Blaze: Building A Foundation for Array-Oriented Computing in Python

Mark Wiebe‡*, Matthew Rocklin‡, TJ Alumbaugh‡, Andy Terrel‡

http://www.youtube.com/watch?v=9HPR-1PdZUk

Abstract—We present the motivation and architecture of Blaze, a library for cross-backend data-oriented computation. Blaze provides a standard interface to connect users familiar with NumPy and Pandas to other data analytics libraries like SQLAlchemy and Spark. We motivate the use of these projects through Blaze and discuss the benefits of standard interfaces on top of an increasingly varied software ecosystem. We give an overview of the Blaze architecture and then demonstrate its use on a typical problem. We use the abstract nature of Blaze to quickly benchmark and compare the performance of a variety of backends on a standard problem.

Index Terms—array programming, big data, numpy, scipy, pandas

Introduction

Standard Interfaces

Software and user communities around data analysis have changed remarkably in the last few years. The growth in this ecosystem come both from new computational systems and also from an increasing breadth of users. On the software side we see activity in different languages like Python [Pyt14], R [RLa14], and Julia [Jul12], and also in distributed systems like the projects surrounding the Hadoop File System (HDFS) [Bor07]. On the user side we see increased adoption both from physical sciencists, with a strong tradition of computation, and also from social scientists and policy makers with less rigorous training. While these upward trends are encouraging, they also place significant strain on the programming ecosystem. Keeping novice users adapted to quickly changing programming paradigms and operational systems is challenging.

Standard interfaces facilitate interactions between layers of complex and changing systems. For example, NumPy fancy indexing syntax has become a standard interface among array programming systems within the Python ecosystem. Projects with very different implementations (e.g. NumPy [Van11], SciPy.sparse [Jon01], Theano [Ber10]), SciDB [Bro10]) all provide the same indexing interface despite operating very differently.

Standard interfaces help users to adapt to changing technologies without learning new programming paradigms. Standard interfaces help project developers by bootstrapping a well trained community of users. Uniformity smoothes adoption and allows the ecosystem to evolve rapidly without the drag of everyone having to constantly learn new technologies.

* *Corresponding author: mwiebe@continuum.io*
‡ *Continuum Analytics*

Interactive Arrays and Tables

Analysis libraries like NumPy and Pandas demonstrate the value of interactive array and table objects. Projects such as these connect a broad base of users to efficient low-level operations through a high-level interface. This approach has given rise to large and productive software ecosystems within numeric Python (e.g. SciPy, Scikits, etc.) However, both NumPy and Pandas are largely restricted to an in-memory computational model, limiting problem sizes to a certain scale.

Concurrently developed data analytic ecosystems in other languages like R and Julia provide similar styles of functionality with different application foci. The Hadoop File System (HDFS) has accrued a menagerie of powerful distributed computing systems such as Hadoop, Spark, and Impala. The broader scientific computing community has produced projects like Elemental and SciDB for distributed array computing in various contexts. Finally, traditional SQL databases such as MySQL and Postgres remain both popular and very powerful.

As problem sizes increase and applications become more interdisciplinary, analysts increasingly require interaction with projects outside of the NumPy/Pandas ecosystem. Unfortunately, these foreign projects rarely feel as comfortable or as usable as the Pandas DataFrame.

What is Blaze

Blaze provides a familiar interface around computation on a diverse set of computational systems, or backends. It provides extensible mechanisms to connect this interface to new computational backends. Backends which the Blaze project explicitly provides hooks to include Python, Pandas, SQLAlchemy, and Spark.

This abstract connection to a variety of projects has the following virtues:

- Novice users gain access to relatively exotic technologies
- Users can trivially shift computational backends within a single workflow
- Projects can trivially shift backends as technologies change
- New technologies are provided with a stable interface and a trained set of users

Blaze doesn't do any computation itself. Instead it depends heavily on existing projects to perform computations. Currently

Blaze covers tabular computations as might fit into the SQL or Pandas model of computation. We intend to extend this model to arrays and other highly-regular computational models in the future.

Related Work

We separate related work into two categories:

1) Computational backends useful to Blaze
2) Similar efforts in uniform interfaces

Computational backends on which Blaze currently relies include Pandas, SQLAlchemy, PyToolz, Spark, PyTables, NumPy, and DyND. Pandas [McK10] provides an efficient in-memory table object. SQLAlchemy [sqlal] handles connection to a variety of SQL dialects like SQLite and Postgres. PyToolz [Roc13] provides tuned functions for streaming computation on core Python data structures. NumPy [Van11] and DyND [Wie13] serve as in-memory arrays and common data interchange formats. PyTables [Alt03] provides efficient sequential computations on out-of-core HDF5 files.

Uniform symbolic interfaces on varied computational resources are also common. SQLAlchemy provides a uniform interface onto various SQL implementations. Theano [Ber10] maps array operations onto Python/NumPy, C, or CUDA code generation. While computer algebra projects like SymPy [Sym08] often have expression trees they also commonly include some form of code generation to low-level languages like `C`, `Fortran` but also to languages like `LaTeX` and `DOT` for visualization.

Blaze Architecture

Blaze separates data analytics into three isolated components:

- Data access: *access* data efficiently across different storage systems,
  e.g. `CSV`, `HDF5`, `HDFS`, ...
- Symbolic Expression: *reason* symbolically about the desired result,
  e.g. `Join`, `Sum`, `Split-Apply-Combine`, ...
- Backend Computation: *execute* computations on a variety of backends,
  e.g. `SQL`, `Pandas`, `Spark`, ...

We isolate these elements to enable experts to create well crafted solutions in each domain without needing to understand the others, e.g., a Pandas expert can contribute without knowing Spark and vice versa. Blaze provides abstraction layers between these components to enable them to work together cleanly.

The assembly of these components creates in a multi-format, multi-backend computational engine capable of common data analytics operations in a variety of contexts.

Blaze Data

Blaze Data Descriptors are a family of Python objects that provide uniform access to a variety of common data formats. They provide standard iteration, insertion, and NumPy-like fancy indexing over on-disk files in common formats like CSV, JSON, and HDF5 in memory data strutures like core Python data structures and NumPy arrays as well as more sophisticated data stores like SQL databases. The data descriptor interface is analogous to the Python buffer interface described in PEP 3118 [Oli06], but with a more flexible API.

Over the course of this article we'll refer to the following simple `accounts.csv` file:

```
id, name, balance
1, Alice, 100
2, Bob, -200
3, Charlie, 300
4, Denis, 400
5, Edith, -500
```

```
>>> from blaze import *
>>> csv = CSV('accounts.csv')  # Create data object
```

Iteration: Data descriptors expose the `__iter__` method, which provides an iterator over the outermost dimension of the data. This iterator yields vanilla Python objects by default.

```
>>> list(csv)
[(1L, u'Alice', 100L),
 (2L, u'Bob', -200L),
 (3L, u'Charlie', 300L),
 (4L, u'Denis', 400L),
 (5L, u'Edith', -500L)]
```

Data descriptors also expose a `chunks` method, which also iterates over the outermost dimension but instead of yielding single rows of Python objects instead yields larger chunks of compactly stored data. These chunks emerge as DyND arrays that are more efficient for bulk processing and data transfer. DyND arrays support the `__array__` interface and so can be easily converted to NumPy arrays.

```
>>> next(csv.chunks())
nd.array([[1, "Alice", 100],
          [2, "Bob", -200],
          [3, "Charlie", 300],
          [4, "Denis", 400],
          [5, "Edith", -500]],
 type="5 * {id : int64, name : string, balance : int64}")
```

Insertion: Analagously to `__iter__` and `chunks`, the methods `extend` and `extend_chunks` allow for insertion of data into the data descriptor. These methods take iterators of Python objects and DyND arrays respectively. The data is coerced into whatever form is native for the storage medium, e.g. text for CSV, or `INSERT` statements for SQL.

```
>>> csv = CSV('accounts.csv', mode='a')
>>> csv.extend([(6, 'Frank', 600),
...             (7, 'Georgina', 700)])
```

Migration: The combination of uniform iteration and insertion along with robust type coercion enables trivial data migration between storage systems.

```
>>> sql = SQL('postgresql://user:pass@host/',
              'accounts', schema=csv.schema)
>>> sql.extend(iter(csv))  # Migrate csv file to DB
```

Indexing: Data descriptors also support fancy indexing. As with iteration, this supports either Python objects or DyND arrays through the `.py[...]` and `.dynd[...]` interfaces.

```
>>> list(csv.py[::2, ['name', 'balance']])
[(u'Alice', 100L),
 (u'Charlie', 300L),
 (u'Edith', -500L),
 (u'Georgina', 700L)]

>>> csv.dynd[::2, ['name', 'balance']]
nd.array([["Alice", 100],
```

```
          ["Charlie", 300],
          ["Edith", -500],
          ["Georgina", 700]],
     type="var * {name : string, balance : int64}")
```

Performance of this approach varies depending on the underlying storage system. For file-based storage systems like CSV and JSON, it is necessary to seek through the file to find the right line (see [iopro]), but don't incur needless deserialization costs (i.e. converting text into floats, ints, etc.) which tend to dominate ingest times. Some storage systems, like HDF5, support random access natively.

Cohesion: Different storage techniques manage data differently. Cohesion between these disparate systems is accomplished with the two projects `datashape`, which specifies the intended meaning of the data, and DyND, which manages efficient type coercions and serves as an efficient intermediate representation.

Blaze Expr

To be able to run analytics on a wide variety of computational backends, it's important to have a way to represent them independent of any particular backend. Blaze uses abstract expression trees for this, including convenient syntax for creating them and a pluggable multiple dispatch mechanism for lowering them to a computation backend. Once an analytics computation is represented in this form, there is an opportunity to do analysis and transformation on it prior to handing it off to a backend, both for optimization purposes and to give heuristic feedback to the user about the expected performance.

To illustrate how Blaze expression trees work, we will build up an expression on a table from the bottom , showing the structure of the trees along the way. Let's start with a single table, for which we'll create an expression node

```
>>> accts = TableSymbol('accounts',
...       '{id: int, name: string, balance: int}')
```

to represent a abstract table of accounts. By defining operations on expression nodes which construct new abstract expression trees, we can provide a familiar interface closely matching that of NumPy and of Pandas. For example, in structured arrays and dataframes you can access fields as `accts['name']`.

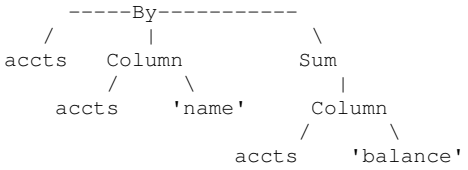
Extracting fields from the table gives us `Column` objects, to which we can now apply operations. For example, we can select all accounts with a negative balance.

```
>>> deadbeats = accts[accts['balance'] < 0]['name']
```

or apply the split-apply-combine pattern to get the highest grade in each class

```
>>> By(accts, accts['name'], accts['balance'].sum())
```

In each of these cases we get an abstract expression tree representing the analytics operation we have performed, in a form independent of any particular backend.



Blaze Compute

Once an analytics expression is represented as a Blaze expression tree, it needs to be mapped onto a backend. This is done by walking the tree using the multiple dispatch `compute` function, which defines how an abstract Blaze operation maps to an operation in the target backend.

To see how this works, let's consider how to map the `By` node from the previous section into a Pandas backend. The code that handles this is an overload of `compute` which takes a `By` node and a `DataFrame` object. First, each of the child nodes must be computed, so `compute` gets called on the three child nodes. This validates the provided dataframe against the `accts` schema and extracts the 'name' and 'balance' columns from it. Then, the pandas `groupby` call is used to group the 'balance' column according to the 'name' column, and apply the `sum` operation.

Each backend can map the common analytics patterns supported by Blaze to its way of dealing with it, either by computing it on the fly as the Pandas backend does, or by building up an expression in the target system such as an SQL statement or an RDD map and groupByKey in Spark.

Multiple dispatch provides a pluggable mechanism to connect new back ends, and handle interactions between different backends.

Example

We demonstrate the pieces of Blaze in a small toy example.

Recall our accounts dataset

```
>>> L = [(1, 'Alice',   100),
         (2, 'Bob',    -200),
         (3, 'Charlie', 300),
         (4, 'Denis',   400),
         (5, 'Edith',  -500)]
```

And our computation for names of account holders with negative balances

```
>>> deadbeats = accts[accts['balance'] < 0]['name']
```

We compose the abstract expression, `deadbeats` with the data `L` using the function `compute`.

```
>>> list(compute(deadbeats, L))
['Bob', 'Edith']
```

Note that the correct answer was returned as a list.

If we now store our same data `L` into a Pandas DataFrame and then run the exact same `deadbeats` computation against it, we find the same semantic answer.

```
>>> df=DataFrame(L, columns=['id', 'name', 'balance'])
>>> compute(deadbeats, df)
1      Bob
4    Edith
Name: name, dtype: object
```

Similarly against Spark

```
>>> sc = pyspark.SparkContext('local', 'Spark-app')
>>> rdd = sc.parallelize(L) # Distributed DataStructure

>>> compute(deadbeats, rdd)
PythonRDD[1] at RDD at PythonRDD.scala:37

>>> _.collect()
['Bob', 'Edith']
```

In each case of calling `compute(deadbeats, ...)` against a different data source, Blaze orchestrates the right computational backend to execute the desired query. The result is given in the form received and computation is done either with streaming Python, in memory Pandas, or distributed memory Spark. The user experience is identical in all cases.

Blaze Interface

The separation of expressions and backend computation provides a powerful multi-backend experience. Unfortunately, this separation may also be confusing for a novice programmer. To this end we provide an interactive object that feels much like a Pandas DataFrame, but in fact can be driving any of our backends.

```
>>> sql = SQL('postgresql://postgres@localhost',
...           'accounts')
>>> t = Table(sql)
>>> t
   id     name  balance
0   1    Alice      100
1   2      Bob     -200
2   3  Charlie      300
3   4    Denis      400
4   5    Edith     -500

>>> t[t['balance'] < 0]['name']
    name
0    Bob
1  Edith
```

The astute reader will note the use of Pandas-like user experience and output. Note however, that these outputs are the result of computations on a Postgres database.

Discussion

Blaze provides both the ability to migrate data between data formats and to rapidly prototype common analytics operations against a wide variety of computational backends. It allows one to easily compare options and choose the best for a particular setting. As that setting changes, for example when data size grows considerably, our implementation can transition easily to a more suitable backend.

This paper gave an introduction to the benefits of separating expression of a computation from its computation. We expect future work to focus on integrating new backends, extending to array computations, and composing Blaze operations to transform existing in-memory backends like Pandas and DyND into an out-of-core and distributed setting.

REFERENCES

[Zah10] Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

[McK10] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[sqlal] http://www.sqlalchemy.org/

[iopro] http://docs.continuum.io/iopro/index.html

[Roc13] Rocklin, Matthew and Welch, Erik and Jacobsen, John. *Toolz Documentation*, 2014 http://toolz.readthedocs.org/

[Wie13] Wiebe, Mark. *LibDyND* https://github.com/ContinuumIO/libdynd

[Sym08] SymPy Development Team. "SymPy: Python library for symbolic mathematics." (2008).

[Ber10] Bergstra, James, et al. "Theano: a CPU and GPU math compiler in Python." Proc. 9th Python in Science Conf. 2010.

[Bor07] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." Hadoop Project Website 11 (2007): 21.

[Alt03] Alted, Francesc, and Mercedes Fernández-Alonso. "PyTables: processing and analyzing extremely large amounts of data in Python." PyCon 2003 (2003).

[Van11] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011),

[Oli06] Oliphant, Travis and Banks, Carl. http://legacy.python.org/dev/peps/pep-3118/

[Pyt14] G. Van Rossum. The Python Language Reference Manual. Network Theory Ltd., September 2003.

[RLa14] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

[Jul12] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. CoRR, abs/1209.5145, 2012.

[Jon01] Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/ [Online; accessed 2014-09-25].

[Bro10] Paul G. Brown, Overview of sciDB: large scale array storage, processing and analysis, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, June 06-10, 2010, Indianapolis, Indiana, USA