Blaze Documentation

Release 0.11.3

Continuum

Nov 18, 2017

Contents

The Blaze Ecosystem provides Python users high-level access to efficient computation on inconveniently large data. Blaze can refer to both a particular *library* as well as an *ecosystem* of related projects that have spun off of Blaze development.

Blaze is sponsored primarily by Continuum Analytics, and a DARPA XDATA grant.

Parts of the Blaze ecosystem are described below:

Several projects have come out of Blaze development other than the Blaze project itself.

- The Blaze Project: Translates NumPy/Pandas-like syntax to data computing systems (e.g. database, in-memory, distributed-computing). This provides Python users with a familiar interface to query data living in a variety of other data storage systems. One Blaze query can work across data ranging from a CSV file to a distributed database.

  Blaze presents a pleasant and familiar interface to us regardless of what computational solution or database we use (e.g. Spark, Impala, SQL databases, No-SQL data-stores, raw-files). It mediates our interaction with files, data structures, and databases, optimizing and translating our query as appropriate to provide a smooth and interactive session. It allows the data scientists and analyst to write their queries in a unified way that does not have to change because the data is stored in another format or a different data-store. It also provides a server-component that allows URIs to be used to easily serve views on data and refer to Data remotely in local scripts, queries, and programs.

- DataShape: A data type system

  DataShape combines NumPy's dtype and shape and extends to missing data, variable length strings, ragged arrays, and more arbitrary nesting. It allows for the common description of data types from databases to HDF5 files, to JSON blobs.

- Odo: Migrates data between formats.

  Odo moves data between formats (CSV, JSON, databases) and locations (local, remote, HDFS) efficiently and robustly with a dead-simple interface by leveraging a sophisticated and extensible network of conversions.

- DyND: In-memory dynamic arrays

  DyND is a dynamic ND-array library like NumPy that implements the datashape type system. It supports variable length strings, ragged arrays, and GPUs. It is a standalone C++ codebase with Python bindings. Generally it is more extensible than NumPy but also less mature.

- Dask.array: Multi-core / on-disk NumPy arrays

  Dask.dataframe : Multi-core / on-disk Pandas data-frames

  Dask.arrays provide blocked algorithms on top of NumPy to handle larger-than-memory arrays and to leverage multiple cores. They are a drop-in replacement for a commonly used subset of NumPy algorithms.

Dask.dataframes provide blocked algorithms on top of Pandas to handle larger-than-memory data-frames and to leverage multiple cores. They are a drop-in replacement for a subset of Pandas use-cases.

Dask also has a general "Bag" type and a way to build "task graphs" using simple decorators as well as nascent distributed schedulers in addition to the multi-core and multi-threaded schedulers.

These projects are mutually independent. The rest of this documentation is just about the Blaze project itself. See the pages linked to above for `datashape`, `odo`, `dynd`, or `dask`.

CHAPTER 1

Blaze

Blaze is a high-level user interface for databases and array computing systems. It consists of the following components:

- A symbolic expression system to describe and reason about analytic queries
- A set of interpreters from that query system to various databases / computational engines

This architecture allows a single Blaze code to run against several computational backends. Blaze interacts rapidly with the user and only communicates with the database when necessary. Blaze is also able to analyze and optimize queries to improve the interactive experience.

CHAPTER 2

Presentations

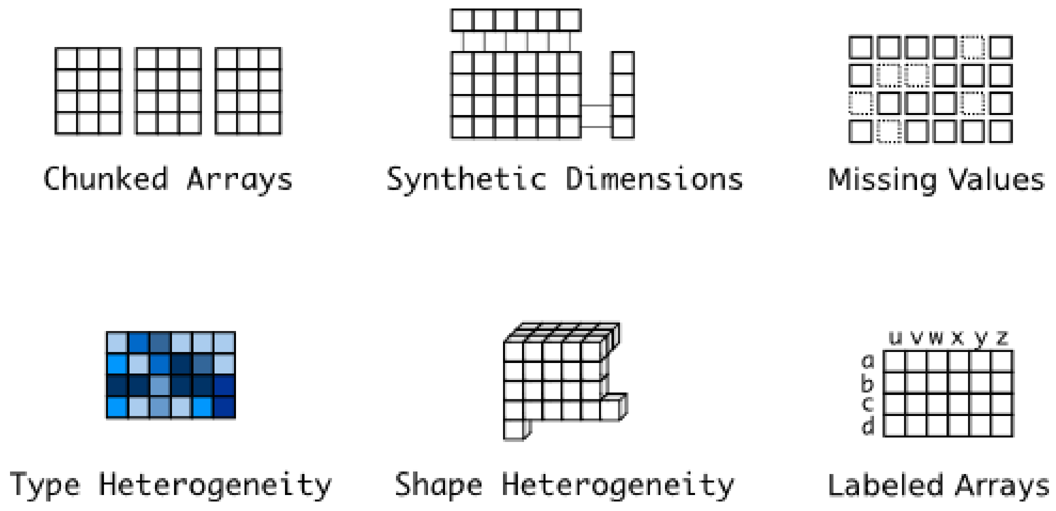
- See previous presentations about Blaze
- See previous blog posts about Blaze

2.1 Index

User facing

2.1.1 Overview

Blaze Abstracts Computation and Storage



Several projects provide rich and performant data analytics. Competition between these projects gives rise to a vibrant and dynamic ecosystem. Blaze augments this ecosystem with a uniform and adaptable interface. Blaze orchestrates computation and data access among these external projects. It provides a consistent backdrop to build standard interfaces usable by the current Python community.

Demonstration

Blaze separates the computations that we want to perform:

```
>>> from blaze import *
>>> accounts = symbol('accounts', 'var * {id: int, name: string, amount: int}')

>>> deadbeats = accounts[accounts.amount < 0].name
```

From the representation of data

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

Blaze enables users to solve data-oriented problems

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

But the separation of expression from data allows us to switch between different backends.

Here we solve the same problem using Pandas instead of Pure Python.

```
>>> df = DataFrame(L, columns=['id', 'name', 'amount'])

>>> compute(deadbeats, df)
1      Bob
4    Edith
Name: name, dtype: object
```

Blaze doesn't compute these results, Blaze intelligently drives other projects to compute them instead. These projects range from simple Pure Python iterators to powerful distributed Spark clusters. Blaze is built to be extended to new systems as they evolve.

Scope

Blaze speaks Python and Pandas as seen above and also several other technologies, including NumPy, SQL, Mongo, Spark, PyTables, etc.. Blaze is built to make connecting to a new technology easy.

Blaze currently targets database and array technologies used for analytic queries. It strives to orchestrate and provide interfaces on top of and in between other computational systems. We provide performance by providing data scientists with intuitive access to a variety of tools.

2.1.2 Install

Installing

Blaze can be most easily installed from conda

```
$ conda install blaze
```

More up-to-date builds are available on the `blaze` anaconda channel: http://anaconda.org/blaze

```
conda install -c blaze blaze
```

Blaze may also be installed using `pip`:

```
pip install blaze --upgrade
or
pip install git+https://github.com/blaze/blaze  --upgrade
```

If you are interested in the development version of Blaze you can obtain the source from Github.

```
$ git clone git@github.com:blaze/blaze.git
```

Anaconda can be downloaded for all platforms here: http://continuum.io/anaconda.html .

Introduction

To build project from source:

```
$ python setup.py install
```

To build documentation on a unix-based system:

```
$ cd docs
$ make docs
```

To run tests:

```
$ py.test --doctest-modules --pyargs blaze
```

Strict Dependencies

Blaze depends on NumPy, Pandas, and a few pure-python libraries. It should be easy to install on any Numeric Python setup.

- numpy >= 1.7
- datashape >= 0.4.4
- odo >= 0.3.1
- toolz >= 0.7.0
- cytoolz
- multipledispatch >= 0.4.7
- pandas

Optional Dependencies

Blaze can help you use a variety of other libraries like `sqlalchemy` or `h5py`. If these are installed then Blaze will use them. Some of these are non-trivial to install. We recommend installation throgh `conda`.

- sqlalchemy
- h5py
- spark >= 1.1.0
- pymongo
- pytables
- bcolz
- flask >= 0.10.1
- pytest (for running tests)

2.1.3 Quickstart

This quickstart is here to show some simple ways to get started created and manipulating Blaze Symbols. To run these examples, import blaze as follows.

```
>>> from blaze import *
```

Blaze Interactive Data

Create simple Blaze expressions from nested lists/tuples. Blaze will deduce the dimensionality and data type to use.

```
>>> t = data([(1, 'Alice', 100),
...           (2, 'Bob', -200),
...           (3, 'Charlie', 300),
...           (4, 'Denis', 400),
...           (5, 'Edith', -500)],
...          fields=['id', 'name', 'balance'])

>>> t.peek()
   id     name  balance
0   1    Alice      100
1   2      Bob     -200
2   3  Charlie      300
3   4    Denis      400
4   5    Edith     -500
```

Simple Calculations

Blaze supports simple computations like column selection and filtering with familiar Pandas getitem or attribute syntax.

```
>>> t[t.balance < 0]
   id   name  balance
0   2    Bob     -200
1   5  Edith     -500

>>> t[t.balance < 0].name
    name
0    Bob
1  Edith
```

Stored Data

Define Blaze expressions directly from storage like CSV or HDF5 files. Here we operate on a CSV file of the traditional iris dataset.

```
>>> from blaze.utils import example
>>> iris = data(example('iris.csv'))
>>> iris.peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
...
```

Use remote data like SQL databases or Spark resilient distributed data-structures in exactly the same way. Here we operate on a SQL database stored in a sqlite file.

```
>>> iris = data('sqlite:///%s::iris' % example('iris.db'))
>>> iris.peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
...
```

More Computations

Common operations like Joins and split-apply-combine are available on any kind of data

```
>>> by(iris.species,                # Group by species
...    min=iris.petal_width.min(),  # Minimum of petal_width per group
...    max=iris.petal_width.max())  # Maximum of petal_width per group
           species  max  min
0      Iris-setosa  0.6  0.1
1  Iris-versicolor  1.8  1.0
2   Iris-virginica  2.5  1.4
```

Finishing Up

Blaze computes only as much as is necessary to present the results on screen. Fully evaluate the computation, returning an output similar to the input type by calling `compute`.

```
>>> t[t.balance < 0].name                   # Still an Expression
    name
0    Bob
1  Edith

>>> list(compute(t[t.balance < 0].name))    # Just a raw list
['Bob', 'Edith']
```

Alternatively use the `odo` operation to push your output into a suitable container type.

```
>>> result = by(iris.species, avg=iris.petal_width.mean())
>>> result_list = odo(result, list)  # Push result into a list
>>> odo(result, DataFrame)  # Push result into a DataFrame
           species    avg
0      Iris-setosa  0.246
1  Iris-versicolor  1.326
2   Iris-virginica  2.026
>>> odo(result, example('output.csv'))  # Write result to CSV file
<odo.backends.csv.CSV object at ...>
```

2.1.4 Basic Queries

Here we give a quick overview of some of the more common query functionality.

We use the well known iris dataset

```
>>> from blaze import data
>>> from blaze.utils import example
>>> iris = data(example('iris.csv'))
>>> iris.peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
...
```

Column Access

Select individual columns using attributes

```
>>> iris.species
        species
0   Iris-setosa
1   Iris-setosa
2   Iris-setosa
3   Iris-setosa
...
```

Or item access

```
>>> iris['species']
        species
0   Iris-setosa
1   Iris-setosa
2   Iris-setosa
3   Iris-setosa
...
```

Select many columns using a list of names

```
>>> iris[['sepal_length', 'species']]
    sepal_length      species
0            5.1  Iris-setosa
1            4.9  Iris-setosa
2            4.7  Iris-setosa
3            4.6  Iris-setosa
...
```

Mathematical operations

Use mathematical operators and functions as normal

```
>>> from blaze import log
>>> log(iris.sepal_length * 10)
    sepal_length
```

```
0      3.931826
1      3.891820
2      3.850148
3      3.828641
...
```

Note that mathematical functions like `log` should be imported from `blaze`. These will translate to `np.log`, `math.log`, `sqlalchemy.sql.func.log`, etc. based on the backend.

Reductions

As with many Blaze operations reductions like `sum` and `mean` may be used either as methods or as base functions.

```
>>> iris.sepal_length.mean()
5.84333333333333...

>>> from blaze import mean
>>> mean(iris.sepal_length)
5.84333333333333...
```

Split-Apply-Combine

The `by` operation expresses split-apply-combine computations. It has the general format

```
>>> by(table.grouping_columns, name_1=table.column.reduction(),
...                            name_2=table.column.reduction(),
...                            ...)
```

Here is a concrete example. Find the shortest, longest, and average petal length by species.

```
>>> from blaze import by
>>> by(iris.species, shortest=iris.petal_length.min(),
...                   longest=iris.petal_length.max(),
...                   average=iris.petal_length.mean())
           species  average  longest  shortest
0      Iris-setosa    1.462      1.9       1.0
1  Iris-versicolor    4.260      5.1       3.0
2   Iris-virginica    5.552      6.9       4.5
```

This simple model can be extended to include more complex groupers and more complex reduction expressions.

Add Computed Columns

Add new columns using the `transform` function

```
>>> transform(iris, sepal_ratio = iris.sepal_length / iris.sepal_width,
...                 petal_ratio = iris.petal_length / iris.petal_width)
   sepal_length  sepal_width  petal_length  petal_width      species  \
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa

   sepal_ratio  petal_ratio
```

```
0      1.457143     7.000000
1      1.633333     7.000000
2      1.468750     6.500000
3      1.483871     7.500000
...
```

Text Matching

Match text with glob strings, specifying columns with keyword arguments.

```
>>> iris[iris.species.like('*versicolor')]
    sepal_length  sepal_width  petal_length  petal_width          species
50           7.0          3.2           4.7          1.4  Iris-versicolor
51           6.4          3.2           4.5          1.5  Iris-versicolor
52           6.9          3.1           4.9          1.5  Iris-versicolor
```

Relabel Column names

```
>>> iris.relabel(petal_length='PETAL-LENGTH', petal_width='PETAL-WIDTH')
    sepal_length  sepal_width  PETAL-LENGTH  PETAL-WIDTH      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
```

2.1.5 Examples

Blaze can help solve many common problems that data analysts and scientists encounter. Here are a few examples of common issues that can be solved using blaze.

Combining separate, gzipped csv files.

```
>>> from blaze import odo
>>> from pandas import DataFrame
>>> odo(example('accounts_*.csv.gz'), DataFrame)
   id     name  amount
0   1    Alice     100
1   2      Bob     200
2   3  Charlie     300
3   4      Dan     400
4   5    Edith     500
```

Split-Apply-Combine

```
>>> from blaze import data, by
>>> t = data('sqlite:///%s::iris' % example('iris.db'))
>>> t.peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
```

```
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
5           5.4          3.9           1.7          0.4  Iris-setosa
6           4.6          3.4           1.4          0.3  Iris-setosa
7           5.0          3.4           1.5          0.2  Iris-setosa
8           4.4          2.9           1.4          0.2  Iris-setosa
9           4.9          3.1           1.5          0.1  Iris-setosa
...
>>> by(t.species, max=t.petal_length.max(), min=t.petal_length.min())
           species  max  min
0      Iris-setosa  1.9  1.0
1  Iris-versicolor  5.1  3.0
2   Iris-virginica  6.9  4.5
```

2.1.6 Split-Apply-Combine – Grouping

Grouping operations break a table into pieces and perform some reduction on each piece. Consider the `iris` dataset:

```
>>> from blaze import data, by
>>> from blaze.utils import example
>>> d = data('sqlite:///%s::iris' % example('iris.db'))
>>> d
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
```

We find the average petal length, grouped by species:

```
>>> by(d.species, avg=d.petal_length.mean())
           species    avg
0      Iris-setosa  1.462
1  Iris-versicolor  4.260
2   Iris-virginica  5.552
```

Split-apply-combine operations are a concise but powerful way to describe many useful transformations. They are well supported in all backends and are generally efficient.

Arguments

The `by` function takes one positional argument, the expression on which we group the table, in this case `d.species`, and any number of keyword arguments which define reductions to perform on each group. These must be named and they must be reductions.

```
>>> by(grouper, name=reduction, name=reduction, ...)
```

```
>>> by(d.species, minimum=d.petal_length.min(),
...               maximum=d.petal_length.max(),
...               ratio=d.petal_length.max() - d.petal_length.min())
           species  maximum  minimum  ratio
0      Iris-setosa      1.9      1.0    0.9
```

```
1  Iris-versicolor      5.1       3.0     2.1
2   Iris-virginica      6.9       4.5     2.4
```

Limitations

This interface is restrictive in two ways when compared to in-memory dataframes like `pandas` or `dplyr`.

1. You must specify both the grouper and the reduction at the same time
2. The "apply" step must be a reduction

These restrictions make it *much* easier to translate your intent to databases and to efficiently distribute and parallelize your computation.

Things that you can't do

So, as an example, you can't "just group" a table separately from a reduction

```
>>> groups = by(mytable.mycolumn)  # Can't do this
```

You also can't do non-reducing apply operations (although this could change for some backends with work)

```
>>> groups = by(d.A, result=d.B / d.B.max())  # Can't do this
```

2.1.7 Pandas to Blaze

This page maps pandas constructs to blaze constructs.

Imports and Construction

```
import numpy as np
import pandas as pd
from blaze import data, by, join, merge, concat

# construct a DataFrame
df = pd.DataFrame({
   'name': ['Alice', 'Bob', 'Joe', 'Bob'],
   'amount': [100, 200, 300, 400],
   'id': [1, 2, 3, 4],
})

# put the `df` DataFrame into a Blaze Data object
df = data(df)
```

Computation	Pandas	Blaze
Column Arithmetic	`df.amount * 2`	`df.amount * 2`
Multiple Columns	`df[['id', 'amount']]`	`df[['id', 'amount']]`
Selection	`df[df.amount > 300]`	`df[df.amount > 300]`
Group By	`df.groupby('name').amount.mean()` `df.groupby(['name', 'id']).amount.mean()`	`by(df.name, amount=df.amount.mean())` `by(merge(df.name, df.id), amount=df.amount.mean())`
Join	`pd.merge(df, df2, on='name')`	`join(df, df2, 'name')`
Map	`df.amount.map(lambda x: x + 1)`	`df.amount.map(lambda x: x + 1, 'int64')`
Relabel Columns	`df.rename(columns={'name': 'alias', 'amount': 'dollars'})`	`df.relabel(name='alias', amount='dollars')`
Drop duplicates	`df.drop_duplicates()` `df.name.drop_duplicates()`	`df.distinct()` `df.name.distinct()`
Reductions	`df.amount.mean()` `df.amount.value_counts()`	`df.amount.mean()` `df.amount.count_values()`
Concatenate	`pd.concat((df, df))`	`concat(df, df)`
Column Type Information	`df.dtypes` `df.amount.dtype`	`df.dshape` `df.amount.dshape`

Blaze can simplify and make more readable some common IO tasks that one would want to do with pandas. These examples make use of the odo library. In many cases, blaze will able to handle datasets that can't fit into main memory, which is something that can't be easily done with pandas.

```
from odo import odo
```

Operation	Pandas	Blaze
Load directory of CSV files	`df = pd.concat([pd.read_csv(filename) for filename in glob.glob('path/to/*.csv')])`	`df = data('path/to/*.csv')`
Save result to CSV file	`df[df.amount < 0].to_csv('output.csv')`	`odo(df[df.amount < 0], 'output.csv')`

Read from SQL database	`df = pd.read_sql('select * from t', con='sqlite:///db.db')`
	 `df = pd.read_sql('select * from t', con=sa.create_engine('sqlite:///db.db'))`
		`df = data('sqlite:///db.db::t')`

2.1.8 SQL to Blaze

This page maps SQL expressions to blaze expressions.

Note: The following SQL expressions are somewhat specific to PostgreSQL, but blaze itself works with any database for which a SQLAlchemy dialect exists.

Prerequisites

If you're interested in testing these against a PostgreSQL database, make sure you've executed the following code in `psql` session:

```
CREATE TABLE df (
    id BIGINT,
    amount DOUBLE PRECISION,
    name TEXT
);
```

On the blaze side of things, the table below assumes the following code has been executed:

```
>>> from blaze import symbol, by, join, concat
>>> df = symbol('df', 'var * {id: int64, amount: float64, name: string}')
```

Note: Certain SQL constructs such as window functions don't directly correspond to a particular Blaze expression. `Map` expressions are the closest representation of window functions in Blaze.

Computation	SQL	Blaze
Column Arithmetic	`select amount * 2 from df`	`df.amount * 2`
Multiple Columns	`select id, amount from df`	`df[['id', 'amount']]`
Selection	`selelct * from df where amount > 300`	`df[df.amount > 300]`
Group By	`select avg(amount) from df group by name`	`by(df.name, amount=df.amount.mean())`

	`select avg(amount) from df group by name, id`	`by(merge(df.name, df.id),`
		`   amount=df.amount.mean())`

Join	`select * from`
	`    df inner join df2`
	`on df.name = df2.name`
		`join(df, df2, 'name')`

Map	`select amount + 1 over () from df`	`df.amount.map(lambda x: x + 1,`
		`              'int64')`

Relabel Columns	`select`
	`    id,`
	`    name as alias,`
	`    amount as dollars`
	` from df`
		`df.relabel(name='alias',`
		`           amount='dollars')`

Drop duplicates	`select distinct * from df`	`df.distinct()`
	`select distinct(name) from df`	`df.name.distinct()`

	`/* postgresql only */`
	`select distinct on (name) * from`
	`df order by name`	

Reductions	`select avg(amount) from df`	`df.amount.mean()`

	`select amount, count(amount)`
	`from df group by amount`
		`df.amount.count_values()`

Concatenate	`select * from df`
	`union all`
	`select * from df`
		`concat(df, df)`

2.1.9 URI strings

Blaze uses strings to specify data resources. This is purely for ease of use.

Example

Interact with a set of CSV files or a SQL database

```
>>> from blaze import *
>>> from blaze.utils import example
>>> t = data(example('accounts_*.csv'))
>>> t.peek()
   id     name  amount
0   1    Alice     100
1   2      Bob     200
2   3  Charlie     300
3   4      Dan     400
4   5    Edith     500

>>> t = data('sqlite:///%s::iris' % example('iris.db'))
>>> t.peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
...
```

Migrate CSV files into a SQL database

```
>>> from odo import odo
>>> odo(example('iris.csv'), 'sqlite:///myfile.db::iris')
Table('iris', MetaData(bind=Engine(sqlite:///myfile.db)), ...)
```

What sorts of URIs does Blaze support?

- **Paths to files on disk, including the following extensions**
  - `.csv`
  - `.json`
  - `.csv.gz/json.gz`
  - `.hdf5` (uses `h5py`)
  - `.hdf5::/datapath`
  - `hdfstore://filename.hdf5` (uses special `pandas.HDFStore` format)
  - `.bcolz`
  - `.xls(x)`

- **SQLAlchemy strings like the following**
  - `sqlite:////absolute/path/to/myfile.db::tablename`
  - `sqlite:////absolute/path/to/myfile.db` (specify a particular table)
  - `postgresql://username:password@hostname:port`
  - `impala://hostname` (uses `impyla`)
  - *anything supported by SQLAlchemy*
- **MongoDB Connection strings of the following form**
  - `mongodb://username:password@hostname:port/database_name::collection_name`
- **Blaze server strings of the following form**
  - `blaze://hostname:port` (port defaults to 6363)

In all cases when a location or table name is required in addition to the traditional URI (e.g. a data path within an HDF5 file or a Table/Collection name within a database) then that information follows on the end of the URI after a separator of two colons `::`.

How it works

Blaze depends on the Odo library to handle URIs. URIs are managed through the `resource` function which is dispatched based on regular expressions. For example a simple resource function to handle `.json` files might look like the following (although Blaze's actual solution is a bit more comprehensive):

```
from blaze import resource
import json

@resource.register('.+\.json')
def resource_json(uri):
    with open(uri):
        data = json.load(uri)
    return data
```

Can I extend this to my own types?

Absolutely. Import and extend `resource` as shown in the "How it works" section. The rest of Blaze will pick up your change automatically.

2.1.10 Tips for working with CSV files

How to

Typically one provides a csv filename to the `data` constructor like so

```
>>> d = data('myfile.csv')
```

GZip extensions or collections of csv files are handled in the same manner.

```
>>> d = data('myfile-2014-01-*.csv.gz')
```

In the case of collections of CSV files the files are sorted by filename and then considered to be concatenated into a single table.

How does it work?

Blaze primarily relies on Pandas to parse CSV files into DataFrames. In the case of large CSV files it may parse them into several DataFrames and then use techniques laid out in the *Out of Core Processing* section.

What to do when things go wrong

The same thing that makes CSV files so popular with humans, simple readability/writability, makes them challenging for computers to reason about robustly.

Interacting with CSV files often breaks down in one of two ways

1. We incorrectly guess the dialect of the CSV file (e.g. wrong delimiter, presence or absense of a header, ...)
2. We incorrectly guess the type of a column with the CSV file (e.g. an integer column turns out to have floats in it)

Because Blaze operates in a lazy way, giving you access to large CSV files without reading the entire file into memory it is forced to do some guesswork. By default it guesses the dialect and types on the first few hundred lines of text. When this guesswork fails the user must supply additional information.

Correcting CSV Dialects

In the first case of incorrect guessing of CSV dialect (e.g. delimiter) Blaze respects and passes through all keyword arguments to pandas.read_csv.

Note: In the case of a CSV file with all string data, you must pass the `has_header=True` argument if the first row is the header row.

Correcting Column Types

In the second case of incorrect guessing of column types Blaze accepts a *Datashape* as an additional keyword argument. Common practice is to create a `data` object around a csv file, ask for its datashape, tweak that datashape and then recreate the data object.

```
>>> d = data('myfile.csv')
>>> d
Exception: Integer column has NA values

>>> d.dshape  # Perhaps that integer column should be a float
dshape("var * {name: string, amount: int64}")

# <Copy-Paste>
>>> ds = dshape("var * {name: string, amount: float64}")  # change int to float

>>> d = data('myfile.csv', dshape=ds)
```

Migrate to Binary Storage Formats

If you plan to reuse the same CSV files many times it may make sense to convert them to an efficient binary store like HDF5 (common) or BColz (less common but faster). These storage formats provide better performance on your data and also avoid the ambiguity that surrounds CSV files.

One can migrate from CSV files to a binary storage format using the `odo` function.

```
>>> from odo import odo
>>> odo('myfiles-*.csv', 'myfile.bcolz')

# or

>>> odo('myfiles-*.csv', 'myfile.hdf5::/mydataset')

# or

>>> odo('myfiles-*.csv', 'sqlite:///mydb.db::mytable')
```

When migrating from a loosely formatted system like CSV to a more strict system like HDF5 or BColz there are a few things to keep in mind

1. Neither supports variable length strings well
2. But each supports fixed-length strings well and supports compression to cover up overly large/wasteful fixed-lengths
3. HDF5 does not support datetimes well but can easily encode datetimes as strings
4. BColz is a column store, offering much better performance on tables with many columns
5. HDF5 is a standard technology with excellent library support outside of the Python ecosystem

To ensure that you encode your dataset appropriately we recommend passing a datashape explicitly. As in our previous example this can often be done by editing automatically generated datashapes

```
>>> d = data('myfile.csv')
>>> d.dshape
dshape("var * {name: string, amount: int64}")

# <Copy-Paste>
>>> ds = dshape("var * {name: string[20, 'ascii'], amount: float64}")

>>> from odo import odo
>>> odo('myfiles-*.csv', 'myfile.bcolz', dshape=ds)
```

Providing a datashape removes data type ambiguity from the transfer.

2.1.11 Interacting with SQL Databases

How to

Typically one provides a SQL connection string to the `data` constructor

```
>>> db = data('postgresql:///user:pass@hostname')

or

>>> t = data('postgresql://user:pass@hostname::my-table-name')
```

Alternatively users familiar with SQLAlchemy can pass any SQLAlchemy engine, metadata, or Table objects to `data`. This can be useful if you need to specify more information that does not fit comfortably into a URI (like a desired schema.)

```
>>> import sqlalchemy
>>> engine = sqlalchemy.create_engine('postgreqsql://hostname')

>>> db = data(engine)
```

How does it work?

As you manipulate a Blaze expression Blaze in turn manipulates a SQLAlchemy expression. When you ask for a result SQLAlchemy generates the SQL appropriate for your database and sends the query to the database to be run.

What databases does Blaze support?

Blaze derives all SQL support from SQLAlchemy so really one should ask, *What databases does SQLAlchemy support?*. The answer is *quite a few* in the main SQLAlchemy project and *most* when you include third party libraries.

However, URI support within Blaze is limited to a smaller set. For exotic databases you may have to create a `sqlalchemy.engine` explicitly as shown above.

What operations work on SQL databases?

Most tabular operations, but not all. SQLAlchemy translation is a high priority. Failures include array operations like slicing and dot products don't make sense in SQL. Additionally some operations like datetime access are not yet well supported through SQLAlchemy. Finally some databases, like SQLite, have limited support for common mathematical functions like `sin`.

How can I try this out?

The easiest way to play with SQL is to download a SQLite database. We recommend the Lahman baseball statistics database. After downloading one could connect blaze to that database with the following code

```
>>> from blaze import data
>>> db = data('sqlite:///Downloads/lahman2013.sqlite')
>>> db.<tab>  # see available tables
db.AllstarFull          db.FieldingOF           db.Schools              db.fields
db.Appearances          db.FieldingPost         db.SchoolsPlayers       db.isidentical
db.AwardsManagers       db.HallOfFame           db.SeriesPost           db.like
db.AwardsPlayers        db.Managers             db.Teams                db.map
db.AwardsShareManagers  db.ManagersHalf         db.TeamsFranchises      db.relabel
db.AwardsSharePlayers   db.Master               db.TeamsHalf            db.schema
db.Batting              db.Pitching             db.apply                db.temp
db.BattingPost          db.PitchingPost         db.data
db.Fielding             db.Salaries             db.dshape
>>> db.Teams.peek()   # view one particular database
    yearID lgID teamID franchID divID  Rank   G  Ghome   W   L      ...        \
0     1871   NA    BS1      BNA  None     3  31    NaN  20  10      ...
1     1871   NA    CH1      CNA  None     2  28    NaN  19   9      ...
2     1871   NA    CL1      CFC  None     8  29    NaN  10  19      ...
3     1871   NA    FW1      KEK  None     7  19    NaN   7  12      ...

    DP    FP                     name                               park   \
0  NaN  0.83      Boston Red Stockings                South End Grounds I
1  NaN  0.82   Chicago White Stockings            Union Base-Ball Grounds
2  NaN  0.81    Cleveland Forest Citys       National Association Grounds
```

```
3  NaN  0.80     Fort Wayne Kekiongas                    Hamilton Field

    attendance  BPF  PPF  teamIDBR  teamIDlahman45  teamIDretro
0          NaN  103   98       BOS             BS1          BS1
1          NaN  104  102       CHI             CH1          CH1
2          NaN   96  100       CLE             CL1          CL1
3          NaN  101  107       KEK             FW1          FW1
...
```

One can then query and compute results as with a normal blaze workflow.

Connecting to a Schema Other than `public` with PostgreSQL

To connect to a non-default schema, one may pass a `sqlalchemy.MetaData` object to `data`. For example:

```
>>> from blaze import data
>>> from sqlalchemy import MetaData
>>> ds = data(MetaData('postgresql://localhost/test', schema='my_schema'))
>>> ds.dshape
dshape("{table_a: var * {a: ?int32}, table_b: var * {b: ?int32}}")
```

Foreign Keys and Automatic Joins

Often times one wants to access the columns of a table into which we have a foreign key.

For example, given a `products` table with this schema:

```
create table products (
    id integer primary key,
    name text
)
```

and an `orders` table with this schema:

```
create table orders (
    id integer primary key,
    product_id integer references (id) products,
    quantity integer
)
```

we want to get the name of the products in every order. In SQL, you would write the following join:

```
select
    o.id, p.name
from
    orders o
        inner join
    products p
        on o.product_id = p.id
```

This is fairly straightforward. However, when you have more than two joins the SQL gets unruly and hard to read. What we really want is a syntactically simply way to follow the chain of foreign key relationships and be able to access columns in foreign tables without having to write a lot of code. This is where blaze comes in.

Blaze can generate the above joins for you, so instead of writing a bunch of joins in SQL you can simply access the columns of a foreign table as if they were columns on the foreign key column.

The previous example in blaze looks like this:

```
>>> from blaze import data, compute
>>> d = data('postgresql://localhost/db')
>>> d.fields
['products', 'orders']
>>> expr = d.orders.product_id.name
>>> print(compute(expr))
SELECT orders.id, p.name
FROM orders as o, products as p
WHERE o.product_id = p.id
```

Warning: The above feature is very experimental right now. We would appreciate bug reports and feedback on the API.

2.1.12 Out of Core Processing

Blaze includes nascent support for out-of-core processing with Pandas DataFrames and NumPy NDArrays. It combines a computationally-rich in-memory solution (like pandas/numpy) with a computationally-poor out-of-core solution.

How do I use this?

Naive use of Blaze triggers out-of-core systems automatically when called on large files.

```
>>> d = data('my-small-file.csv')
>>> d.my_column.count()  # Uses Pandas

>>> d = data('my-large-file.csv')
>>> d.my_column.count()  # Uses Chunked Pandas
```

How does it work?

Blaze breaks up the data resource into a sequence of chunks. It pulls one chunk into memory, operates on it, pulls in the next, etc.. After all chunks are processed it often has to finalize the computation with another operation on the intermediate results.

In the example above one might accomplish the computation above, counting the number of non-null elements, with pure Pandas as follows:

```
# Operate on each chunk
intermediate = []
for chunk in pd.read_csv('my-large-file.csv', chunksize=1000000):
    intermediate.append(chunk.my_column.count())

# Finish computation by operating on the intermediate result
result = sum(intermediate)
```

This example accomplishes a single computation on the entire dataset, `d.my_column.count()`, by separating it into two stages

1. compute `chunk.my_column.count()` on each in-memory chunk

2. compute `intermediate.sum()` on the aggregated intermediate results

Blaze figures out this process for you. The code above only serves as an example of the kind of thing that Blaze does automatically. Blaze knows how to separate a broad range of computations. Notable exceptions include joins and sorts. Blaze does not currently support out-of-core computation on joins and sorts.

Complex Example

To investigate further try out the `split` function in `blaze.expr.split`. It will tell you exactly how Blaze intends to break up your computation. Here is a more complex example doing an out-of-core split-apply-combine operation:

```
>>> from blaze import *
>>> from blaze.expr.split import split

>>> bank = symbol('bank', 'var * {name: string, balance: int}')

>>> expr = by(bank.name, avg=bank.balance.mean())

>>> split(bank, expr)
((chunk,
  by(chunk.name, avg_count=count(chunk.balance),
                 avg_total=sum(chunk.balance))),
(aggregate,
  by(aggregate.name, avg=(sum(aggregate.avg_total)) /
                          sum(aggregate.avg_count))))
```

As in the first example this chunked split-apply-combine operation translates the intended results into two different computations, one to perform on each in-memory chunk of the data and one to perform on the aggregated results.

Note that you do not need to use `split` yourself. Blaze does this for you automatically.

Parallel Processing

If a data source is easily separable into chunks in a parallel manner then computation may be accelerated by a parallel map function provided by the `multiprocessing` module (or any similar module).

For example a dataset comprised of many CSV files may be easily split up (one csv file = one chunk.) To supply a parallel map function one currently must use the explicit `compute` function.

```
>>> d = data('my-many-csv-files-*.csv')
>>> d.my_column.count()  # Single core by default
...

>>> import multiprocessing
>>> pool = multiprocessing.Pool(4)  # Four processes

>>> compute(d.my_column.count(), map=pool.map)  # Parallel over four cores
...
```

Note that one can only parallelize over datasets that can be easily split in a non-serial fashion. In particular one can not parallelize computation over a single CSV file. Collections of CSV files and binary storage systems like HDF5 and BColz all support multiprocessing.

Beyond CSVs

While pervasive, CSV files may not be the best choice for speedy processing. Binary storage formats like HDF5 and BColz provide more opportunities for parallelism and are generally much faster for large datasets.

2.1.13 Server

Blaze provides uniform access to a variety of common data formats. Blaze Server builds off of this uniform interface to host data remotely through a JSON web API.

Setting up a Blaze Server

To demonstrate the use of the Blaze server we serve the iris csv file.

```
>>> # Server code, run this once.  Leave running.

>>> from blaze import *
>>> from blaze.utils import example
>>> csv = CSV(example('iris.csv'))
>>> data(csv).peek()
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
...
```

Then we host this publicly on port 6363

```
from blaze.server import Server
server = Server(csv)
server.run(host='0.0.0.0', port=6363)
```

A Server is the following

1. A dataset that blaze understands or dictionary of such datasets
2. A Flask app.

With this code our machine is now hosting our CSV file through a web-application on port 6363. We can now access our CSV file, through Blaze, as a service from a variety of applications.

Serving Data from the Command Line

Blaze ships with a command line tool called `blaze-server` to serve up data specified in a YAML file.

Note: To use the YAML specification feature of Blaze server please install the `pyyaml` library. This can be done easily with `conda`:

```
conda install pyyaml
```

YAML Specification

The structure of the specification file is as follows:

```
name1:
  source: path or uri
  dshape: optional datashape
name2:
  source: path or uri
  dshape: optional datashape
...
nameN:
  source: path or uri
  dshape: optional datashape
```

Note: When `source` is a directory, Blaze will recurse into the directory tree and call `odo.resource` on the leaves of the tree.

Here's an example specification file:

```
iriscsv:
  source: ../examples/data/iris.csv
irisdb:
  source: sqlite:///../examples/data/iris.db
accounts:
  source: ../examples/data/accounts.json.gz
  dshape: "var * {name: string, amount: float64}"
```

The previous YAML specification will serve the following dictionary:

```
>>> from odo import resource
>>> resources = {
...  'iriscsv': resource('../examples/data/iris.csv'),
...  'irisdb': resource('sqlite:///../examples/data/iris.db'),
...  'accounts': resource('../examples/data/accounts.json.gz',
...                       dshape="var * {name: string, amount: float64}")
... }
```

The only required key for each named data source is the `source` key, which is passed to `odo.resource`. You can optionally specify a `dshape` parameter, which is passed into `odo.resource` along with the `source` key.

Advanced YAML usage

If `odo.resource` requires extra keyword arguments for a particular resource type and they are provided in the YAML file, these will be forwarded on to the `resource` call.

If there is an `imports` entry for a resource whose value is a list of module or package names, Blaze server will `import` each of these modules or packages before calling `resource`.

For example:

```
name1:
    source: path or uri
    dshape: optional datashape
    kwarg1: extra kwarg
    kwarg2: etc.
name2:
    source: path or uri
    imports: ['mod1', 'pkg2']
```

For this YAML file, Blaze server will pass on `kwarg1=...` and `kwarg2=...` to the `resource()` call for `name1` in addition to the `dshape=...` keyword argument.

Also, before calling `resource` on the `source` of `name2`, Blaze server will first execute an `import mod1` and `import pkg2` statement.

Command Line Interface

1. UNIX

```
# YAML file specifying resources to load and optionally their␣
↪datashape
$ cat example.yaml
iriscsv:
  source: ../examples/data/iris.csv
irisdb:
  source: sqlite:///../examples/data/iris.db
accounts:
  source: ../examples/data/accounts.json.gz
  dshape: "var * {name: string, amount: float64}"

# serve data specified in a YAML file and follow symbolic links
$ blaze-server example.yaml --follow-links

# You can also construct a YAML file from a heredoc to pipe to blaze-
↪server
$ cat <<EOF
datadir:
  source: /path/to/data/directory
EOF | blaze-server
```

2. Windows

```
# If you're on Windows you can do this with powershell
PS C:\> @'
datadir:
  source: C:\path\to\data\directory
'@ | blaze-server
```

Interacting with the Web Server from the Client

Computation is now available on this server at `localhost:6363/compute.json`. To communicate the computation to be done we pass Blaze expressions in JSON format through the request. See the examples below.

Fully Interactive Python-to-Python Remote work

The highest level of abstraction and the level that most will probably want to work at is interactively sending computations to a Blaze server process from a client.

We can use Blaze server to have one Blaze process control another. Given our iris web server we can use Blaze on the client to drive the server to do work for us

```
# Client code, run this in a separate process from the Server

>>> from blaze import data, by
>>> t = data('blaze://localhost:6363')  # doctest: +SKIP

>>> t  # doctest: +SKIP
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
...

>>> by(t.species, min=t.petal_length.min(),
...               max=t.petal_length.max())  # doctest: +SKIP
           species  max  min
0   Iris-virginica  6.9  4.5
1      Iris-setosa  1.9  1.0
2  Iris-versicolor  5.1  3.0
```

We interact on the client machine through the data object but computations on this object cause communications through the web API, resulting in seemlessly interactive remote computation.

The blaze server and client can be configured to support various serialization formats. These formats are exposed in the `blaze.server` module. The server and client must both be told to use the same serialization format. For example:

```
# Server setup.
>>> from blaze import Server
>>> from blaze.server import msgpack_format, json_format
>>> Server(my_data, formats=(msgpack_format, json_format).run()  # doctest: +SKIP

# Client code, run this in a separate process from the Server
>>> from blaze import Client, data
>>> from blaze.server import msgpack_format, json_format
>>> msgpack_client = data(Client('localhost', msgpack_format))  # doctest: +SKIP
>>> json_client = data(Client('localhost', json_format))  # doctest +SKIP
```

In this example, `msgpack_client` will make requests to the `/compute.msgpack` endpoint and will send and receive data using the msgpack protocol; however, the `json_client` will make requests to the `/compute.json` endpoint and will send and receive data using the json protocol.

Using the Python Requests Library

Moving down the stack, we can interact at the HTTP request level with Blaze serer using the `requests` library.

```
# Client code, run this in a separate process from the Server

>>> import json
>>> import requests
>>> query = {'expr': {'op': 'sum',
...                   'args': [{'op': 'Field',
...                             'args': [':leaf', 'petal_length']}]}}
>>> r = requests.get('http://localhost:6363/compute.json',
...                  data=json.dumps(query),
...                  headers={'Content-Type': 'application/vnd.blaze+json'})   #␣
↪doctest: +SKIP
>>> json.loads(r.content)  # doctest: +SKIP
{u'data': 563.8000000000004,
 u'names': ['petal_length_sum'],
 u'datashape': u'{petal_length_sum: float64}'}
```

Now we use Blaze to generate the query programmatically

```
>>> from blaze import symbol
>>> from blaze.server import to_tree
>>> from pprint import pprint

>>> # Build a Symbol like our served iris data
>>> dshape = """var * {
...     sepal_length: float64,
...     sepal_width: float64,
...     petal_length: float64,
...     petal_width: float64,
...     species: string
... }"""  # matching schema to csv file
>>> t = symbol('t', dshape)
>>> expr = t.petal_length.sum()
>>> d = to_tree(expr, names={t: ':leaf'})
>>> query = {'expr': d}
>>> pprint(query)
{'expr': {'args': [{'args': [':leaf', 'petal_length'], 'op': 'Field'},
                   [0],
                   False],
          'op': 'sum'}}
```

Alternatively we build a query to grab a single column

```
>>> pprint(to_tree(t.species, names={t: ':leaf'}))
{'args': [':leaf', 'species'], 'op': 'Field'}
```

Using `curl`

In fact, any tool that is capable of sending requests to a server is able to send computations to a Blaze server.

We can use standard command line tools such as `curl` to interact with the server:

```
$ curl \
    -H "Content-Type: application/vnd.blaze+json" \
    -d '{"expr": {"op": "Field", "args": [":leaf", "species"]}}' \
    localhost:6363/compute.json

{
  "data": [
      "Iris-setosa",
      "Iris-setosa",
      ...
      ],
  "datashape": "var * {species: string}",
}

$ curl \
    -H "Content-Type: application/vnd.blaze+json" \
    -d  '{"expr": {"op": "sum", \
                   "args": [{"op": "Field", \
                             "args": [":leaf", "petal_Length"]}]}}' \
    localhost:6363/compute.json

{
  "data": 563.8000000000004,
  "datashape": "{petal_length_sum: float64}",
}
```

These queries deconstruct the Blaze expression as nested JSON. The `":leaf"` string is a special case pointing to the base data. Constructing these queries can be difficult to do by hand, fortunately Blaze can help you to build them.

Adding Data to the Server

Data resources can be added to the server from the client by sending a resource URI to the server. The data initially on the server must have a dictionary-like interface to be updated.

```
>>> from blaze.utils import example
>>> query = {'accounts': example('accounts.csv')}
>>> r = requests.get('http://localhost:6363/add',
...                  data=json.dumps(query),
...                  headers={'Content-Type': 'application/vnd.blaze+json'})
```

Advanced Use

Blaze servers may host any data that Blaze understands from a single integer

```
>>> server = Server(1)
```

To a dictionary of several heterogeneous datasets

```
>>> server = Server({
...     'my-dataframe': df,
...     'iris': resource('iris.csv'),
...     'baseball': resource('sqlite:///baseball-statistics.db')
... })
```

A variety of hosting options are available through the Flask project

```
>>> help(server.app.run)
Help on method run in module flask.app:

run(self, host=None, port=None, debug=None, **options) method of  flask.app.Flask␣
↪instance
Runs the application on a local development server.  If the
:attr:`debug` flag is set the server will automatically reload
for code changes and show a debugger in case an exception happened.

...
```

Caching

Caching results on frequently run queries may significantly improve user experience in some cases. One may wrap a Blaze server in a traditional web-based caching system like memcached or use a data centric solution.

The Blaze `CachedDataset` might be appropriate in some situations. A cached dataset holds a normal dataset and a `dict` like object.

```
>>> dset = {'my-dataframe': df,
...         'iris': resource('iris.csv'),
...         'baseball': resource('sqlite:///baseball-statistics.db')}

>>> from blaze.cached import CachedDataset
>>> cached = CachedDataset(dset, cache=dict())
```

Queries and results executed against a cached dataset are stored in the cache (here a normal Python `dict`) for fast future access.

If accumulated results are likely to fill up memory then other, on-disk `dict`-like structures can be used like Shove or Chest.

```
>>> from chest import Chest
>>> cached = CachedDataset(dset, cache=Chest())
```

These cached objects can be used anywhere normal objects can be used in Blaze, including an interactive (and now performance cached) `data` object

```
>>> d = data(cached)
```

or a Blaze server

```
>>> server = Server(cached)
```

Flask Blueprint

If you would like to use the blaze server endpoints from within another flask application, you can register the blaze API blueprint with your application. For example:

```
>>> from blaze.server import api, json_format
>>> my_app.register_blueprint(api, data=my_data, formats=(json_format,))
```

When registering the API, you must pass the data that the API endpoints will serve. You must also pass an iterable of serialization format objects that the server will respond to.

Profiling

The blaze server allows users and server administrators to profile computations run on the server. This allows developers to better understand the performance profile of their computations to better tune their queries or the backend code that is executing the query. This profiling will also track the time spent in serializing the data.

By default, blaze servers will not allow profiling. To enable profiling on the blaze server, pass `allow_profiler=True` to the `Server` object. Now when we try to compute against this server, we may pass `profile=True` to `compute`. For example:

```
>>> client = Client(...)
>>> compute(expr, client, profile=True)
```

After running the above code, the server will have written a new pstats file containing the results of the run. This fill will be found at: `profiler_output/<md5>/<timestamp>`. We use the md5 hash of the str of the expression so that users can more easily track down their stats information. Users can find the hash of their expression with `expr_md5()`.

The profiler output directory may be configured with the `profiler_output` argument to the `Server`.

Clients may also request that the profiling data be sent back in the response so that analysis may happen on the client. To do this, we change our call to compute to look like:

```
>>> from io import BytesIO
>>> buf = BytesIO()
>>> compute(expr, client, profile=True, profiler_output=buf)
```

After that computation, `buf` will have the the marshalled stats data suitable for reading with `pstats`. This feature is useful when blaze servers are being run behind a load balancer and we do not want to search all of the servers to find the output.

Note: Because the data is serialized with `marshal` it must be read by the same version of python as the server. This means that a python 2 client cannot unmarshal the data written by a python 3 server. This is to conform with the file format expected by `pstats`, the standard profiling output inspection library.

System administrators may also configure all computations to be profiled by default. This is useful if the client code cannot be easily changed or threading arguments to compute is hard in an application setting. This may be set with `profile_by_default=True` when constructing the server.

Conclusion

Because this process builds off Blaze expressions it works equally well for data stored in any format on which Blaze is trained, including in-memory DataFrames, SQL/Mongo databases, or even Spark clusters.

2.1.14 Datashape

Blaze uses datashape, a data layout language for array programming, as its type system.

- Documentation
- Source

2.1.15 What Blaze Doesn't Do

Blaze occasionally suffers from over-hype. The terms *Big-Data* and *Pandas* inevitably conflate in people's minds to become something unattainable and lead to disappointment. Blaze is limited; learning those limitations can direct you to greater productivity.

First and foremost, Blaze does not replace Pandas. Pandas will always be more feature rich and more mature than Blaze. There are things that you simply can't do if you want to generalize out of memory.

If your data fits nicely in memory then use NumPy/Pandas. Your data probably fits nicely in memory.

Some concrete things Blaze doesn't do

1. Clean unstructured data. Blaze only handles analytic queries on structured data.
2. Most things in SciPy. Including things like FFT, and gradient descent.
3. Most things in SciKit Learn/Image/etc..
4. Statistical inference - We invite you to build this (this one is actually pretty doable.)
5. Parallelize your existing Python code
6. Replace Spark - Blaze may operate on top of Spark, it doesn't compete with it.
7. Compute quickly - Blaze uses other things to compute, it doesn't compute anything itself. So asking questions about how fast Blaze is are determined entirely by how fast other things are.

That's not to say that these can't be done

Blaze aims to be a foundational data interface like `numpy/pandas` rather than try to implement the entire PyData stack (`scipy, scikit-learn`, etc..) Only by keeping scope small do we have a chance at relevance.

Of course, others can build off of Blaze in the same way that `scipy` and `scikit-learn` built off of `numpy/pandas`. Blaze devs often also do this work (it's important) but we generally don't include it in the Blaze library.

It's also worth mentioning that different classes of algorithms work well on small vs large datasets. It could be that the algorithm that you like most may not easily extend beyond the scope of memory. A direct translation of scikit-learn algorithms to Blaze would likely be computationally disastrous.

What Blaze Does

Blaze is a query system that looks like NumPy/Pandas. You write Blaze queries, Blaze translates those queries to something else (like SQL), and ships those queries to various database to run on other people's fast code. It smoothes out this process to make interacting with foreign data as accessible as using Pandas. This is actually quite difficult.

Blaze increases human accessibility, not computational performance.

But we work on other things

Blaze devs interact with a lot of other computational systems. Sometimes we find holes where systems should exist, but don't. In these cases we may write our own computational system. In these cases we naturally hook up Blaze to serve as a front-end query system. We often write about these experiments.

As a result you may see us doing some of the things we just said "Blaze doesn't do". These things aren't Blaze (but you *can* use Blaze to do them easily.)

2.1.16 API

This page contains a comprehensive list of functionality within `blaze`. Docstrings should provide sufficient understanding for any individual function or class.

Interactive Use

`_Data`	Bind a data resource to a symbol, for use in expressions and computation.

Expressions

`Projection`	Select a subset of fields from data.
`Selection`	Filter elements of expression based on predicate
`Label`	An expression with a name.
`ReLabel`	Table with same content but with new labels
`Map`	Map an arbitrary Python function across elements in a collection
`Apply`	Apply an arbitrary Python function onto an expression
`Coerce`	Coerce an expression to a different type.
`Coalesce`	SQL like coalesce.
`Cast`	Cast an expression to a different type.

`Sort`	Table in sorted order
`Distinct`	Remove duplicate elements from an expression
`Head`	First *n* elements of collection
`Merge`	Merge many fields together
`Join`	Join two tables on common columns
`Concat`	Stack tables on common columns
`IsIn`	Check if an expression contains values from a set.

`By`	Split-Apply-Combine Operator

Blaze Server

`Server`

`Client`

Additional Server Utilities

`expr_md5`
`to_tree`
`from_tree`

`data_spider`
`from_yaml`

Definitions

`blaze.interactive.`**`data`**(*data_source*, *dshape=None*, *name=None*, *fields=None*, *schema=None*, ***kwargs*)

Bind a data resource to a symbol, for use in expressions and computation.

A `data` object presents a consistent view onto a variety of concrete data sources. Like `symbol` objects, they are meant to be used in expressions. Because they are tied to concrete data resources, `data` objects can be used with `compute` directly, making them convenient for interactive exploration.

Parameters **data_source** : object

Any type with `discover` and `compute` implementations

fields : list, optional

Field or column names, will be inferred from data_source if possible

dshape : str or DataShape, optional

DataShape describing input data

name : str, optional

A name for the data.

Examples

```
>>> t = data([(1, 'Alice', 100),
...           (2, 'Bob', -200),
...           (3, 'Charlie', 300),
...           (4, 'Denis', 400),
...           (5, 'Edith', -500)],
...          fields=['id', 'name', 'balance'])
>>> t[t.balance < 0].name
    name
0    Bob
1  Edith
```

class `blaze.expr.collections.`**`Concat`**

Stack tables on common columns

Parameters **lhs, rhs** : Expr

Collections to concatenate

axis : int, optional

The axis to concatenate on.

See also:

`blaze.expr.collections.Merge`

Examples

```
>>> from blaze import symbol
```

Vertically stack tables:

```
>>> names = symbol('names', '5 * {name: string, id: int32}')
>>> more_names = symbol('more_names', '7 * {name: string, id: int32}')
>>> stacked = concat(names, more_names)
>>> stacked.dshape
dshape("12 * {name: string, id: int32}")
```

Vertically stack matrices:

```
>>> mat_a = symbol('a', '3 * 5 * int32')
>>> mat_b = symbol('b', '3 * 5 * int32')
>>> vstacked = concat(mat_a, mat_b, axis=0)
>>> vstacked.dshape
dshape("6 * 5 * int32")
```

Horizontally stack matrices:

```
>>> hstacked = concat(mat_a, mat_b, axis=1)
>>> hstacked.dshape
dshape("3 * 10 * int32")
```

`blaze.expr.collections.`**`concat`**(*lhs*, *rhs*, *axis=0*)

Stack tables on common columns

Parameters lhs, rhs : Expr

Collections to concatenate

axis : int, optional

The axis to concatenate on.

See also:

`blaze.expr.collections.Merge`

Examples

```
>>> from blaze import symbol
```

Vertically stack tables:

```
>>> names = symbol('names', '5 * {name: string, id: int32}')
>>> more_names = symbol('more_names', '7 * {name: string, id: int32}')
>>> stacked = concat(names, more_names)
>>> stacked.dshape
dshape("12 * {name: string, id: int32}")
```

Vertically stack matrices:

```
>>> mat_a = symbol('a', '3 * 5 * int32')
>>> mat_b = symbol('b', '3 * 5 * int32')
>>> vstacked = concat(mat_a, mat_b, axis=0)
>>> vstacked.dshape
dshape("6 * 5 * int32")
```

Horizontally stack matrices:

```
>>> hstacked = concat(mat_a, mat_b, axis=1)
>>> hstacked.dshape
dshape("3 * 10 * int32")
```

class `blaze.expr.collections.`**`Distinct`**

Remove duplicate elements from an expression

Parameters **on** : tuple of *`Field`*

The subset of fields or names of fields to be distinct on.

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> e = distinct(t)
```

```
>>> data = [('Alice', 100, 1),
...         ('Bob', 200, 2),
...         ('Alice', 100, 1)]
```

```
>>> from blaze.compute.python import compute
>>> sorted(compute(e, data))
[('Alice', 100, 1), ('Bob', 200, 2)]
```

Use a subset by passing *on*:

```
>>> import pandas as pd
>>> e = distinct(t, 'name')
>>> data = pd.DataFrame([['Alice', 100, 1],
...                      ['Alice', 200, 2],
...                      ['Bob', 100, 1],
...                      ['Bob', 200, 2]],
...                     columns=['name', 'amount', 'id'])
>>> compute(e, data)
    name  amount  id
0  Alice     100   1
1    Bob     100   1
```

`blaze.expr.collections.`**`distinct`**`(`*expr*, **on*`)`

Remove duplicate elements from an expression

Parameters **on** : tuple of *`Field`*

The subset of fields or names of fields to be distinct on.

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> e = distinct(t)
```

```
>>> data = [('Alice', 100, 1),
...         ('Bob', 200, 2),
...         ('Alice', 100, 1)]
```

```
>>> from blaze.compute.python import compute
>>> sorted(compute(e, data))
[('Alice', 100, 1), ('Bob', 200, 2)]
```

Use a subset by passing *on*:

```
>>> import pandas as pd
>>> e = distinct(t, 'name')
>>> data = pd.DataFrame([['Alice', 100, 1],
...                      ['Alice', 200, 2],
...                      ['Bob', 100, 1],
...                      ['Bob', 200, 2]],
...                     columns=['name', 'amount', 'id'])
>>> compute(e, data)
    name  amount  id
0  Alice     100   1
1    Bob     100   1
```

class `blaze.expr.collections.`**`Head`**

First *n* elements of collection

See also:

`blaze.expr.collections.Tail`

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.head(5).dshape
dshape("5 * {name: string, amount: int32}")
```

`blaze.expr.collections.`**`head`**(*child*, *n=10*)

First *n* elements of collection

See also:

`blaze.expr.collections.Tail`

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.head(5).dshape
dshape("5 * {name: string, amount: int32}")
```

class `blaze.expr.collections.`**`IsIn`**

Check if an expression contains values from a set.

Return a boolean expression indicating whether another expression contains values that are members of a collection.

Parameters **expr** : Expr

Expression whose elements to check for membership in *keys*

keys : Sequence

Elements to test against. Blaze stores this as a `frozenset`.

Examples

Check if a vector contains any of 1, 2 or 3:

```
>>> from blaze import symbol
>>> t = symbol('t', '10 * int64')
>>> expr = t.isin([1, 2, 3])
>>> expr.dshape
dshape("10 * bool")
```

`blaze.expr.collections.`**`isin`**`(`*expr*, *keys*`)`

Check if an expression contains values from a set.

Return a boolean expression indicating whether another expression contains values that are members of a collection.

Parameters **expr** : Expr

Expression whose elements to check for membership in *keys*

keys : Sequence

Elements to test against. Blaze stores this as a `frozenset`.

Examples

Check if a vector contains any of 1, 2 or 3:

```
>>> from blaze import symbol
>>> t = symbol('t', '10 * int64')
>>> expr = t.isin([1, 2, 3])
>>> expr.dshape
dshape("10 * bool")
```

class `blaze.expr.collections.`**`Join`**

Join two tables on common columns

Parameters **lhs, rhs** : Expr

Expressions to join

on_left : str, optional

The fields from the left side to join on. If no `on_right` is passed, then these are the fields for both sides.

on_right : str, optional

The fields from the right side to join on.

how : {'inner', 'outer', 'left', 'right'}

What type of join to perform.

suffixes: pair of str

The suffixes to be applied to the left and right sides in order to resolve duplicate field names.

See also:

blaze.expr.collections.Merge

Examples

```
>>> from blaze import symbol
>>> names = symbol('names', 'var * {name: string, id: int}')
>>> amounts = symbol('amounts', 'var * {amount: int, id: int}')
```

Join tables based on shared column name

```
>>> joined = join(names, amounts, 'id')
```

Join based on different column names

```
>>> amounts = symbol('amounts', 'var * {amount: int, acctNumber: int}')
>>> joined = join(names, amounts, 'id', 'acctNumber')
```

`blaze.expr.collections.`**`join`**(*lhs*, *rhs*, *on_left=None*, *on_right=None*, *how='inner'*, *suffixes=('_left', '_right')*)

Join two tables on common columns

Parameters **lhs, rhs** : Expr

Expressions to join

on_left : str, optional

The fields from the left side to join on. If no `on_right` is passed, then these are the fields for both sides.

on_right : str, optional

The fields from the right side to join on.

how : {'inner', 'outer', 'left', 'right'}

What type of join to perform.

suffixes: pair of str

The suffixes to be applied to the left and right sides in order to resolve duplicate field names.

See also:

blaze.expr.collections.Merge

Examples

```
>>> from blaze import symbol
>>> names = symbol('names', 'var * {name: string, id: int}')
>>> amounts = symbol('amounts', 'var * {amount: int, id: int}')
```

Join tables based on shared column name

```
>>> joined = join(names, amounts, 'id')
```

Join based on different column names

```
>>> amounts = symbol('amounts', 'var * {amount: int, acctNumber: int}')
>>> joined = join(names, amounts, 'id', 'acctNumber')
```

class `blaze.expr.collections.`**`Merge`**

Merge many fields together

Examples

```
>>> from blaze import symbol, label
>>> accounts = symbol('accounts', 'var * {name: string, x: int, y: real}')
>>> merge(accounts.name, z=accounts.x + accounts.y).fields
['name', 'z']
```

To control the ordering of the fields, use `label`:

```
>>> merge(label(accounts.name, 'NAME'), label(accounts.x, 'X')).dshape
dshape("var * {NAME: string, X: int32}")
>>> merge(label(accounts.x, 'X'), label(accounts.name, 'NAME')).dshape
dshape("var * {X: int32, NAME: string}")
```

`blaze.expr.collections.`**`merge`**`(`**exprs*, ***kwargs*`)`

Merge many fields together

Examples

```
>>> from blaze import symbol, label
>>> accounts = symbol('accounts', 'var * {name: string, x: int, y: real}')
>>> merge(accounts.name, z=accounts.x + accounts.y).fields
['name', 'z']
```

To control the ordering of the fields, use `label`:

```
>>> merge(label(accounts.name, 'NAME'), label(accounts.x, 'X')).dshape
dshape("var * {NAME: string, X: int32}")
>>> merge(label(accounts.x, 'X'), label(accounts.name, 'NAME')).dshape
dshape("var * {X: int32, NAME: string}")
```

class `blaze.expr.collections.`**`Sample`**

Random row-wise sample. Can specify *n* or *frac* for an absolute or fractional number of rows, respectively.

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.sample(n=2).dshape
dshape("var * {name: string, amount: int32}")
>>> accounts.sample(frac=0.1).dshape
dshape("var * {name: string, amount: int32}")
```

`blaze.expr.collections.`**`sample`**(*child*, *n=None*, *frac=None*)

Random row-wise sample. Can specify *n* or *frac* for an absolute or fractional number of rows, respectively.

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.sample(n=2).dshape
dshape("var * {name: string, amount: int32}")
>>> accounts.sample(frac=0.1).dshape
dshape("var * {name: string, amount: int32}")
```

class `blaze.expr.collections.`**`Shift`**

Shift a column backward or forward by N elements

Parameters **expr** : Expr

The expression to shift. This expression's dshape should be columnar

n : int

The number of elements to shift by. If n < 0 then shift backward, if n == 0 do nothing, else shift forward.

`blaze.expr.collections.`**`shift`**(*expr*, *n*)

Shift a column backward or forward by N elements

Parameters **expr** : Expr

The expression to shift. This expression's dshape should be columnar

n : int

The number of elements to shift by. If n < 0 then shift backward, if n == 0 do nothing, else shift forward.

class `blaze.expr.collections.`**`Sort`**

Table in sorted order

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.sort('amount', ascending=False).schema
dshape("{name: string, amount: int32}")
```

Some backends support sorting by arbitrary rowwise tables, e.g.

```
>>> accounts.sort(-accounts.amount)
```

`blaze.expr.collections.`**`sort`**`(`*child*, *key=None*, *ascending=True*`)`

Sort a collection

Parameters **key** : str, list of str, or Expr

Defines by what you want to sort.

- A single column string: `t.sort('amount')`
- A list of column strings: `t.sort(['name', 'amount'])`
- An expression: `t.sort(-t.amount)`

If sorting a columnar dataset, the `key` is ignored, as it is not necessary:

- `t.amount.sort()`
- `t.amount.sort('amount')`
- `t.amount.sort('foobar')`

are all equivalent.

ascending : bool, optional

Determines order of the sort

class `blaze.expr.collections.`**`Tail`**

Last *n* elements of collection

See also:

`blaze.expr.collections.Head`

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.tail(5).dshape
dshape("5 * {name: string, amount: int32}")
```

`blaze.expr.collections.`**`tail`**`(`*child*, *n=10*`)`

Last *n* elements of collection

See also:

`blaze.expr.collections.Head`

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.tail(5).dshape
dshape("5 * {name: string, amount: int32}")
```

`blaze.expr.collections.`**`transform`**`(`*t*, *replace=True*, ***kwargs*`)`

Add named columns to table

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {x: int, y: int}')
>>> transform(t, z=t.x + t.y).fields
['x', 'y', 'z']
```

class `blaze.expr.expressions.`**`Apply`**

Apply an arbitrary Python function onto an expression

See also:

`blaze.expr.expressions.Map`

Examples

```
>>> t = symbol('t', 'var * {name: string, amount: int}')
>>> h = t.apply(hash, dshape='int64')  # Hash value of resultant dataset
```

You must provide the datashape of the result with the `dshape=` keyword. For datashape examples see http://datashape.pydata.org/grammar.html#some-simple-examples

If using a chunking backend and your operation may be safely split and concatenated then add the `splittable=True` keyword argument

```
>>> t.apply(f, dshape='...', splittable=True)
```

class `blaze.expr.expressions.`**`Cast`**

Cast an expression to a different type.

This is only an expression time operation.

Examples

```
>>> s = symbol('s', '?int64')
>>> s.cast('?int32').dshape
dshape("?int32")
```

# Cast to correct mislabeled optionals >>> s.cast('int64').dshape dshape("int64")

# Cast to give concrete dimension length >>> t = symbol('t', 'var * float32') >>> t.cast('10 * float32').dshape dshape("10 * float32")

class `blaze.expr.expressions.`**`Coalesce`**

SQL like coalesce.

coalesce(a, b) = { a if a is not NULL b otherwise

}

Examples

```
>>> coalesce(1, 2)
1
>>> coalesce(1, None)
1
>>> coalesce(None, 2)
```

```
2
>>> coalesce(None, None) is None
True
```

class `blaze.expr.expressions.`**`Coerce`**

Coerce an expression to a different type.

Examples

```
>>> t = symbol('t', '100 * float64')
>>> t.coerce(to='int64')
t.coerce(to='int64')
>>> t.coerce('float32')
t.coerce(to='float32')
>>> t.coerce('int8').dshape
dshape("100 * int8")
```

class `blaze.expr.expressions.`**`ElemWise`**

Elementwise operation.

The shape of this expression matches the shape of the child.

class `blaze.expr.expressions.`**`Expr`**

Symbolic expression of a computation

All Blaze expressions (Join, By, Sort, ...) descend from this class. It contains shared logic and syntax. It in turn inherits from `Node` which holds all tree traversal logic

`cast` (*expr*, *to*)

Cast an expression to a different type.

This is only an expression time operation.

Examples

```
>>> s = symbol('s', '?int64')
>>> s.cast('?int32').dshape
dshape("?int32")
```

# Cast to correct mislabeled optionals >>> s.cast('int64').dshape dshape("int64")

# Cast to give concrete dimension length >>> t = symbol('t', 'var * float32') >>> t.cast('10 * float32').dshape dshape("10 * float32")

`map` (*func*, *schema=None*, *name=None*)

Map an arbitrary Python function across elements in a collection

See also:

`blaze.expr.expresions.Apply`

Examples

```
>>> from datetime import datetime
>>> t = symbol('t', 'var * {price: real, time: int64}')  # times as integers
>>> datetimes = t.time.map(datetime.utcfromtimestamp)
```

Optionally provide extra schema information

```
>>> datetimes = t.time.map(datetime.utcfromtimestamp,
...                           schema='{time: datetime}')
```

class `blaze.expr.expressions.`**`Field`**

A single field from an expression.

Get a single field from an expression with record-type schema. We store the name of the field in the `_name` attribute.

Examples

```
>>> points = symbol('points', '5 * 3 * {x: int32, y: int32}')
>>> points.x.dshape
dshape("5 * 3 * int32")
```

For fields that aren't valid Python identifiers, use `[]` syntax:

```
>>> points = symbol('points', '5 * 3 * {"space station": float64}')
>>> points['space station'].dshape
dshape("5 * 3 * float64")
```

class `blaze.expr.expressions.`**`Label`**

An expression with a name.

See also:

`blaze.expr.expressions.ReLabel`

Examples

```
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> expr = accounts.amount * 100
>>> expr._name
'amount'
>>> expr.label('new_amount')._name
'new_amount'
```

class `blaze.expr.expressions.`**`Map`**

Map an arbitrary Python function across elements in a collection

See also:

`blaze.expr.expresions.Apply`

Examples

```
>>> from datetime import datetime
>>> t = symbol('t', 'var * {price: real, time: int64}')  # times as integers
>>> datetimes = t.time.map(datetime.utcfromtimestamp)
```

Optionally provide extra schema information

```
>>> datetimes = t.time.map(datetime.utcfromtimestamp,
...                        schema='{time: datetime}')
```

class `blaze.expr.expressions.`**`Projection`**

Select a subset of fields from data.

See also:

`blaze.expr.expressions.Field`

Examples

```
>>> accounts = symbol('accounts',
...                   'var * {name: string, amount: int, id: int}')
>>> accounts[['name', 'amount']].schema
dshape("{name: string, amount: int32}")
>>> accounts[['name', 'amount']]
accounts[['name', 'amount']]
```

class `blaze.expr.expressions.`**`ReLabel`**

Table with same content but with new labels

See also:

`blaze.expr.expressions.Label`

Notes

When names are not valid Python names, such as integers or string with spaces, you must pass a dictionary to `relabel`. For example

```
>>> s = symbol('s', 'var * {"0": int64}')
>>> s.relabel({'0': 'foo'})
s.relabel({'0': 'foo'})
>>> t = symbol('t', 'var * {"whoo hoo": ?float32}')
>>> t.relabel({"whoo hoo": 'foo'})
t.relabel({'whoo hoo': 'foo'})
```

Examples

```
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.schema
dshape("{name: string, amount: int32}")
>>> accounts.relabel(amount='balance').schema
dshape("{name: string, balance: int32}")
>>> accounts.relabel(not_a_column='definitely_not_a_column')
Traceback (most recent call last):
    ...
ValueError: Cannot relabel non-existent child fields: {'not_a_column'}
```

```
>>> s = symbol('s', 'var * {"0": int64}')
>>> s.relabel({'0': 'foo'})
s.relabel({'0': 'foo'})
>>> s.relabel(0='foo')
Traceback (most recent call last):
    ...
SyntaxError: keyword can't be an expression
```

class `blaze.expr.expressions.`**`Selection`**

Filter elements of expression based on predicate

Examples

```
>>> accounts = symbol('accounts',
...                   'var * {name: string, amount: int, id: int}')
>>> deadbeats = accounts[accounts.amount < 0]
```

class `blaze.expr.expressions.`**`SimpleSelection`**

Internal selection class that does not treat the predicate as an input.

class `blaze.expr.expressions.`**`Slice`**

Elements *start* until *stop*. On many backends, a *step* parameter is also allowed.

Examples

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts[2:7].dshape
dshape("5 * {name: string, amount: int32}")
>>> accounts[2:7:2].dshape
dshape("3 * {name: string, amount: int32}")
```

class `blaze.expr.expressions.`**`Symbol`**

Symbolic data. The leaf of a Blaze expression

Examples

```
>>> points = symbol('points', '5 * 3 * {x: int, y: int}')
>>> points
<`points` symbol; dshape='5 * 3 * {x: int32, y: int32}'>
>>> points.dshape
dshape("5 * 3 * {x: int32, y: int32}")
```

`blaze.expr.expressions.`**`apply`**(*expr*, *func*, *dshape*, *splittable=False*)

Apply an arbitrary Python function onto an expression

See also:

`blaze.expr.expressions.Map`

Examples

```
>>> t = symbol('t', 'var * {name: string, amount: int}')
>>> h = t.apply(hash, dshape='int64')  # Hash value of resultant dataset
```

You must provide the datashape of the result with the `dshape=` keyword. For datashape examples see http://datashape.pydata.org/grammar.html#some-simple-examples

If using a chunking backend and your operation may be safely split and concatenated then add the `splittable=True` keyword argument

```
>>> t.apply(f, dshape='...', splittable=True)
```

`blaze.expr.expressions.`**`cast`**`(`*expr*, *to*`)`

Cast an expression to a different type.

This is only an expression time operation.

Examples

```
>>> s = symbol('s', '?int64')
>>> s.cast('?int32').dshape
dshape("?int32")
```

# Cast to correct mislabeled optionals >>> s.cast('int64').dshape dshape("int64")

# Cast to give concrete dimension length >>> t = symbol('t', 'var * float32') >>> t.cast('10 * float32').dshape dshape("10 * float32")

`blaze.expr.expressions.`**`coalesce`**`(`*a*, *b*`)`

SQL like coalesce.

coalesce(a, b) = { a if a is not NULL b otherwise

}

Examples

```
>>> coalesce(1, 2)
1
>>> coalesce(1, None)
1
>>> coalesce(None, 2)
2
>>> coalesce(None, None) is None
True
```

`blaze.expr.expressions.`**`coerce`**`(`*expr*, *to*`)`

Coerce an expression to a different type.

Examples

```
>>> t = symbol('t', '100 * float64')
>>> t.coerce(to='int64')
t.coerce(to='int64')
>>> t.coerce('float32')
t.coerce(to='float32')
>>> t.coerce('int8').dshape
dshape("100 * int8")
```

`blaze.expr.expressions.`**`label`**(*expr*, *lab*)

An expression with a name.

See also:

blaze.expr.expressions.ReLabel

Examples

```
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> expr = accounts.amount * 100
>>> expr._name
'amount'
>>> expr.label('new_amount')._name
'new_amount'
```

`blaze.expr.expressions.`**`ndim`**(*expr*)

Number of dimensions of expression

```
>>> symbol('s', '3 * var * int32').ndim
2
```

`blaze.expr.expressions.`**`projection`**(*expr*, *names*)

Select a subset of fields from data.

See also:

blaze.expr.expressions.Field

Examples

```
>>> accounts = symbol('accounts',
...                   'var * {name: string, amount: int, id: int}')
>>> accounts[['name', 'amount']].schema
dshape("{name: string, amount: int32}")
>>> accounts[['name', 'amount']]
accounts[['name', 'amount']]
```

`blaze.expr.expressions.`**`relabel`**(*child*, *labels=None*, ***kwargs*)

Table with same content but with new labels

See also:

blaze.expr.expressions.Label

Notes

When names are not valid Python names, such as integers or string with spaces, you must pass a dictionary to `relabel`. For example

```
>>> s = symbol('s', 'var * {"0": int64}')
>>> s.relabel({'0': 'foo'})
s.relabel({'0': 'foo'})
>>> t = symbol('t', 'var * {"whoo hoo": ?float32}')
>>> t.relabel({"whoo hoo": 'foo'})
t.relabel({'whoo hoo': 'foo'})
```

Examples

```
>>> accounts = symbol('accounts', 'var * {name: string, amount: int}')
>>> accounts.schema
dshape("{name: string, amount: int32}")
>>> accounts.relabel(amount='balance').schema
dshape("{name: string, balance: int32}")
>>> accounts.relabel(not_a_column='definitely_not_a_column')
Traceback (most recent call last):
    ...
ValueError: Cannot relabel non-existent child fields: {'not_a_column'}
>>> s = symbol('s', 'var * {"0": int64}')
>>> s.relabel({'0': 'foo'})
s.relabel({'0': 'foo'})
>>> s.relabel(0='foo')
Traceback (most recent call last):
    ...
SyntaxError: keyword can't be an expression
```

`blaze.expr.expressions.`**`selection`**(*table*, *predicate*)
Filter elements of expression based on predicate

Examples

```
>>> accounts = symbol('accounts',
...                   'var * {name: string, amount: int, id: int}')
>>> deadbeats = accounts[accounts.amount < 0]
```

`blaze.expr.expressions.`**`symbol`**(*name*, *dshape*, *token=None*)
Symbolic data. The leaf of a Blaze expression

Examples

```
>>> points = symbol('points', '5 * 3 * {x: int, y: int}')
>>> points
<`points` symbol; dshape='5 * 3 * {x: int32, y: int32}'>
>>> points.dshape
dshape("5 * 3 * {x: int32, y: int32}")
```

class `blaze.expr.reductions.`**`Reduction`**

A column-wise reduction

Blaze supports the same class of reductions as NumPy and Pandas.

sum, min, max, any, all, mean, var, std, count, nunique

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> e = t['amount'].sum()
```

```
>>> data = [['Alice', 100, 1],
...         ['Bob', 200, 2],
...         ['Alice', 50, 3]]
```

```
>>> from blaze.compute.python import compute
>>> compute(e, data)
350
```

class `blaze.expr.reductions.`**`Summary`**

A collection of named reductions

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> expr = summary(number=t.id.nunique(), sum=t.amount.sum())
```

```
>>> data = [['Alice', 100, 1],
...         ['Bob', 200, 2],
...         ['Alice', 50, 1]]
```

```
>>> from blaze import compute
>>> compute(expr, data)
(2, 350)
```

class `blaze.expr.reductions.`**`count`**

The number of non-null elements

class `blaze.expr.reductions.`**`nelements`**

Compute the number of elements in a collection, including missing values.

See also:

`blaze.expr.reductions.count` compute the number of non-null elements

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: float64}')
>>> t[t.amount < 1].nelements()
nelements(t[t.amount < 1])
```

class `blaze.expr.reductions.`**`std`**

Standard Deviation

Parameters **child** : Expr

An expression

unbiased : bool, optional

Compute the square root of an unbiased estimate of the population variance if this is `True`.

Warning: This does *not* return an unbiased estimate of the population standard deviation.

See also:

`var`

`blaze.expr.reductions.`**`summary`**(*keepdims=False*, *axis=None*, ***kwargs*)

A collection of named reductions

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> expr = summary(number=t.id.nunique(), sum=t.amount.sum())
```

```
>>> data = [['Alice', 100, 1],
...         ['Bob', 200, 2],
...         ['Alice', 50, 1]]
```

```
>>> from blaze import compute
>>> compute(expr, data)
(2, 350)
```

class `blaze.expr.reductions.`**`var`**

Variance

Parameters **child** : Expr

An expression

unbiased : bool, optional

Compute an unbiased estimate of the population variance if this is `True`. In NumPy and pandas, this parameter is called `ddof` (delta degrees of freedom) and is equal to 1 for unbiased and 0 for biased.

`blaze.expr.reductions.`**`vnorm`**(*expr*, *ord=None*, *axis=None*, *keepdims=False*)

Vector norm

See np.linalg.norm

class `blaze.expr.arrays.`**`Transpose`**

Transpose dimensions in an N-Dimensional array

Examples

```
>>> x = symbol('x', '10 * 20 * int32')
>>> x.T
transpose(x)
>>> x.T.shape
(20, 10)
```

Specify axis ordering with axes keyword argument

```
>>> x = symbol('x', '10 * 20 * 30 * int32')
>>> x.transpose([2, 0, 1])
transpose(x, axes=[2, 0, 1])
>>> x.transpose([2, 0, 1]).shape
(30, 10, 20)
```

class `blaze.expr.arrays.`**`TensorDot`**

Dot Product: Contract and sum dimensions of two arrays

```
>>> x = symbol('x', '20 * 20 * int32')
>>> y = symbol('y', '20 * 30 * int32')
```

```
>>> x.dot(y)
tensordot(x, y)
```

```
>>> tensordot(x, y, axes=[0, 0])
tensordot(x, y, axes=[0, 0])
```

`blaze.expr.arrays.`**`dot`**(*lhs*, *rhs*)

Dot Product: Contract and sum dimensions of two arrays

```
>>> x = symbol('x', '20 * 20 * int32')
>>> y = symbol('y', '20 * 30 * int32')
```

```
>>> x.dot(y)
tensordot(x, y)
```

```
>>> tensordot(x, y, axes=[0, 0])
tensordot(x, y, axes=[0, 0])
```

`blaze.expr.arrays.`**`transpose`**(*expr*, *axes=None*)

Transpose dimensions in an N-Dimensional array

Examples

```
>>> x = symbol('x', '10 * 20 * int32')
>>> x.T
transpose(x)
>>> x.T.shape
(20, 10)
```

Specify axis ordering with axes keyword argument

```
>>> x = symbol('x', '10 * 20 * 30 * int32')
>>> x.transpose([2, 0, 1])
transpose(x, axes=[2, 0, 1])
>>> x.transpose([2, 0, 1]).shape
(30, 10, 20)
```

blaze.expr.arrays.**tensordot**(*lhs*, *rhs*, *axes=None*)

Dot Product: Contract and sum dimensions of two arrays

```
>>> x = symbol('x', '20 * 20 * int32')
>>> y = symbol('y', '20 * 30 * int32')
```

```
>>> x.dot(y)
tensordot(x, y)
```

```
>>> tensordot(x, y, axes=[0, 0])
tensordot(x, y, axes=[0, 0])
```

class blaze.expr.arithmetic.**Arithmetic**

Super class for arithmetic operators like add or mul

class blaze.expr.math.**notnull**

Return whether an expression is not null

Examples

```
>>> from blaze import symbol, compute
>>> s = symbol('s', 'var * int64')
>>> expr = notnull(s)
>>> expr.dshape
dshape("var * bool")
>>> list(compute(expr, [1, 2, None, 3]))
[True, True, False, True]
```

class blaze.expr.math.**UnaryMath**

Mathematical unary operator with real valued dshape like sin, or exp

class blaze.expr.broadcast.**Broadcast**

Fuse scalar expressions over collections

Given elementwise operations on collections, e.g.

```
>>> from blaze import sin
>>> a = symbol('a', '100 * int')
>>> t = symbol('t', '100 * {x: int, y: int}')
```

```
>>> expr = sin(a) + t.y**2
```

It may be best to represent this as a scalar expression mapped over a collection

```
>>> sa = symbol('a', 'int')
>>> st = symbol('t', '{x: int, y: int}')
>>> sexpr = sin(sa) + st.y**2
```

```
>>> expr = Broadcast((a, t), (sa, st), sexpr)
```

This provides opportunities for optimized computation.

In practice, expressions are often collected into Broadcast expressions automatically. This class is mainly intented for internal use.

`blaze.expr.broadcast.`**`scalar_symbols`**`(`*exprs*`)`

Gives a sequence of scalar symbols to mirror these expressions

Examples

```
>>> x = symbol('x', '5 * 3 * int32')
>>> y = symbol('y', '5 * 3 * int32')
```

```
>>> xx, yy = scalar_symbols([x, y])
```

```
>>> xx._name, xx.dshape
('x', dshape("int32"))
>>> yy._name, yy.dshape
('y', dshape("int32"))
```

`blaze.expr.broadcast.`**`broadcast_collect`**`(`*expr, broadcastable=(<class 'blaze.expr.expressions.Map'>, <class 'blaze.expr.expressions.Field'>, <class 'blaze.expr.datetime.DateTime'>, <class 'blaze.expr.arithmetic.UnaryOp'>, <class 'blaze.expr.arithmetic.BinOp'>, <class 'blaze.expr.expressions.Coerce'>, <class 'blaze.expr.collections.Shift'>, <class 'blaze.expr.strings.Like'>, <class 'blaze.expr.strings.StrCat'>), want_to_broadcast=(<class 'blaze.expr.expressions.Map'>, <class 'blaze.expr.datetime.DateTime'>, <class 'blaze.expr.arithmetic.UnaryOp'>, <class 'blaze.expr.arithmetic.BinOp'>, <class 'blaze.expr.expressions.Coerce'>, <class 'blaze.expr.collections.Shift'>, <class 'blaze.expr.strings.Like'>, <class 'blaze.expr.strings.StrCat'>), no_recurse=None*`)`

Collapse expression down using Broadcast - Tabular cases only

Expressions of type Broadcastables are swallowed into Broadcast operations

```
>>> t = symbol('t', 'var * {x: int, y: int, z: int, when: datetime}')
>>> expr = (t.x + 2*t.y).distinct()
```

```
>>> broadcast_collect(expr)
distinct(Broadcast(_children=(t,), _scalars=(t,), _scalar_expr=t.x + (2 * t.y)))
```

```
>>> from blaze import exp
>>> expr = t.x + 2 * exp(-(t.x - 1.3) ** 2)
>>> broadcast_collect(expr)
Broadcast(_children=(t,), _scalars=(t,), _scalar_expr=t.x + (2 * (exp(-((t.x - 1.
↪3) ** 2)))))
```

class `blaze.expr.datetime.`**`DateTime`**

Superclass for datetime accessors

class `blaze.expr.split_apply_combine.`**`By`**

Split-Apply-Combine Operator

Examples

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {name: string, amount: int, id: int}')
>>> e = by(t['name'], total=t['amount'].sum())
```

```
>>> data = [['Alice', 100, 1],
...         ['Bob', 200, 2],
...         ['Alice', 50, 3]]
```

```
>>> from blaze.compute.python import compute
>>> sorted(compute(e, data))
[('Alice', 150), ('Bob', 200)]
```

`blaze.expr.split_apply_combine.`**`count_values`**(*expr*, *sort=True*)

Count occurrences of elements in this column

Sort by counts by default Add `sort=False` keyword to avoid this behavior.

2.1.17 Release Notes

Release 0.11.0

Release 0.11.0

New Expressions

- Many new string utility expressions were added that follow the Pandas vectorized string methods API closely http://pandas.pydata.org/pandas-docs/stable/text.html#text-string-methods. These are gathered under the `.str` sub-namespace, allowing the user to say:

  ```
  t.col.str.lower()
  ```

  to compute a new column with the string contents lowercased.

- Likewise, many new datetime utility expressions were added to the `.dt` sub-namespace, following Pandas vectorized datetime methods API http://pandas.pydata.org/pandas-docs/stable/timeseries.html.

Improved Expressions

None

New Backends

None

Improved Backends

None

Experimental Features

None

API Changes

- The following functions were deprecated in favor of equivalent functions without the *str_* name prefix:

deprecated function	replacement function
`str_len()`	`len()`
`str_upper()`	`upper()`
`str_lower()`	`lower()`
`str_cat()`	`cat()`

Bug Fixes

None

Miscellaneous

None

Release 0.10.2

Release 0.10.2

New Expressions

None

Improved Expressions

- Adds support for `any` and `all` to the sql backend (#1511).

New Backends

None

Improved Backends

- To allow access to the `map` and `apply` expressions in client / server interactions when in a trusted environment, new `_trusted` versions of the several default `SerializationFormat` instances were added. These trusted variants allow (de)serialization of builtin functions, NumPy functions, and Pandas functions. They are intentially kept separate from the default versions to ensure they are not accidentally enabled in untrusted environments (#1497 #1504).

Experimental Features

None

API Changes

None

Bug Fixes

- Fixed a bug with `to_tree()` and `slice` objects. Have to change the order of cases in `to_tree()` to ensure `slice` objects are handled before lookups inside the `names` namespace (#1516).
- Perform more input validation for `sort()` expression arguments (#1517).
- Fixes issue with string and datetime coercions on Pandas objects (#1519 #1524).
- Fixed a bug with `isin` and `Selections` on sql selectables (#1528).

Miscellaneous

Expression Identity Rework

Expression are now memoized by their inputs. This means that two identical expressions will always be the same object, or that `a.isidentical(b)` is the same as `a is b`. `isidentical` is called hundreds of thousands of times in a normal blaze workload. Moving more work to expression construction time has been shown to dramatically improve compute times when the expressions grow in complexity or size. In the past, blaze was spending linear time relative to the expression size to compare expressions because it needed to recurse through the entire expression tree but now it can do `isidentical` in constant time.

Users should still use `a.isidentical(b)` instead of `a is b` because we reserve the right to add more arguments or change the implementation of `isidentical` in the future.

Release 0.10.1

Release 0.10.1

Date TBD

New Expressions

None

Improved Expressions

None

New Backends

None

Improved Backends

- Blaze server's `/add` endpoint was enhanced to take a more general payload (#1481).
- Adds consistency check to blaze server at startup for YAML file and dynamic addition options (#1491).

Experimental Features

- The `str_cat()` expression was added, mirroring Pandas' `Series.str.cat()` API (#1496).

API Changes

None

Bug Fixes

- The content type specification parsing was improved to accept more elaborate headers (#1490).
- The discoverablility consistency check is done before a dataset is dynamically added to the server (#1498).

Miscellaneous

None

Release 0.10.0

Release 0.10.0

Date TBD

New Expressions

- The `sample` expression allows random sampling of rows to facilitate interactive data exploration (#1410). It is implemented for the Pandas, Dask, SQL, and Python backends.
- Adds *`coalesce()`* expression which takes two arguments and returns the first non missing value. If both are missing then the result is missing. For example: `coalesce(1, 2) == 1`, `coalesce(None, 1) == 1`, and `coalesce(None, None) == None`. This is inspired by the sql function of the same name (#1409).
- Adds *`cast()`* expression to reinterpret an expression's dshape. This is based on C++ `reinterpret_cast`, or just normal C casts. For example: `symbol('s', 'int32').cast('uint32').dshape == dshape('uint32')`. This expression has no affect on the computation, it merely tells blaze to treat the result of the expression as the new dshape. The compute definition for `cast` is simply:

  ```
  @dispatch(Cast, object)
  def compute_up(expr, data, **kwargs):
      return data
  ```

  (#1409).

Improved Expressions

- The test suite was expanded to validate proper expression input error handling (#1420).
- The `truncate()` function was refactored to raise an exception for incorrect inputs, rather than using assertions (#1443).
- The docstring for *`Merge`* was expanded to include examples using *`Label`* to control the ordering of the columns in the result (#1447).

New Backends

None

Improved Backends

- Adds `greatest` and `least` support to the sql backend (#1428).
- Generalize `Field` to support `collections.Mapping` object (#1467).

Experimental Features

- The `str_upper` and `str_lower` expressions were added for the Pandas and SQL backends (#1462). These are marked experimental since their names are subject to change. More string methods will be added in coming versions.

API Changes

- The `strlen` expression was deprecated in favor of `str_len` (#1462).
- Long deprecated `Table()` and `TableSymbol()` were removed (#1441). The `TableSymbol` tests in `test_table.py` were migrated to `test_symbol.py`.

- `Data()` has been deprecated in favor of *`data()`*. `InteractiveSymbol` has been deprecated and temporarily replaced by `_Data`. These deprecations will be in place for the 0.10 release. In the 0.11 release, `_Data` will be renamed to `Data`, calls to *`data()`* will create `Data` instances, and `InteractiveSymbol` will be removed (#1431 and #1421).
- `compute()` has a new keyword argument `return_type` which defaults to `'native'` (#1401, #1411, #1417), which preserves existing behavior. In the 0.11 release, `return_type` will be changed to default to `'core'`, which will `odo` non-core backends into core backends as the final step in a call to `compute`.
- Due to API instability and on the recommendation of DyND developers, we removed the DyND dependency temporarily (#1379). When DyND achieves its 1.0 release, DyND will be re-incorporated into Blaze. The existing DyND support in Blaze was rudimentary and based on an egregiously outdated and buggy version of DyND. We are aware of no actual use of DyND via Blaze in practice.
- The *`Expr`* `__repr__` method's triggering of implicit computation has been deprecated. Using this aspect of Blaze will trigger a `DeprecationWarning` in version 0.10, and this behavior will be replaced by a standard (boring) `__repr__` implementation in version 0.11. Users can explicitly trigger a computation to see a quick view of the results of an interactive expression by means of the `peek()` method. By setting the `use_new_repr` flag to `True`, users can use the new (boring) `__repr__` implementation in version 0.10 (#1414 and #1395).

Bug Fixes

- The `str_upper` and `str_lower` schemas were fixed to pass through their underlying `_child`'s schema to ensure option types are handled correctly (#1472).
- Fixed a bug with Pandas' implementation of `compute_up` on *`Broadcast`* expressions (#1442). Added tests for Pandas frame and series and dask dataframes on `Broadcast` expressions.
- Fixed a bug with *`Sample`* on SQL backends (#1452 #1423 #1424 #1425).
- Fixed several bugs relating to adding new datasets to blaze server instances (#1459). Blaze server will make a best effort to ensure that the added dataset is valid and loadable; if not, it will return appropriate HTTP status codes.

Miscellaneous

- Adds logging to server compute endpoint. Includes expression being computed and total time to compute. (#1436)
- Merged the `core` and `all` conda recipes (#1451). This simplifies the build process and makes it consistent with the single `blaze` package provided by the Anaconda distribution.
- Adds a `--yaml-dir` option to `blaze-server` to indicate the server should load path-based `yaml` resources relative to the yaml file's directory, not the CWD of the process (#1460).

Release 0.9.1

Release 0.9.1

Date December 17th, 2015

New Expressions

Improved Expressions

- The `Like` expression was improved to support more general `Select` queries that result from *`Join`* operations rather than soely `ColumnElement` queries (#1371 #1373).
- Adds `std` and `var` reductions for `timedelta` types for sql and pandas backends (#1382).

New Backends

None

Improved Backends

- Blaze Server no longer depends on *Bokeh* for CORS handling, and now uses the *flask-cors* third-party package (#1378).

Experimental Features

None

API Changes

None

Bug Fixes

- Fixed a `blaze-server` entry point bug regarding an ambiguity between the `spider()` function and the **:module:‘~blaze.server.spider‘** module (#1385).
- Fixed `blaze.expr.datetime.truncate()` handling for the sql backend (#1393).
- Fix `blaze.expr.core.isidentical()` to check the `_hashargs` instead of the `_args`. This fixes a case that caused objects that hashed the same to not compare equal when somewhere in the tree of `_args` was a non hashable structure (#1387).
- Fixed a type issue where `datetime - datetime :: datetime` instead of `timedelta` (#1382).
- Fixed a bug that caused *`coerce()`* to fail when computing against `ColumnElements`. This would break `coerce` for many sql operations (#1382).
- Fixed reductions over `timedelta` returning `float` (#1382).
- Fixed interactive repr for `timedelta` not coercing to `timedelta` objects (#1382).
- Fixed weakkeydict cache failures that were causing `.dshape` lookups to fail sometimes (#1399).
- Fixed relabeling columns over selects by using *reconstruct_select* (:issue: *1471*).

Miscellaneous

- Removed support for Spark 1.3 (#1386) based on community consensus.
- Added `blaze.utils.literal_compile()` for converting sqlalchemy expressions into sql strings with bind parameters inlined as sql literals. `blaze.utils.normalize()` now accepts a sqlalchemy selectable and uses `literal_compile` to convert into a string first (#1386).

Release 0.9.0

Release 0.9.0

Date December 17th, 2015

New Expressions

- Add a *`shift()`* expression for shifting data backwards or forwards by *N* rows (#1266).

Improved Expressions

New Backends

- Initial support for dask.dataframe has been added, see (#1317). Please send feedback via an issue or pull request if we missed any expressions you need.

Improved Backends

- Adds support for *`tail()`* in the sql backend (#1289).
- Blaze Server now supports dynamically adding datasets (#1329).
- Two new keyword only arguments are added to `compute()` for use when computing against a `Client` object:
  1. `compute_kwargs`: This is a dictionary to send to the server to use as keyword arguments when calling `compute` on the server.
  2. `odo_kwargs`: This is a dictionary to send to the server to use as keyword arguments when calling `odo` on the server.

  This extra information is completely optional and will have different meanings based on the backend of the data on the server (#1342).
- Can now point `Data()` to URLs of CSVs (#1336).

Experimental Features

- There is now support for joining tables from multiple sources. This is **very experimental** right now, so use it at your own risk. It currently only works with things that fit in memory (#1282).
- Foreign columns in database tables that have foreign key relationships can now be accessed with a more concise syntax (#1192).

API Changes

- Removed support for Python 2.6 (#1267).
- Removed support for Python 3.3 (#1270).
- When a CSV file consists of all strings, you must pass `has_header=True` when using the `Data` constructor (#1254).
- Comparing `date` and `datetime` datashaped things to the empty string now raises a `TypeError` (#1308).
- `Like` expressions behave like a predicate, and operate on columns, rather than performing the selection for you on a table (#1333, #1340).
- `blaze.server.Server.run()` no longer retries binding to a new port by default. Also, positional arguments are no longer forwarded to the inner flask app's `run` method. All keyword arguments not consumed by the blaze server `run` are still forwarded (#1316).
- `Server` represents datashapes in a canonical form with consistent linebreaks for use by non-Python clients (#1361).

Bug Fixes

- Fixed a bug where `Merge` expressions would unpack option types in their fields. This could cause you to have a table where `expr::{a:  int32}` but `expr.a::?int32`. Note that the dotted access is an option (#1262).
- Explicitly set `Node.__slots__` and `Expr.__slots__` to `()`. This ensures instances of slotted subclasses like `Join` do not have a useless empty `__dict__` attribute (#1274 and #1268).
- Fixed a bug that prevented creating a `InteractiveSymbol` that wrapped `nan` if the dshape was `datetime`. This now correctly coerces to *NaT* (#1272).
- Fixed an issue where blaze client/server could not use *isin* expressions because the `frozenset` failed to serialize. This also added support for rich serialization over json for things like datetimes (#1255).
- Fixed a bug where `len` would fail on an interactive expression whose resources were sqlalchemy objects (#1273).
- Use aliases instead of common table expressions (CTEs) because MySQL doesn't have CTEs (#1278).
- Fixed a bug where we couldn't display an empty string identifier in interactive mode (#1279).
- Fixed a bug where comparisons with optionals that should have resulted in optionals did not (#1292).
- Fixed a bug where `Join.schema` would not always be instantiated (#1288).
- Fixed a bug where comparisons to a empty string magically converted the empty string to `None` (#1308).
- Fix the `retry` kwarg to the blaze server. When `retry` is False, an exception is now properly raised if the port is in use. (#1316).
- Fixed a bug where leaves that only appeared in the predicate of a selection would not be in scope in time to compute the predicate. This would cause whole expressions like `a[a > b]` to fail because `b` was not in scope (#1275).
- Fix a broken test on machines that don't allow postgres to read from the local filesystem (#1323).
- Updated a test to reflect changes from odo #366 (#1323).
- Fixed pickling of blaze expressions with interactive symbols (#1319).
- Fixed repring partials in blaze expression to show keyword arguments (#1319).

- Fixed a memory leak that would preserve the lifetime of any blaze expression that had cached an attribute access (#1335).
- Fixed a bug where `common_subexpression()` gave the wrong answer (#1325, #1338).
- `BinaryMath` operations without numba installed were failing (#1343).
- win32 tests were failing for `hypot` and `atan2` due to slight differences in numpy vs numba implementations of those functions (#1343).
- Only start up a `ThreadPool` when using the h5py backend (#1347, #1331).
- Fix return type for sum and mean reductions whose children have a `Decimal` dshape.

Miscellaneous

- `blaze.server.Server.run()` now uses `warnings.warn()` instead of `print` when it fails to bind to a port and is retrying (#1316).
- Make expressions (subclasses of Expr) weak referencable (:issue:‘1319).
- Memoize dshape and schema methods (#1319).
- Use `pandas.DataFrame.sort_values()` with pandas version >= 0.17.0 (#1321).

Release 0.8.3

Release 0.8.3

Date September 15, 2015

New Expressions

- Adds *`Tail`* which acts as an opposite to head. This is exposed throught the *`tail()`* function. This returns the last `n` elements from a collection. (#1187)
- Adds *`notnull`* returning an indicator of whether values in a field are null (#697, #733)

Improved Expressions

- *`Distinct`* expressions now support an *on* parameter to allow distinct on a subset of columns (#1159)
- *`Reduction`* instances are now named as their class name if their `_child` attribute is named `'_'` (#1198)
- *`Join`* expressions now promotes the types of the fields being joined on. This allows us to join things like `int32` and `int64` and have the result be an int64. This also allows us to join any type `a` with `?a`. (#1193, #1218).

New Backends

Improved Backends

- Blaze now tries a bit harder to avoid generating `ScalarSelects` when using the SQL backend (#1201, #1205)
- ReLabel expressions on the SQL backend are now flattened (#1217).

API Changes

- Serialization format in blaze server is now passed in as a mimetype (#1176)
- We only allow and use HTTP `POST` requests when sending a computation to Blaze server for consistency with the HTTP spec (#1172)
- Allow `Client` objects to explicitly disable verification of ssl certificates by passing `verify_ssl=False`. (#1170)
- Enable basic auth for the blaze server. The server now accepts an `authorization` keyword which must be a callable that accepts an object holding the username and password, or None if no auth was given and returns a bool indicating if the request should be allowed. `Client` objects can pass an optional `auth` keyword which should be a tuple of (username, password) to send to the server. (#1175)
- We now allow *`Distinct`* expressions on `ColumnElement` to be more general and let things like `sa.sql.elements.Label` objects through (#1212)
- Methods now take priority over field names when using attribute access for *`Field`* instances to fix a bug that prevented accessing the method at all (#1204). Here's an example of how this works:

```
>>> from blaze import symbol
>>> t = symbol('t', 'var * {max: float64, isin: int64, count: int64}')
>>> t['count'].max()
t.count.max()
>>> t.count()  # calls the count method on t
t.count()
>>> t.count.max()  # AttributeError
Traceback (most recent call last):
    ...
AttributeError: ...
```

Bug Fixes

- Upgrade versioneer so that our version string is now **PEP 440** compliant (#1171)
- Computed columns (e.g., the result of a call to *`transform()`*) can now be accessed via standard attribute access when using the SQL backend (#1201)
- Fixed a bug where blaze server was depending on an implementation detail of CPython regarding `builtins` (#1196)
- Fixed incorrect SQL generated by count on a subquery (#1202).
- Fixed an `ImportError` generated by an API change in dask.
- Fixed an issue where columns were getting trampled if there were column name collisions in a sql join. (#1208)
- Fixed an issue where arithmetic in a Merge expression wouldn't work because arithmetic wasn't defined on `sa.sql.Select` objects (#1207)
- Fixed a bug where the wrong value was being passed into `time()` (#1213)
- Fixed a bug in sql relabel that prevented relabeling anything that generated a subselect. (#1216)
- Fixed a bug where methods weren't accessible on fields with the same name (#1204)
- Fixed a bug where optimized expressions going into a pandas group by were incorrectly assigning extra values to the child DataFrame (#1221)

- Fixed a bug where multiple same-named operands caused incorrect scope to be constructed ultimately resulting in incorrect results on expressions like `x + x + x` (#1227). Thanks to @llllllllll and @jcrist for discussion around solving the issue.
- Fixed a bug where `minute()` and `Minute` were not being exported which made them unusable from the blaze server (#1232).
- Fixed a bug where repr was being called on data resources rather than string, which caused massive slowdowns on largish expressions running against blaze server (#1240, #1247).
- Skip a test on Win32 + Python 3.4 and PyTables until this gets sorted out on the library side (#1251).

Miscellaneous

- We now run tests against pandas master to catch incompatibility issues (#1243).

Release 0.8.2

Release 0.8.2

Date July 9th, 2015

Bug Fixes

- Fix broken sdist tarball

Release 0.8.1

Release 0.8.1

Date July 7th, 2015

New Expressions

- String arithmetic is now possible across the numpy and pandas backends via the + (concatenation) and `*` (repeat) operators (#1058).
- Datetime arithmetic is now available (#1112).
- Add a *`Concat`* expression that implements Union-style operations (#1128).
- Add a *`Coerce`* expression that casts expressions to a different datashape. This maps to `astype` in numpy and `cast` in SQL (#1137).

Improved Expressions

- *`ReLabel`* expressions `repr` differently depending on whether the existing column names are valid Python variable names (#1070).

New Backends

None

Improved Backends

- *In-memory* merges of CSV files are now possible (#1121).
- Tie blueprint registration to data registration (#1061).
- Don't catch import error when flask doesn't exist, since blaze does this in its `__init__.py` (#1087).
- Multiple serialization formats including JSON, pickle, and msgpack are now available. Additionally, one can add custom serialization formats with this implementation (#1102, #1122).
- Add a `'names'` field to the response of the `compute.<format>` route for Bokeh compatibility (#1129).
- Add cross origin resource sharing for Bokeh compatibility (#1134).
- Add a command line interface (#1115).
- Add a way to tell the blaze server command line interface what to server via a YAML file (#1115).
- Use aliases to allow expressions on the SQL backend that involve a multiple step reduction operation (#1066, #1126).
- Fix unary not operator ~ (#1091).
- Postgres uses == to compare `NaN` so we do it that way as well for the postgresql backend (#1123).
- Find table inside non-default schema when serving off a SQLAlchemy `MetaData` object (#1145).

API Changes

- Remove old `ExprClient()`. Use `Client` instead (#1154).
- Make sort + slice and sort + slice semantics of the SQL backend reflect those of numpy (#1125).
- The following syntax is no longer allowed for Blaze server (#1154):

```
>>> Data('blaze://localhost::accounts')  # raises an error
```

  Use this syntax instead:

```
>>> Data('blaze://localhost').accounts  # valid
```

Bug Fixes

- Handle SQLAlchemy API churn around reference of `ColumnElement` objects in the 1.0.x series (#1071, #1076).
- Obscure hashing bug when passing in both a pandas Timestamp and a `datetime.datetime` object. Both objects hash to the same value but don't necessarily compare equal; this makes Python call `__eq__` which caused an `Eq` expression to be constructed (#1097).
- Properly handle `And` expressions that involve the same field in MongoDB (#1099).
- Handle Dask API changes (#1114).
- Use the `date` function in SQLAlchemy when getting the `date` attribute of a `datetime` dshaped expression. Previously this was calling extract, which is incorrect for the postgres backend (#1120).
- Fix API compatibility with different versions of psutil (#1136).
- Use explicit `int64` comparisons on Windows, since the default values may be different (#1148).

- Fix name attribute propagation in pandas `Series` objects (#1152).
- Raise a more informative error when trying to subset with an unsupported expression in the MongoDB backend (#1155).

Release 0.7.3

- General maturation of many backends through use.
- Renamed `into` to `odo`

Release 0.7.0

- Pull out data migration utilities to `into` project
- Out-of-core CSV support now depends on chunked pandas computation
- h5py and bcolz backends support multi-threading/processing
- Remove `data` directory including `SQL`, `HDF5` objects. Depend on standard types within other projects instead (e.g. `sqlalchemy.Table`, `h5py.Dataset`, ...)
- Better support SQL nested queries for complex queries
- Support databases, h5py files, servers as first class datasets

Release 0.6.6

- Not intended for public use, mostly for internal build systems
- Bugfix

Release 0.6.5

- Improve uri string handling #715
- Various bug fixes #715

Release 0.6.4

- Back CSV with `pandas.read_csv`. Better performance and more robust unicode support but less robust missing value support (some regressions) #597
- Much improved SQL support #626 #650 #652 #662
- Server supports remote execution of computations, not just indexing #631
- Better PyTables and datetime support #608 #639
- Support SparkSQL #592

Release 0.6.3

- by takes only two arguments, the grouper and apply child is inferred using common_subexpression
- Better handling of pandas Series object
- Better printing of empty results in interactive mode
- **Regex dispatched resource function bound to Table, e.g.** `Table('/path/to/file.csv')`

Release 0.6.2

- Efficient CSV to SQL migration using native tools #454
- Dispatched `drop` and `create_index` functions #495
- DPlyr interface at `blaze.api.dplyr`. #484
- **Various bits borrowed from that interface**
  - `transform` function adopted to main namespace
  - `Summary` object for named reductions
  - Keyword syntax in `by` and `merge` e.g. `by(t, t.col, label=t.col2.max(), label2=t.col2.min())`
- New Computation Server #527
- Better PyTables support #487 #496 #526

Release 0.6.1

- More consistent behavior of `into`
- `bcolz` backend
- Control namespace leakage

Release 0.6

- Nearly complete rewrite
- Add abstract table expression system
- Translate expressions onto a variety of backends
- Support Python, NumPy, Pandas, h5py, sqlalchemy, pyspark, PyTables, pymongo

Release 0.5

- HDF5 in catalog.
- Reductions like any, all, sum, product, min, max.
- Datetime design and some initial functionality.
- Change how Storage and ddesc works.
- Some preliminary rolling window code.
- Python 3.4 now in the test harness.

Release 0.4.2

- Fix bug for compatibility with numba 0.12
- Add sql formats
- Add hdf5 formats
- Add support for numpy ufunc operators

Release 0.4.1

- Fix bug with compatibility for numba 0.12

Release 0.4

- Split the datashape and blz modules out.
- Add catalog and server for blaze arrays.
- Add remote arrays.
- Add csv and json persistence formats.
- Add python3 support
- Add scidb interface

Release 0.3

- Solidifies the execution subsystem around an IR based on the pykit project, as well as a ckernel abstraction at the ABI level.
- Supports ufuncs running on ragged array data.
- Cleans out previous low level data descriptor code, the data descriptor will have a higher level focus.
- Example out of core groupby operation using BLZ.

Release 0.2

- Brings in dynd as a required dependency for in-memory data.

Release 0.1

- Initial preview release

2.1.18 Contributors

Current Core Developers

- Phillip Cloud
- Joe Jevnik
- Matt Rocklin

- Kurt Smith

Contributors

- Andy R. Terrel
- Mark Wiebe
- Majid alDosari
- Francesc Alted
- Tyler Alumbaugh
- Dav Clark
- Stephen Diehl
- Christine Doig
- Mark Florisson
- Damien Garaud
- Valentin Haenel
- Lila Hickey
- Maggie Mari
- Travis Oliphant
- Milos Popovic
- Stan Seibert
- Hugo Shi
- Oscar Villellas Guillén
- Peter Wang
- Matt Westcott
- Ben Zaitlen

2.1.19 Legal

Blaze is a community project much like Numpy. It is released under a permissive BSD license.

The BSD 2-clause license allows you almost unlimited freedom with the software so long as you include the BSD copyright notice in it (found below).

Continuum Analytics sponsors development on Blaze.

License:

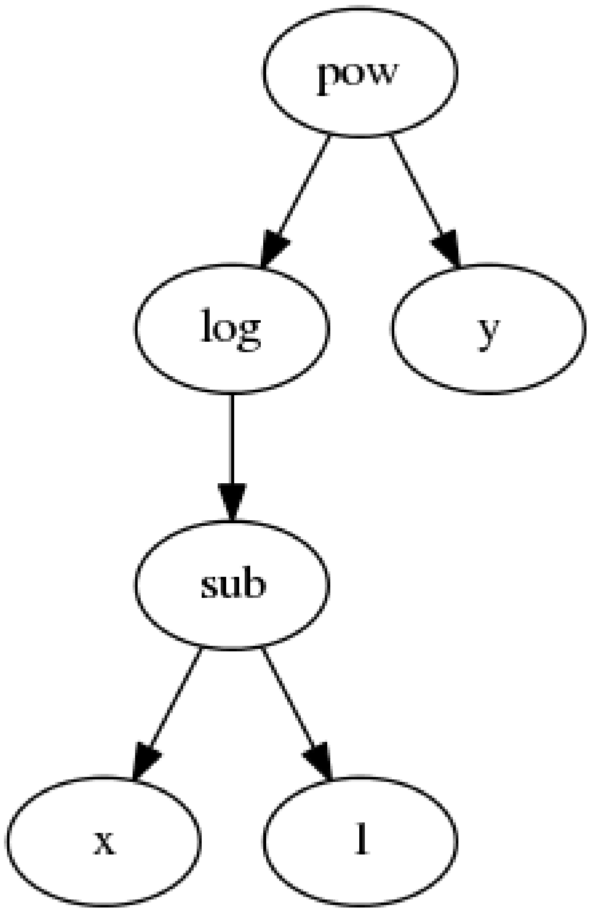
```
Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

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.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Internal

2.1.20 Expression Design

At its core Blaze expresses analytics queries symbolicly. It represents these queries as an abstract expression tree. This tree dictates user interaction, optimizations, and is eventually walked to translate into other computational systems. Deep knowledge of abstact expressions is not necessary to use Blaze; it is essential to develop in it.



Consider the following example:

```
>>> from blaze import symbol, log
>>> x = symbol('x', 'int64')
>>> y = symbol('y', 'float32')
>>> z = log(x - 1)**y
```

We represent the mathematical expression `log(x - 1)**y` as a tree where every operator (e.g. `log`, `pow`) is a node whose children are its arguments. All Blaze expressions and indeed all expressions in any programming language can be represnted this way. Blaze encodes this tree as a data structure where every node is an object with type corresponding to its operation. For example there exists the following classes

```
class pow(Expr):
    ...
class sub(Expr):
    ...
class log(Expr):
    ...
```

And our expression, written explicitly as Blaze sees it, would look like the following:

```
>>> from blaze.expr import Pow, Sub, log, symbol
>>> z = Pow(log(Sub(symbol('x', 'int64'), 1)),
...         symbol('y', 'float32'))
>>> z
(log(x - 1)) ** y
```

Common Expression Attributes

There are a few important attributes and methods to investigate a Blaze expression.

- `__class__`: The type of a node corresponds to its operation:

```
type(z) == pow
```

- `dshape`: The shape and type of the data for this expression. This is often more important that the actual type of the node:

```
.. code-block:: python
```

```
>>> s = symbol('s', 'var * float64')
>>> s.dshape
dshape("var * float64")
>>> s.mean().dshape
dshape("float64")
```

- `._args`: All children of a node, including parameters. Args may include both Blaze expressions and other variables like strings:

```
z._args == (log(x - 1), y)
x._args == ('x', 'int64')
```

- `._hashargs`: The arguments in a form that is hashable. This is only different from `._args` when the arguments contain things that are not normally hashable with the `hash` builtin function.
- `._inputs`: All children of a node, excluding parameters. All inputs are Blaze expressions.:

```
z._inputs == (log(x - 1), y)
x._inputs == ()
```

- `._leaves()`: The symbols at the bottom of the expression tree:

```
z._leaves() == (x, y)
x._leaves() == (x,)
```

By recursively traversing either `._args` or `._inputs` you may inspect or transform either all information pertaining to the tree or just the expressions.

To clear up confusion between `._args` from `._inputs` consider the following tabular example with sort. `._inputs` contains only other Blaze expressions while `._args` also contains parameters like the string `'balance'`.

```
>>> t = symbol('t', 'var * {name: string, balance: int}')
>>> expr = t.sort('balance', ascending=True)
>>> expr._args
(<`t` symbol; dshape='...'>, 'balance', True)
>>> expr._inputs
(<`t` symbol; dshape='...'>,)
```

Some convenience functions for common traversals already exist:

- `._subs`: replace nodes in the tree according to replacement dictionary:

```
>>> z
(log(x - 1)) ** y
>>> z._subs({'x': 'a', 'y': 'b'})
(log(a - 1)) ** b
```

- `._subterms`, a traversal along `._inputs`:

```
>>> list(z._subterms())
[(log(x - 1)) ** y, log(x - 1), x - 1, <`x` symbol; dshape=...>, <`y` symbol;␣
↪dshape=...>]
```

- `._traverse`, a traversal along `._args`:

```
>>> list(z._traverse())
[(log(x - 1)) ** y,
 log(x - 1),
 x - 1,
 <`x` symbol; dshape=...>,
 'x',
 dshape("int64"),
 0,
 1,
 <`y` symbol; dshape=...>,
 'y',
 dshape("float32"),
 0]
```

2.1.21 Expressions

Blaze expressions describe computational workflows symbolically. They allow developers to architect and check their computations rapidly before applying them to data. These expressions can then be compiled down to a variety of supported backends.

Tables

Table expressions track operations found in relational algebra or your standard Pandas/R DataFrame object. Operations include projecting columns, filtering, mapping and basic mathematics, reductions, split-apply-combine (group by) operations, and joining. This compact set of operations can express a surprisingly large set of common computations. They are widely supported.

Symbol

A `Symbol` refers to a single collection of data. It must be given a name and a datashape.

```
>>> from blaze import symbol
>>> accounts = symbol('accounts', 'var * {id: int, name: string, balance: int}')
```

Projections, Selection, Arithmetic

Many operations follow from standard Python syntax, familiar from systems like NumPy and Pandas.

The following example defines a collection, `accounts`, and then selects the names of those accounts with negative balance.

```
>>> accounts = symbol('accounts', 'var * {id: int, name: string, balance: int}')
>>> deadbeats = accounts[accounts.balance < 0].name
```

Internally this doesn't do any actual work because we haven't specified a data source. Instead it builds a symbolic representation of a computation to execute in the future.

```
>>> deadbeats
accounts[accounts.balance < 0].name
>>> deadbeats.dshape
dshape("var * string")
```

Split-apply-combine, Reductions

Blaze borrows the `by` operation from `R` and `Julia`. The `by` operation is a combined `groupby` and reduction, fulfilling split-apply-combine workflows.

```
>>> from blaze import by
>>> by(accounts.name,                 # Splitting/grouping element
...    total=accounts.balance.sum())  # Apply and reduction
by(accounts.name, total=sum(accounts.balance))
```

This operation groups the collection by name and then sums the balance of each group. It finds out how much all of the "Alice"s, "Bob"s, etc. of the world have in total.

Note the reduction `sum` in the third apply argument. Blaze supports the standard reductions of numpy like `sum`, `min`, `max` and also the reductions of Pandas like `count` and `nunique`.

Join

Collections can be joined with the `join` operation, which allows for advanced queries to span multiple collections.

```
>>> from blaze import join
>>> cities = symbol('cities', 'var * {name: string, city: string}')
>>> join(accounts, cities, 'name')
Join(lhs=accounts, rhs=cities, _on_left='name', _on_right='name', how='inner',␣
↪suffixes=('_left', '_right'))
```

If given no inputs, `join` will join on all columns with shared names between the two collections.

```
>>> shared_names = join(accounts, cities)
```

Type Conversion

Type conversion of expressions can be done with the `coerce` expression. Here's how to compute the average account balance for all the deadbeats in my `accounts` table and then cast the result to a 64-bit integer:

```
>>> deadbeats = accounts[accounts.balance < 0]
>>> avg_deliquency = deadbeats.balance.mean()
>>> chopped = avg_deliquency.coerce(to='int64')
>>> chopped
mean(accounts[accounts.balance < 0].balance).coerce(to='int64')
```

Other

Blaze supports a variety of other operations common to our supported backends. See our API docs for more details.

2.1.22 Backends

Blaze backends include projects like streaming Python, Pandas, SQLAlchemy, MongoDB, PyTables, and Spark. Most Blaze expressions can run well on any of these backends, allowing developers to easily transition their computation to changing performance needs.

Existing Backends

Streaming Python

via toolz and cytoolz

Blaze can operate on core Python data structures like lists, tuples, ints and strings. This can be useful both in small cases like rapid prototyping or unit testing but also in large cases where streaming computation is desired.

The performance of Python data structures like `dict` make Python a surprisingly powerful platform for data-structure bound computations commonly found in split-apply-combine and join operations. Additionally, Python's support for lazy iterators (i.e. generators) means that it can easily support *streaming* computations that pull data in from disk, taking up relatively little memory.

Pandas

Pandas DataFrames are the gold standard for in-memory data analytics. They are fast, intuitive, and come with a wealth of additional features like plotting, and data I/O.

SQLAlchemy

Blaze levarages the SQLAlchemy project, which provides a uniform interface over the varied landscape of SQL systems. Blaze manipulates SQLAlchemy expressions which are then compiled down to SQL query strings of the appropriate backend.

The prevalance of SQL among data technologies makes this backend particularly useful. Databases like Impala and Hive have SQLAlchemy dialects, enabling easy Blaze interoperation.

MongoDB

Blaze drives MongoDB through the pymongo interface and is able to use many of the built in operations such as aggregration and group by.

PyTables

PyTables provides compressed Table objects backed by the popular HDF5 library. Blaze can compute simple expressions using PyTables, such as elementwise operations and row-wise selections.

Spark

Spark provides resilient distributed in-memory computing and easy access to HDFS storage. Blaze drives Spark through the PySpark interface.

Benefits of Backend Agnostic Computation

For maximum performance and expressivity it is best to use the backends directly. Blaze is here when absolute customization is not required.

Familiarity

Users within the numeric Python ecosystem may be familiar with the NumPy and Pandas interfaces but relatively unfamiliar with SQL or the functional idioms behind Spark or Streaming Python. In this case Blaze provides a familiar interface which can drive common computations in these more exotic backends.

Prototyping and Testing

Blaze allows you to prototype and test your computation on a small dataset using Python or Pandas and then scale that computation up to larger computational systems with confidence that nothing will break.

A changing hardware landscape drives a changing software landscape. Analytic code written for systems today may not be relevant for systems five years from now. Symbolic systems like Blaze provide some stability on top of this rapidly changing ecosystem.

Static Analysis

Not yet implemented

Blaze is able to inspect and optimize your computation before it is run. Common optimizations include loop fusion, rearranging joins and projections to minimize data flow, etc..

2.1.23 Interactive Expressions

Internally Blaze is abstract; this limits interactivity. Blaze *interactive expressions* resolve this issue and provide a smooth experience to handling foreign data.

Expressions with Data

Internally Blaze separates the intent of the computation from the data/backend. While powerful, this abstract separation limits interactivity, one of the core goals of Blaze.

Blaze *interactive expressions* are like normal expressions but their leaves may hold on to a concrete data resource (like a DataFrame or SQL database.) This embeds a specific data context, providing user interface improvements at the cost of full generality.

Example

We create an interactive expression by calling the `data` constructor on any object or URI with which Blaze is familiar.

```
>>> from blaze import data, Symbol
>>> from blaze.utils import example
>>> db = data('sqlite:///%s' % example('iris.db'))  # an interactive expression
>>> db.iris
    sepal_length  sepal_width  petal_length  petal_width      species
0            5.1          3.5           1.4          0.2  Iris-setosa
```

```
1            4.9          3.0           1.4          0.2   Iris-setosa
2            4.7          3.2           1.3          0.2   Iris-setosa
3            4.6          3.1           1.5          0.2   Iris-setosa
4            5.0          3.6           1.4          0.2   Iris-setosa
5            5.4          3.9           1.7          0.4   Iris-setosa
6            4.6          3.4           1.4          0.3   Iris-setosa
7            5.0          3.4           1.5          0.2   Iris-setosa
8            4.4          2.9           1.4          0.2   Iris-setosa
9            4.9          3.1           1.5          0.1   Iris-setosa
...

>>> db.iris.species.<tab>
db.iris.species.columns       db.iris.species.max
db.iris.species.count         db.iris.species.min
db.iris.species.count_values  db.iris.species.ndim
db.iris.species.distinct      db.iris.species.nunique
db.iris.species.dshape        db.iris.species.relabel
db.iris.species.expr          db.iris.species.resources
db.iris.species.fields        db.iris.species.schema
db.iris.species.head          db.iris.species.shape
db.iris.species.isidentical   db.iris.species.sort
db.iris.species.label         db.iris.species.species
db.iris.species.like          db.iris.species.to_html
db.iris.species.map

>>> db.iris.species.distinct()
          species
0     Iris-setosa
1  Iris-versicolor
2   Iris-virginica
```

In the case above `db` is a `Symbol`, just like any normal Blaze leaf expresion

```
>>> isinstance(db, Symbol)
True
```

But `db` has one additional field, `db.data` which points to a SQLAlchemy Table.

```
>>> db.data
<sqlalchemy.Table at 0x7f0f64ffbdd0>
```

Compute calls including `db` may omit the customary namespace, e.g.

```
>>> from blaze import compute
>>> expr = db.iris.species.distinct()

>>> # compute(expr, {db: some_sql_object})  # Usually provide a namespace
>>> compute(expr)
['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
```

This implicit namespace can be found with the `._resources` method

```
>>> expr._resources()
{db: <sqlalchemy.Table object>}
```

Additionally, we override the `__repr__` and `_repr_html_` methods to include calls to `compute`. This way, whenever an expression is printed to the screen a small computation is done to print the computed data instead.

As an example, this `__repr__` function looks something like the following:

```
from odo import odo
from pandas import DataFrame
from blaze import Expr

def __repr__(expr):
    expr = expr.head(10)          # Only need enough to print to the screen
    result = compute(expr)        # Do the work necessary to get a result
    df = odo(result, DataFrame) # Shove into a DataFrame
    return repr(df)               # Use pandas' nice printing

Expr.__repr__ = __repr__    # Override normal __repr__ method
```

This provides smooth interactive feel of interactive expressions. Work is only done when an expression is printed to the screen and excessive results are avoided by wrapping all computations in a `.head(10)`.

2.1.24 Developer Workflow

This page describes how to install and improve the development version of Blaze.

If this documentation isn't sufficiently clear or if you have other questions then please email blaze-dev@continuum.io.

Installing Development Blaze

Blaze depends on many other projects, both projects that develop alongside blaze (like `odo`) as well a number of community projects (like `pandas`).

Blaze development happens in the following projects, all of which are available on github.com/blaze/project-name

- Blaze
- DataShape
- Odo
- Dask

Bleeding edge binaries are kept up-to-date on the `blaze` conda channel. New developers likely only need to interact with one or two of these libraries so we recommend downloading everything by the conda channel and then only cloning those git repositories that you actively need:

```
conda install -c blaze blaze  # install everything from dev channel
git clone git://github.com/blaze/blaze.git  # only clone blaze and odo
git clone git://github.com/blaze/odo.git  # only clone blaze and odo
```

GitHub Flow

Source code and issue management are hosted in this github page, and usage of git roughly follows GitHub Flow. What this means is that the *master* branch is generally expected to be stable, with tests passing on all platforms, and features are developed in descriptively named feature branches and merged via github's Pull Requests.

Coding Standards

Unified Python 2 and 3 Codebase:

Blaze source code simultaneously supports both Python 2 and Python 3 with a single codebase.

To support this, all .py files must begin with a few *__future__* imports, as follows.:

```
from __future__ import absolute_import, division, print_function
```

Testing:

In order to keep the `master` branch functioning with passing tests, there are two automated testing mechanisms being used. First is Travis CI, which is configured to automatically build any pull requests that are made. This provides a smoke test against both Python 2 and Python 3 before a merge.

The Travis tests only run on Linux, but Blaze is supported on Linux, OS X, and Windows. Further tests and bleeding-edge builds are carried out using *Anaconda build* which tests and builds Blaze on the following platforms/versions

- Python versions 2.6, 2.7, 3.3, 3.4
- Operating systems Windows, OS-X, Linux
- 32-bit and 64-bit

Relative Imports:

To avoid the side effects of top level imports, e.g. *import blaze*, all internal code should be imported relatively. Thus:

```
#file: blaze/objects/table.py
from blaze import Array
```

should be:

```
#file: blaze/objects/table.py
from .array import Array
```

For cross submodule imports, import from the module api. For example:

```
#file: blaze/objects/table.py
from ..io import printing
```

Relation with Continuum

Blaze is developed in part by Continuum Analytics, a for profit company. Continuum's efforts on Blaze are open source and freely available to the public. The open nature of Blaze is protected by a BSD license.

2.1.25 Expressions and Computation

This is a developer level document. It conveys some of the design decisions around the use of expressions and their lowering to computational backends. It is intended for new developers. It is not necessary to understand this document in order to use Blaze.

Expressions

Blaze represents expressions as Python objects. Classes include

- **Symbol**: leaf expression, `t`
- **Projection**: subset of columns, `t[['name', 'amount']]`
- **Selection**: subset of rows `t[t.amount < 0]`

- **Field**: single column of data or field of record dataset `t.name`
- **Broadcast**: a scalar expression broadcast to a collection, `t.amount + 1`
- **Join**: join two expressions on shared fields , `join(t, s, 'id')`
- **Reduction**: perform a sum or min or max on a collection, `t.amount.sum()`
- **By**: split-apply-combine operation, `by(t.name, total=t.amount.sum())`
- **Also**: `Sort, Distinct, Head, Label, Map, Merge, ...`

In each case an operation (like `Selection`) is a Python class. Each expression defines a fixed set of fields in the `_arguments` attribute

```
class Selection(Expr):
    _arguments = '_child', 'predicate'

class Field(ElemWise):
    _arguments = '_child', 'fieldname'
```

To create a node in the tree explicitly we create a Python object of this class

```
>>> from blaze.expr import *
>>> t = symbol('t', 'var * {id: int, name: string, amount: int}')
>>> amounts = Field(t, 'amount')
```

This object contains its information in a .args attribute

```
>>> amounts._args == (t, 'amount')
True
```

And the set of input expressions in a `._inputs` attribute

```
>>> amounts._inputs == (t,)
True
```

By traversing `._args` one can traverse the tree of all identifying information (including annotating strings and values like `'amount'`) or by traversing `._inputs` one can inspect the much sparser tree of just the major expressions, skipping parameters like the particular field name to be selected.

Most terms have only a single child input. And so often the `._inputs` tree is just a single line of nodes. Notable exceptions include operations like `Join` and `BinOp` which contain two inputs.

Expression Invariants

Blaze expressions adhere to the following properties:

1. They and all of their stored fields are immutable
2. Their string representations evaluate to themselves. E.g. `eval(str(expr)) == expr`
3. They have simple `__init__` constructors that only copy in fields to the object. For intelligent argument handling they have functions. E.g. the `Join` class has an analagous `join` function that should be used by users. Same with the internal `By` class as the user-level `by` function.
4. They can compute their datashape `.dshape` given the datashape of their children and their arguments.

Organization

All expr code occurs in `blaze/expr/`. This directory should be self-contained and not dependent on other parts of Blaze like `compute` or `api`.

- `blaze/expr/core.py` contains code related to abstract tree traversal
- `blaze/expr/expr.py` contains code related to datashape imbued expressions
- `blaze/expr/collections.py` contains operations related to expressions with datashapes that contain a dimension. Operations like `Selection` and `Join` live here
- `blaze/expr/datetime.py`, `blaze/expr/string.py`, ... all contain specialized operations for particular domains.

Computation

Once we have a Blaze expression like the following:

```
>>> deadbeats = t[t.amount < 0].name
```

and some data like the following:

```
>>> data = [[1, 'Alice', 100],
...         [2, 'Bob', -200],
...         [3, 'Charlie', 300]]
```

and a mapping of Symbols to data like the following:

```
>>> namespace = {t: data}
```

then we need to evaluate the intent of the expression on the data. We do this in a step-by-step system outlined by various `compute` functions. The user experience is as follows

```
>>> from blaze import compute
>>> list(compute(deadbeats, namespace))
['Bob']
```

But internally `compute` traverses our expression from the leaves (like `t`) on up, transforming `data` as it goes. At each step it looks at a node in the Blaze expression graph like the following:

```
>>> selection_t = t[t.amount < 0]
```

and transforms the data appropriately, like the following:

```
>>> predicate = lambda amt: amt < 0
>>> data = filter(predicate, data)
```

This step-by-step approach is easy to define through dispatched `compute_up` functions. We create a small recipe for how to compute each expression type (e.g. `Projection`, `Selection`, `By`) against each data type (e.g., `list`, `DataFrame`, `sqlalchemy.Table`, ...) Here is the recipe mapping a `Selection` to a `DataFrame`:

```
>>> @dispatch(Selection, DataFrame)
... def compute_up(t, df, **kwargs):
...     predicate = compute(t.predicate, df)
...     return df[predicate]
```

This approach is modular and allows interpretation systems to be built up as a collection of small pieces. One can begin the construction of a new backend by showing Blaze how to perform each individual operation on a new data type. For example here is a start of a backend for PyTables:

```
>>> @dispatch(Selection, tb.Table)
... def compute_up(expr, data):
...     s = eval_str(expr.predicate)  # Produce string like 'amount < 0'
...     return data.read_where(s)     # Use PyTables read_where method

>>> @dispatch(Head, tb.Table)
... def compute_up(expr, data):
...     return data[:expr.n]          # PyTables supports standard indexing

>>> @dispatch(Field, tb.Table)
... def compute_up(expr, data):
...     return data.col(expr._name)  # Use the PyTables .col method
```

These small functions are isolated enough from Blaze to be easy for new developers to write, even without deep knowledge of Blaze internals.

Compute Traversal

The `compute_up` functions expect to be given:

1. The expression containing information about the computation to be performed
2. The data elements corresponding to the `.inputs` of that expression

The `compute` function orchestrates `compute_up` functions and performs the actual traversal, accruing intermediate results from the use of `compute_up`. By default `compute` performs a `bottom_up` traversal. First it evaluates the leaves of the computation by swapping out keys for values in the input dictionary, `{t: data}`. It then calls `compute_up` functions on these leaves to find intermediate nodes in the tree. It repeats this process, walking up the tree, and at each stage translating a Blaze expression into the matching data element given the data elements of the expression's children. It continues this process until it reaches the root node, at which point it can return the result to the user.

Sometimes we want to perform pre-processing or post-processing on the expression or the result. For example when calling `compute` on a `blaze.data.SQL` object we actually want to pre-process this input to extract out the `sqlalchemy.Table` object and call `compute_up` on that. When we're finished and have successfully translated our Blaze expression to a SQLAlchemy expression we want to post-process this result by actually running the query in our SQL database and returning the concrete results.

Adding Expressions

Expressions can be added by creating a new subclass of *`blaze.expr.expressions.Expr`*. When adding a class, one should define all of the arguments the type will accept in the `_arguments` attribute. Blaze expressions are memoized based on these arguments. Memoization happens at object construction time so any custom initialization logic should happen in the `__new__` instead of the `__init__`. Construction should always forward to the superclass's `__new__`. By default, the `__new__` will reflect a signature from the `_arguments` attribute.

To define the shape of our new expression, we should implement the `_dshape` method. This method should use the shapes of the arguments passed in the constructor plus knowledge of this type of transformation to return the datashape of this expression. For example, thinking of `sum`, we would probably want a method like:

```
def _dshape(self):
    # Drop the dimension of the child reducing to a scalar type.
    return self._child.schema.measure
```

Here we see the `.schema` attribute being used. This attribute dispatches to another optional method: `_schema`. This method should return the datashape with the shape stripped off, or just the data type. If this is not defined, it will be implemented in terms of the `_dshape` method. This is often convenient for subclasses where the rules about the `schema` change but the rules for the dimensions are all the same, like *`blaze.expr.reductions.Reduction`*.

The constructor is not public construction point for a blaze expression. After the class is defined a pairing function should be added to construct and type check the new node. For example, if our node is `Concat`, then the functions should be called `concat`. We will want to decorate this function with `odo.utils.copydoc()` to pull the docstring from the class. This function's main job is type checking the operands. Any constructed node should be in a valid state. If the types do not check out, simply raise a `TypeError` with a helpful message to the user.

Now that the new expression class is defined and the types work out, it must be dispatched to in the compute backends. For each backend that can implement this new feature, a corrosponding `compute_up` dispatch should be defined. For example, assuming we just defined `sum`, we would need to implement something like:

```
@dispatch(sum, np.ndarray)
def compute_up(expr, arr, **kwargs):
    ...

@dispatch(sum, pd.Series)
def compute_up(expr, arr, **kwargs):
    ...

@dispatch(sum, (list, tuple))
def compute_up(expr, arr, **kwargs):
    ...

...
```

Each of these function definitions should appear in the `blaze.compute.*` module for the given backend. For example, the `ndarray` definition should go in `blaze.compute.numpy`.

After implementing the various compute up functions, tests should be written for this behavior. Tests should be added to `blaze/expr/tests` for the expression itself, including tests against the construction and the dshape. Tests are also needed for each of the particular backend implementations to assert that the results of performing the computation is correct accross our various backends.

2.1.26 Computation Pipeline

This is a developer level document. It conveys some of the design decisions around the use of expressions and their lowering to computational backends. It is intended for developers. It is not necessary to understand this document in order to use Blaze.

Problem

Given an expression:

```
>>> from blaze import symbol, sum
>>> x = symbol('x', '5 * int')
>>> y = symbol('y', '5 * int')
>>> expr = sum(x ** 2 + y)
>>> expr
sum((x ** 2) + y)
```

And data arranged into a namespace

```
>>> import numpy as np
>>> xdata = np.array([1, 2, 3, 4, 5])
>>> ydata = np.array([10, 20, 30, 40, 50])
>>> ns = {x: xdata, y: ydata}
```

Our goal is to produce the result implied by the expression

```
>>> np.sum(xdata ** 2 + ydata)
205
```

Using many small functions defined for each backend to do small pieces of this computation

```
@dispatch(blaze.expr.sum, numpy.ndarray)
def compute_up(expr, data):
    return numpy.sum(data)
```

Simple Solution

A simple solution to this problem is to walk from the leaves of the expression tree, applying `compute_up` functions to data resources until we reach the top. In cases like the above example this suffices. This is called a *bottom up* traversal.

Complications

Some backends require more sophistication. In principle we may want to do the following:

1. Modify/optimize the expression tree for a given backend. `optimize(expr, data) -> expr`
2. Modify the data resources before we start execution. `pre_compute(expr, data) -> data`
3. Modify the data resources as they change type throughout the computation `pre_compute(expr, data) -> data`
4. Clean up the data result after we complete execution. `post_compute(expr, data) -> data`
5. Process a leaf of the tree in a bottom up fashion as described above. `compute_up(expr, data) -> data`
6. Process large chunks of the tree at once, rather than always start from the bottom. `compute_down(expr, data) -> data`

Each of these steps is critical to one backend or another. We describe each in turn and then give the complete picture of the entire pipeline.

`optimize :: expr, data -> expr`

Optimize takes an expression and some data and changes the expression based on the data type.

For example in columnar stores (like `bcolz.ctable`) we insert projections in the expression to reduce the memory footprint. In numpy-based array backends we insert `Broadcast` operations to perform loop fusion.

This function is applied throughout the tree at the top-most point at which it is applicable. It is not applied at leaves which have little to optimize.

`pre_compute :: expr, data -> data`

Pre-compute is applied to leaf data elements prior to computation (`xdata` and `ydata` in the example above). It might be used for example, to load data into memory.

We apply `pre_compute` at two stages of the pipeline

1. At the beginning of the computation
2. Any time that the data significantly *changes type*

So for example for the dataset:

```
data = {'my_foo': Foo(...)}
```

If we apply the computation:

```
X -> X.my_foo.distinct()
```

Then after the `X -> X.my_foo` computation as the type changes from `dict` to `Foo` we will call `pre_compute` again on the `Foo` object with the remaining expression:

```
data = pre_compute(X.my_foo.distinct(), Foo(...))
```

A real use case is the streaming Python backend which consumes either sequences of tuples or sequences of dicts. `precompute(expr, Sequence)` detects which case we are in and normalizes to sequences of tuples. This pre-computation allows the rest of the Python backend to make useful assumptions.

Another use case is computation on CSV files. If the CSV file is small we'd like to transform it into a pandas DataFrame. If it is large we'd like to transform it into a Python iterator. This logic can be encoded as a `pre_compute` function and so will be triggered whenever a `CSV` object is first found.

`post_compute :: expr, data -> data`

Post-compute finishes a computation. It is handed the data after all computation has been done.

For example, in the case of SQLAlchemy queries the `post_compute` function actually sends the query to the SQL engine and collects results. This occurs only after Blaze finishes translating everything.

`compute_up :: expr, data -> data`

Compute up walks the expression tree bottom up and processes data step by step.

Compute up is the most prolific function in the computation pipeline and encodes most of the logic. A brief example

```
@dispatch(blaze.expr.Add, np.ndarray, np.ndarray)
def compute_up(expr, lhs, rhs):
    return lhs + rhs
```

`compute_down :: expr, data -> data`

In some cases we want to process large chunks of the expression tree at once. Compute-down operates on the tree top-down, being given the root node / full expression first, and proceeding down the tree while it can not find a match.

Compute-down is less common than compute-up. It is most often used when one backend wants to ship an entire expression over to another. This is done, for example, in the SparkSQL backend in which we take the entire expression and execute it against a SQL backend, and then finally apply that computation onto the SchemaRDD.

It is also used extensively in backends that leverage chunking. These backends want to process a large part of the expression tree at once.

Full Pipeline

The full pipeline looks like the following

1. `Pre-compute` all leaves of data
2. `Optimize` the expression
3. Try calling `compute_down` on the entire expression tree
4. Otherwise, traverse up the tree from the leaves, calling `compute_up`. Repeat this until the data significantly changes type (e.g. `list` to `int` after a `sum` operation)
5. Reevaluate `optimize` on the expression and `pre_compute` on all of the data elements.
6. Go to step 3
7. Call `post_compute` on the result

This is outlined in `blaze/compute/core.py` in the functions `compute(Expr, dict)` and `top_then_bottom_then_top_again_etc`.

History

This design is ad-hoc. Each of the stages listed above arose from need, not from principled fore-thought. Undoubtedly this system could be improved. In particular much of the complexity comes from the fact that `compute_up/down` functions may transform our data arbitrarily. This, along with various particular needs from all of the different data types, forces the flip-flopping between top-down and bottom-up traversals. Please note that while this strategy *works well most of the time* pathalogical cases do exist.

2.2 Older Versions

Older versions of these documents can be found here.

Python Module Index

Index