_field_types({0:'i2',1:'u2',2:'f4',3:'S10'})
>>> adapter.set_field_names(['dollars', 'cents',
...                          'percentage', 'month'])
>>> adapter[:]
array([(2, 56, 50.0, b'September'), (1, 23, 23.0, b'April')],
      dtype=[('dollars', '<i2'), ('cents', '<u2'),
             ('percentage', '<f4'), ('month', 'S10')])
```

Big Data in Text Formats

DataFrames

Assuming you have Pandas installed, you can convert slices from adapters into DataFrames. Of course, avoid doing so with huge datasets (the whole point is to avoid exhausting memory).

```
>>> 'adapter[:].to_dataframe()'  # Exists, but bad idea
>>> pd.DataFrame(adapter[:2])    # Slice just a bit
   dollars  cents  percentage          month
0        2     56        50.0  b'September'
1        1     23        23.0      b'April'
```

Big Data in Text Formats

Other stuff

TextAdapter has several features not addressed. IOPro as a whole also includes some database accelerators.

- It handles JSON
- Adapters follow strided and negative-index slicing
- Missing data can be filled in within the adapter
- Adapters can be introspected for attributes

In the rest of the talk, I want to present ideas for future features, especially once we release it under an open license.

Big Data in Text Formats

The Future of TextAdapter

I have some ideas for technical enhancements to IOPro, and I'd love to hear your recommendations during Q&A.

I'd also love to hear about or discuss listeners experiences or ideas about the balance between commercial companies like Continuum, that need to make money to continue what we work on, and open source licensing and participation in community projects.

TextAdapter 2.x: More complete indexing (1)

The index feature is strictly row indexing. This helps with seeks, but has nothing to do with the *content* of a data source.

Being able to do an operation akin to slicing that expresses something about the values in the data could be useful. NumPy and Pandas express this with predicative indexing, and SQL is a whole language for basically just this.

We cannot possibly build something better than dedicated RDBMS's (including sqlite3), but perhaps a few steps in that direction would be worthwhile?

Big Data in Text Formats

TextAdapter 2.x: More complete indexing (2)

What syntax and capabilities would be best to include (and most feasible)? In concept, a NumPy/Pandas syntax might work, but we cannot include generic predicates, so it might be misleading. What actual capabilities should indices have?

```
adapter[adapter[field] == value]                 # Feasible
adapter[adapter[field].str.lower() == value] # Maybe?
adapter[np.degrees(adapter[field]) < 45]     # No way!
```

Remember that the whole point of an index is to avoid a linear scan, so we need comparisons that can be hashed or simplified in some ways to identify rows lists of matches.

TextAdapter 2.x: More complete indexing (3)

The NumPy/Pandas predicative index syntax may or may not be the best choice. Regardless of the API of matching rows, what should be the options for *generating* an index?

Presumably the `.create_index()` method could allow additional signatures. Should we require every column desired be explicitly indexed by the user? Should there be several index styles with different capabilities? When doing a match request, should TextAdapter fail if no index is available or fall back to linear search?