Monary Documentation

Release 0.4.0

David J. C. Beach

August 14, 2016

Contents

CHAPTER 1

Overview

Monary provides a Python interface for fast column queries from MongoDB. It is much faster than PyMongo for large bulk reads from the database into NumPy's ndarrays.

Note that Monary is still in beta. There are no guarantees of API stability; furthermore, dependencies may change in the future.

Monary is written by David J. C. Beach.

CHAPTER 2

Dependencies

Monary depends on the MongoDB C Driver 1.0, which does not come bundled. Please install the MongoDB C driver using the official instructions.

Monary depends on PyMongo 3.0, NumPy, and pkgconfig.

CHAPTER 3

Issues

All issues can be reported by opening up an issue on the Monary BitBucket issues page.

CHAPTER 4

Contributing

Monary is an open-source project and is hosted on BitBucket. To contribute, fork the project and send a pull request.

We encourage contributors!

See the Contributors page for a list of people who have contributed to developing Monary.

CHAPTER 5

Changes

See the Changelog for a full list of changes to Monary.

CHAPTER 6

About This Documentation

This documentation is generated using the Sphinx documentation generator. The source files for the documentation are located in the *doc/* directory of the Monary distribution.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

7.1 Installing / Upgrading

Monary is in the Python Package Index.

7.1.1 Installing from Source

You can install Monary from source, which provides the latest features (but may be unstable). Simply clone the repository and execute the installation command:

```
$ hg clone https://bitbucket.org/djcbeach/monary
$ cd monary
$ python setup.py install
```

Before you install Monary, you need to have the MongoDB C Driver installed.

If you are installing on Windows, there are some hints in windows_install.txt

7.1.2 Installing CMongo

Monary requires the MongoDB C Driver. To install the C driver, you can use a package manager or follow the instructions in the official MongoDB C Driver 1.0 Github. The MongoDB C Driver (libmongoc) uses the Bson library (libbson) which comes bundled.

Note: LIBMONGOC MUST BE COMPILED WITH SSL AND SASL.

Monary uses pkgconfig to find the libmongoc and libbson installations. If pkgconfig cannot find the libraries, it will look in the default locations: C:\Program Files\libmongoc and C:\Program Files\libbson for Windows, and /usr/local for other systems. If you cannot use pkgconfig **and** libmongoc and libbson are not installed in the default directories, you will need to pass the locations to the installation script:

```
$ python setup.py install --default-libmongoc C:\\usr --default-libbson C:\\usr
```

If you are installing via `pip`, and libcmongo and libbson are not installed in the default directories, you must pass `--default-libmongoc` and `--default-libbson` to pip using `--install-option`.

Note: Monary assumes that libmongoc is installed so that the libraries are in **<default-libmongoc>/lib**. It also expects that the headers are located under **<default-libmongoc>/include**. This is also true for libbson.

7.1.3 Installing with pip

You can use pip to install monary in platforms other than Windows:

```
$ pip install monary
```

To get a specific version of monary:

```
$ pip install monary==0.4.0
```

To upgrade using pip:

```
$ pip install --upgrade monary
```

To specify the location of libcmongo and libbson:

```
$ pip install --install-option="--default-libmongoc" --install-option="C:\\usr"
  --install-option="--default-libbson" --install-option="C:\\usr" monary
```

Note: Although Monary provides a Python package in .egg format, pip does not support installing from Python eggs. If you would like to install Monary with a .egg provided on PyPI, use easy_install instead.

7.1.4 Installing with easy_install

To use `easy_install` from setuptools do:

```
$ easy_install monary
```

To upgrade:

```
$ easy_install -U monary
```

7.1.5 Installing with Wheels

Monary provides Python wheels for Windows and OSX. If you install from wheels, **libmongoc must be installed in a default location**. Download the wheel and install with:

```
$ pip install wheel
$ pip install monary-wheel.whl
```

7.1.6 Installing on Other Unix Distributions

Monary uses the MongoDB C driver. If you install Monary on Linux, BSD and Solaris, you'll need to be able to compile the C driver with the GNU C compiler.

7.2 Tutorial

7.2.1 Prerequisites

To start, you need to have **Monary** installed. In a Python shell, the following should run without raising an exception:

```
>>> import monary
```

You'll also need a MongoDB instance running on the default host and port (`localhost:27017`). If you have downloaded and installed MongoDB, you can start the mongo daemon in your system shell:

```
$ mongod
```

7.2.2 Making a Connection with Monary

To use **Monary**, we need to create a connection to a running **mongod** instance. We can make a new Monary object:

```
>>> from monary import Monary
>>> client = Monary()
```

This connects to the default host and port; it can also be specified explicitly:

```
>>> client = Monary("localhost", 27017)
```

Monary can also accept MongoDB URI strings:

```
>>> client = Monary("mongodb://me@password:test.example.net:2500/database?replicaSet=test&connectTime
```

See also:

The MongoDB connection string format for more information about how these URI's are formatted.

7.2.3 Performing Finds

To perform a "find" query, we first need a data set. We can insert some sample data using the mongo shell:

```
$ mongo
```

Then, at the shell prompt, insert some sample documents into the collection named "coll":

```
> use test
switched to db test
> for (var i = 0; i < 5000; i++) {
... db.coll.insert({ a : Math.random(), b : NumberInt(i) })
... }
```

To check that you've successfully inserted documents, you can run:

```
> db.coll.find()
```

Which will print out the first batch of documents. Each document looks something like this:

```
{
    a : 0.34534613435643535,
    b : 1
}
```

To query the database using Monary, you need to specify the name and type of a MongoDB document field.

For example, to retrieve all of the `b` values with Monary, we use the `query()` function and specify the field name we want and its type:

```
>>> with Monary() as m:
...     arrays = m.query("test", "coll", {}, ["b"], ["int32"])
```

`arrays` is now a list containing a NumPy masked array with 5000 values:

```
>>> arrays
[masked_array(data = [0 1 2 ..., 4997 4998 4999],
              mask = [False False False ..., False False False],
       fill_value = 999999)
]
```

We can also query for both the `a` and `b` fields together:

```
>>> with Monary("localhost") as m:
...     arrays = m.query("test", "coll", {}, ["a", "b"], ["float64", "int32"])
...
>>> arrays
[masked_array(data = [0.7288538725115359 0.4277338122483343 0.5252409593667835 ...,
 0.36620052182115614 0.2733050910755992 0.16910275584086776],
              mask = [False False False ..., False False False],
       fill_value = 1e+20)
, masked_array(data = [0 1 2 ..., 4997 4998 4999],
              mask = [False False False ..., False False False],
       fill_value = 999999)
]
```

`arrays` is now an array containing two masked arrays: one for `a` and one for `b`. The indicies of the two masked arrays correspond to the same document: for example, `arrays[0][250]` and `arrays[1][250]` correspond to the values of `a` and `b` in the 250th document:

```
>>> with Monary("localhost") as m:
...     arrays = m.query("test", "coll", {}, ["a", "b"], ["float64", "int32"])
...
>>> a = arrays[0][250]
>>> b = arrays[1][250]
>>> print a, b
0.653997767251 250
```

If we return to the mongo shell to check that our document matches, we can run:

```
> use test
> db.coll.find({"b":250})
{ "_id" : ObjectId("553e815e5d1bdb50241c0e41"), "a" : 0.6539977672509849, "b" : 250 }
```

7.3 Examples

This section gives examples of accomplishing specific tasks with Monary.

Unless otherwise noted, all examples assume that **mongod** is running on the default host and port. If MongoDB is already downloaded and installed on the system, it can be started by issuing the following in a shell:

```
$ mongod
```

7.3.1 Connection Example

To use Monary, we need to create a connection to a running `mongod`. instance. To connect to a MongoDB server, we simply make a new Monary object. The default host and port are `"localhost"` and `27017` respectively. This will connect to the default host and port:

```
>>> from monary import Monary
>>> client = Monary()
```

However, host and port can be specified explicitly:

```
>>> client = Monary("example.database.com", 8123)
```

More options are available for connection:

```
>>> client = Monary("example.database.com", 8123,
...                 username="sampleUser",
...                 password="1234monary5678",
...                 database="sidedishes",
...                 options={"replicaSet": "test",
...                          "connectTimeoutMS": 12345678})
```

If you want to connect to a MongoDB instance with SSL, see `SSL`.

Alternatively, you can make a connection by specifying a MongoDB URI strings:

```
>>> client = Monary("mongodb://me@password:test.example.net:2500/database?replicaSet=test&connectTime
```

See also:

The MongoDB connection string format for more information about how these URI's are formatted.

7.3.2 Query Example

This example shows you how to use Monary's `query` method.

Setup

For this example, let's use Monary to insert documents with numerical data into MongoDB. First, we can set up a connection to the local MongoDB database:

```
>>> from monary import Monary
>>> client = Monary()
```

Next, we generate some documents. These documents will represent financial assets:

```
>>> import numpy as np
>>> from numpy import ma
>>> records = 10000
>>> unmasked = np.zeros(records, dtype="bool")

>>> # All of our assets have been sold.
```

```
>>> sold = ma.masked_array(np.ones(records, dtype="bool"), unmasked)

>>> # The price at which the assets were purchased.
>>> buy_price = ma.masked_array(np.random.uniform(50, 300, records),
...                             np.copy(unmasked))

>>> delta = np.random.uniform(-10, 30, records)
>>> # The price at which the assets were sold.
>>> sell_price = ma.masked_array(buy_price.data + delta, np.copy(unmasked))
```

Finally, we use Monary to insert the data into MongoDB:

```
>>> from monary import MonaryParam
>>> sold, buy_price, sell_price = MonaryParam.from_lists(
...     [sold, buy_price, sell_price],
...     ["sold", "price.bought", "price.sold"])

>>> client.insert(
...     "finance", "assets", [sold, buy_price, sell_price])
```

See also:

The MonaryParam Example and The Monary Insert Example

Using Query

Now we query the database, specifying the keys we want from the MongoDB documents and what type we want the returned data to be:

```
>>> buy_price, sell_price = client.query(
...     "finance", "assets", {"sold": True},
...     ["price.bought", "price.sold"],
...     ["float64", "float64"])
>>> assets_count = sell_price.count()
>>> gain = sell_price - buy_price   # vector subtraction
>>> cumulative_gain = gain.sum()
```

Finally, we can review our financial data:

```
>>> cumulative_gain
100254.10514435501
>>> assets_count
10000
```

7.3.3 Block Query

This example demonstrates the use of Monary's `block_query` command.

`block_query` functions similarly to `query`. The main difference is that `block_query` returns a generator. Furthermore, all but the last NumPy masked arrays that block_query returns will be overwritten as you iterate through the results. This allows you to process unlimited or unknown amounts of data with a fixed amount of memory.

Setup

This setup will be identical to the setup in the query example.

For this example, let's use Monary to insert documents with numerical data into MongoDB. First, we can set up a connection to the local MongoDB database:

```
>>> from monary import Monary
>>> client = Monary()
```

Next, we generate some documents. These documents will represent financial assets:

```
>>> import numpy as np
>>> from numpy import ma
>>> records = 10000
>>> unmasked = np.zeros(records, dtype="bool")

>>> # All of our assets have been sold.
>>> sold = ma.masked_array(np.ones(records, dtype="bool"), unmasked)

>>> # The price at which the assets were purchased.
>>> buy_price = ma.masked_array(np.random.uniform(50, 300, records),
...                             np.copy(unmasked))

>>> delta = np.random.uniform(-10, 30, records)
>>> # The price at which the assets were sold.
>>> sell_price = ma.masked_array(buy_price.data + delta, np.copy(unmasked))
```

Finally, we use Monary to insert the data into MongoDB:

```
>>> from monary import MonaryParam
>>> sold, buy_price, sell_price = MonaryParam.from_lists(
...     [sold, buy_price, sell_price],
...     ["sold", "price.bought", "price.sold"])

>>> client.insert(
...     "finance", "assets", [sold, buy_price, sell_price])
```

See also:

The MonaryParam Example and The Monary Insert Example

Using Block Query

Now we query the database, specifying also how many results we want per block:

```
>>> cumulative_gain = 0.0
>>> assets_count = 0
>>> for buy_price_block, sell_price_block in (
...     client.block_query("finance", "assets", {"sold": True},
...                        ["price.bought", "price.sold"],
...                        ["float64", "float64"],
...                        block_size=1024)):
...     assets_count += sell_price_block.count()
...     gain = sell_price_block - buy_price_block   # vector subtraction
...     cumulative_gain += gain.sum()
```

Finally, we can review our financial data:

```
>>> cumulative_gain
100254.10514435501
>>> assets_count
10000
```

7.3.4 Aggregation Pipeline

This example demonstrates how to use MongoDB's aggregation pipeline with Monary. It assumes that you know the basics of aggregation pipelines. For a complete list of the possible pipeline operators, refer to the aggregation framework operators.

Note that you must have MongoDB 2.2 or later to use the aggregation pipeline.

Setup

This example assumes that **mongod** is running on the default host and port. You can use any test data you want to aggregate on; this example will use the MongoDB zipcode data set. Simply use **mongoimport** to import the collection into MongoDB.

```
$ wget http://media.mongodb.org/zips.json
$ mongoimport --db zips --collection data zips.json
```

The data set will be loaded to the database `zips` under the collection `data`:

```
> use zips
switched to db zips
> db.data.find().limit(1).pretty()
{
    "_id" : "01001",
    "city" : "AGAWAM",
    "loc" : [
        -72.622739,
        42.070206
    ],
    "pop" : 15338,
    "state" : "MA"
}
```

Performing Aggregations

In Monary, a pipeline must be a list of Python dicts. Each dict must contain exactly one aggregation pipeline operator, each representing one stage of the pipeline.

For convenience, you may also pass a dict containing a single aggregation operation if your pipeline contains only one stage.

This example will show all of the states in the dataset and their populations:

```
>>> from monary import Monary
>>> m = Monary()
>>> pipeline = [{"$group" : {"_id" : "$state",
...                          "totPop" : {"$sum" : "$pop"}}}]
>>> states, population = m.aggregate("zips", "data",
...                                  pipeline,
...                                  ["_id", "totPop"],
...                                  ["string:2", "int64"])
>>> strs = list(map(lambda x: x.decode("utf-8"), states))
>>> list("%s: %d" % (state, pop)
         for (state, pop) in zip(strs, population))
['WA: 4866692',
 'HI: 1108229',
 'CA: 29754890',
```

```
 'OR: 2842321',
 'NM: 1515069',
 'UT: 1722850',
 'OK: 3145585',
 'LA: 4217595',
 'NE: 1578139',
 'TX: 16984601',
 'MO: 5110648',
 'MT: 798948',
 'ND: 638272',
 'AK: 544698',
 'SD: 695397',
 'DC: 606900',
 'MN: 4372982',
 'ID: 1006749',
 'KY: 3675484',
 'WI: 4891769',
 ...]
```

7.3.5 MonaryParam Example

A `MonaryParam` represents a single column, i.e. a single field, in a set of BSON documents. It contains three pieces of data: the name of the field it represents, the type of the data stored in that field, and the values of the field itself. For example, say you had a set of 12 documents that all contained the field "count" with the values 1-12:

```
>>> import numpy as np
>>> count_field = "count"
>>> count_type = "int64"
>>> count_values = np.ma.masked_array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

If you wanted to make a MonaryParam that represented the field `count`, you could:

```
>>> from monary import MonaryParam
>>> mp = MonaryParam(count_values, count_field, count_type)
```

Or, because some types can be determined by the type of the NumPy masked array, you could simply call:

```
>>> p = MonaryParam(count_values, count_field)
```

See also:

The Type section in the Reference

If you wanted to represent a few different fields, you can create a set of MonaryParams using lists. Say you have another field, `month`, in your set of 12 BSON documents:

```
>>> month_field = "month"
>>> month_type = "string:9"
>>> month_values = np.ma.masked_array(["january", "february", "march", "april", "may",
...                                    "june", "july", "august", "september", "october",
...                                    "november", "december"])
```

You can create multiple MonaryParams using `from_lists`:

```
>>> fields = [count_field, month_field]
>>> types = [count_type, month_type]
>>> values = [count_values, month_values]
>>> params = MonaryParam.from_lists(values, fields, types)
```

7.3.6 Insert Example

This example shows you how to use Monary's `insert` method to send documents to MongoDB.

Any value that can be queried can also be inserted. Both nested field insertion (via fields containing ".") and BSON value insertion are supported as well.

Purpose of Insert

Inserts allow you to use Monary to convert data from NumPy masked arrays into documents stored in MongoDB. Monary's insert takes in a list of `MonaryParams`.

Monary inserts can also be used to store intermediate data. This can be useful when doing operations on blocks of data with block query.

Setup

For this example, let's insert some unprocessed documents representing students' test scores into MongoDB. Please see the MonaryParam example to understand how to create a `MonaryParam`.

First we need to connect to our local DB:

```
>>> import monary
>>> client = monary.Monary()
```

Next, we generate the documents. Note that we are using `bson.encode` to store our subdocument:

```
>>> import bson
>>> import random
>>> al_num = '0123456789abcdefghijklmnopqrstuvwxyz'
>>> scores = []
>>> ids = []
>>> names = []
>>> for _ in range(1000):
...     ids.append("".join(al_num[random.randint(0, len(al_num)-1)]
...                        for _ in range(14)))
...     score = {"midterm": random.randint(0, 1000) / 10,
...              "final": random.randint(0, 1000) / 10}
...     scores.append(bson.BSON.encode(score))
...     names.append("...")
```

Now that we have generated documents, we need to construct a `MonaryParam`. `MonaryParams` represent one column, i.e. one field, for a set of BSON documents. We need the data itself to be in numpy's masked_array type:

```
>>> import numpy as np
>>> max_length = max(map(len, scores))
>>> scores_ma = np.ma.masked_array(scores, np.zeros(1000), "<V%d"%max_length)
>>> ids_ma = np.ma.masked_array(ids, np.zeros(1000), "S14")
>>> names_ma = np.ma.masked_array(names, np.zeros(1000), "S3")
```

Now we can create a `MonaryParam`:

```
>>> types = ["bson:%d"%max, "string:14", "string:3"]
>>> fields = ["scores", "student_id", "student_name"]
>>> values = [scores_ma, ids_ma, names_ma]
>>> params = monary.MonaryParam.from_lists(values, fields, types)
```

And we can insert it into the database "monary_students", and the collection "raw":

```
>>> client.insert("monary_students", "raw", params)
```

Using Monary Insert

The semester has ended, and it's time to assign grades to each student. Let's first get all the raw test data back into NumPy arrays with Monary:

```
>>> import numpy as np
>>> from monary import Monary
>>> m = Monary()
>>> ids, midterm, final = \
...     m.query("monary_students", "raw", {},
...             ["student_id",
...              "test_scores.midterm",
...              "test_scores.midterm"],
...             ["string:14", "float64",
...              "float64"])
```

Now we process the scores and assign grades to each student:

```
>>> grades = [None, None]
>>> for i, arr in enumerate([midterm, final]):
...     # curve to average of 2.3333
...     mean, stdev = arr.mean(), arr.std()
...     grades[i] = (arr - mean) / stdev
...     grades[i] += 2.3333
...     # bound grades within [0.0, 4.0]
...     fours = np.argwhere(grades[i] > 4.0)
...     zeros = np.argwhere(grades[i] < 0.0)
...     grades[i][fours] = 4.0
...     grades[i][zeros] = 0.0
```

Now weight both tests and assign overall grades:

```
>>> overall_grades = (grades[0] * 0.4 + grades[1] * 0.6).round(3)
```

Then we need to create `MonaryParams`:

```
>>> from monary import MonaryParam
>>> id_mp = MonaryParam(ids, "student_id", "string:14")
>>> overall_mp = MonaryParam(overall_grades, "grades.overall")
>>> midterm_mp = MonaryParam(grades[0], "grades.midterm")
>>> final_mp = MonaryParam(grades[1], "grades.final_exam")
```

Finally, we insert the results to the database:

```
>>> ids = m.insert("monary_students", "graded",
...                [id_mp, overall_mp, midterm_mp, final_mp])
>>> from monary import mvoid_to_bson_id
>>> oids = list(map(mvoid_to_bson_id, ids))
>>> oids[0]
ObjectId('53dba51e61155374af671dc1')
```

We can see that insert returns a Numpy array containing the ObjectId of the inserted documents.

7.3.7 Timestamp Example

This example demonstrates how to extract timestamps with Monary.

Setup

We can use Monary to populate a collection with some test data containing random timestamps. First, make a connection:

```
>>> from monary import Monary
>>> client = Monary()
```

Then we can generate random timestamps:

```
>>> import random
>>> import bson
>>> timestamps = []
>>> for _ in range(10000):
...     time = random.randint(0, 1000000)
...     inc = random.randint(0, 1000000)
...     ts = bson.timestamp.Timestamp(time=time, inc=inc)
...     timestamps.append(ts)
```

Next we put these values into a numpy masked array:

```
>>> import numpy as np
>>> from numpy import ma
>>> timestamps = [(ts.time << 32) + ts.inc for ts in timestamps]
>>> ts_array = ma.masked_array(np.array(timestamps, dtype="uint64"),
...                            np.zeros(len(timestamps), dtype="bool"))
```

Finally we use monary to insert this data into MongoDB:

```
>>> from monary import MonaryParam
>>> client.insert(
...     "test", "data", [MonaryParam(ts_array, "ts", "timestamp")])
```

See also:

The MonaryParam Example and The Monary Insert Example

Finding Timestamp Data

Next we use query to get back our data:

```
>>> timestamps, = client.query("test", "data", {},
...                            ["ts"], ["timestamp"])
```

Finally, we use struct to unpack the resulting data:

```
>>> import struct
>>> data = [struct.unpack("<ii", ts) for ts in timestamps]
>>> timestamps = [bson.timestamp.Timestamp(time=time, inc=inc)
...               for time, inc in data]
>>> timestamps[0]
Timestamp(870767, 595669)
```

7.3.8 Strings Example

This example demonstrates how to extract strings with Monary.

Note: Due to the large difference between Python 2 strings and Python 3 strings, some of the examples will have two different versions. Please pay attention to the comments to decide which lines would work in your version of Python.

Setup

We can use PyMongo to populate a collection with some test data containing random strings. First, make a connection:

```
>>> from monary import Monary
>>> client = Monary()
```

Then, we can create a NumPy masked array of random strings:

```
>>> import random
>>> import string
>>> from numpy import ma
>>> import numpy as np
>>> lowercase = string.ascii_lowercase
>>> def rand_str(length):
...     return "".join(random.choice(lowercase)
...                    for c in range(0, length))
...
>>> strs = ma.masked_array(np.zeros(1000, dtype="S10"),
...                        np.zeros(1000, dtype="bool"))
>>> for i in range(0, 1000):
...     strs[i] = rand_str(random.choice(range(1, 10))).encode("utf-8")
```

Finally we can use Monary to insert these strings:

```
>>> from monary import MonaryParam
>>> client.insert(
...     "test", "data", [MonaryParam(strs, "stringdata", "string:10")])
```

See also:

The MonaryParam Example and The Monary Insert Example

MongoDB encodes strings in UTF-8, so you can use non-ASCII characters. Here is a sample script that inserts non-ASCII strings:

```
# -*- coding: utf-8 -*-
# the above comment is needed for Python 2 files with non-ASCII characters
from monary import Monary, MonaryParam
import numpy as np
from numpy import ma

client = Monary()
strs = ["liberté", "", "ελευθερα"]

# If running in python 2, encode in utf-8
strs = list(map(lambda x: x.encode("utf-8"), strs))

max_len = max(map(len, strs))
str_array = ma.masked_array(strs, [False] * 3, dtype=("S%d" % max_len))
```

```
str_mp = MonaryParam(str_array, "stringdata", "string:%d" % max_len)
client.insert("test", "utf8", [str_mp])
```

Finding String Data

String Sizing

Monary can obtain strings from MongoDB if you specify the size of the strings in bytes.

Note: Unicode characters encoded in UTF-8 can be larger than one byte in size. For example, the size of "99¢" is 4 - one byte for each '9' and two bytes for '¢'.

It is safe to specify larger sizes; if the length of a string is smaller than the specified size, the extra space is padded with `NUL` characters.

If the length of a string is greater than the specified size, **it will be truncated**. For example, storing "hello" in a block of size 3 results in "hel". Multi-byte UTF-8-encoded characters may also be truncated, which may result in an invalid string.

To determine the length of a Unicode (or ASCII) string in Python, encode the desired string object into UTF-8 and take its length:

```
>>> def strlen_in_bytes(string):
...     return len(string.encode("utf-8"))
```

Monary provides a way to query the database directly for the byte sizes of strings.

Performing Queries

If we don't know the maximum size of the strings in advance, we can query for their size, which returns the size of the strings in bytes:

```
>>> from monary import Monary
>>> client = Monary()
>>> sizes, = client.query("test", "data", {}, ["stringdata"], ["size"])
>>> sizes
masked_array(data = [9L 7L 3L ...,, 6L 5L 9L],
             mask = [False False False ... False False False],
       fill_value = 999999)
```

Now that we have the sizes of all the strings, we can find the maximum string size:

```
>>> max_size = sizes.max()
```

Finally, we can use this size to obtain the actual strings from MongoDB:

```
>>> data, = client.query("test", "data", {}, ["stringdata"],
...                      ["string:%d" % max_size])
>>> data
masked_array(data = ['nbuvggamk' 'bkhwkwl' 'tvb' ..., 'rsdefd' 'lpasx' 'wpdlxierd'],
             mask = [False False False ..., False False False],
       fill_value = N/A)
```

Each of these values is a `numpy.string_` instance. You can convert it to a regular Python string if you'd like:

```
>>> mystr = str(data[0])   # Python 2
>>> mystr = data[0].decode("utf-8")   # Python 3
```

If you have non-ASCII UTF-8 characters in this data, you can create a Unicode (Python 2) or Str (Python 3) object by decoding the data:

```
>>> sizes, = client.query("test", "utf8", {}, ["stringdata"], ["size"])
>>> data, = client.query("test", "utf8", {}, ["stringdata"],
...                      ["string:%d" % sizes.max()])

>>> # Python 2:
>>> mystr = unicode(data[0], "utf-8")
>>> mystr
u'libert\xe9'
>>> print mystr
liberté

>>> # Python 3:
>>> mystr = data[0].decode("utf-8")   # Python 3
>>> mystr
'liberté'
>>> print(mystr)
liberté
```

7.3.9 Binary Data Example

This example shows you how to obtain blocks of binary data from MongoDB with Monary.

Setup

We are going to use 100 random files for this example. If you don't happen to have random files lying around, you can issue this command at a Unix shell:

```
$ for ((i = 0; i < 100; i=i+1)); do
> head -c $SIZE < /dev/urandom > "img${i}.jpg"
> done
```

This creates 100 files, each containing `$SIZE` bytes of random data.

For this example, let's use Monary to insert raw binary image data into MongoDB. First, we can set up a connection to the local MongoDB database:

```
>>> from monary import Monary
>>> client = Monary()
```

Next, we open some random image files. Assume we have image files named `img0.jpg` through `img99.jpg`:

```
>>> import numpy as np
>>> from numpy import ma
>>> images = []
>>> sizes = ma.masked_array(np.zeros(100, dtype="uint32"),
...                         np.zeros(100, dtype="bool"))
>>> for i in range(0, 100):
...     with open("img%d.jpg" % i, "rb") as img:
...         f = img.read()
...         images.append(f)
...         sizes[i] = len(f)
```

Next we convert the image list into a numpy masked array:

```
>>> max_size = sizes.max()
>>> img_type = "<V%d" % max_size
>>> img_array = ma.masked_array(np.array(images, dtype=img_type),
...                             np.zeros(100, dtype="bool"))
```

Finally, we use Monary's `binary` type to insert the data into MongoDB:

```
>>> from monary import MonaryParam
>>> img_mp = MonaryParam(img_array, "img", "binary:%d" % max_size)
>>> size_mp = MonaryParam(sizes, "size")
>>> client.insert("test", "data", [img_mp, size_mp])
```

See also:

The MonaryParam Example and The Monary Insert Example

Note: We also store the original file size. This is because `img_array` has a fixed width, so all of the images, once put into MongoDB, will be of size `max_size`. When we retrieve this data, it will be useful to know the original file size so we can truncate the binary data appropriately.

Finding Binary Data

To query binary data, Monary requires the size of the binary to load in. Since the data in different documents can be of different sizes, we need to use the size of the biggest binary blob to avoid truncation.

To find the size of the data in bytes, we use the `size` type:

```
>>> sizes, = client.query("test", "data", {}, ["img"], ["size"])
>>> sizes
masked_array(data = [255L 255L 255L ..., 255L 255L 255L],
             mask = [False False False ..., False False False],
       fill_value = 999999)
```

Note that these sizes are unsigned 32-bit integers:

```
>>> sizes[0]
255
>>> type(sizes[0])
<type 'numpy.uint32'>
```

We can get the maximum image size by calling `max`:

```
>>> max_size = sizes.max()
```

Finally, we can issue a query command to get pointers to the binary data:

```
>>> data, sizes = client.query("test", "data", {},
...                            ["img", "size"],
...                            ["binary:%d" % max_size, "uint32"])
>>> data
masked_array(data = [<read-write buffer ptr 0x7f8a58421b50, size 255 at 0x105b6deb0> ...],
             mask = [False ...]
       fill_value = ???)
```

Each buffer pointer is of type `numpy.ma.core.mvoid`. From here, you can use NumPy to manipulate the data or export it to another location.

7.3.10 SSL Example

Running `mongod` with SSL

In order to run monary with SSL, mongod needs to be compiled with ssl enabled. Instructions can be found here.

To run `mongod` with SSL options:

```
$ mongod --sslOnNormalPorts --sslPEMKeyFile <pem> --sslCAFile=<ca>
         --sslWeakCertificateValidation
```

The test permissions files can be found in the "test/certificates" directory of the Monary source. To run mongod with the SSL permissions provided by the monary source directory:

```
$ mongod --sslOnNormalPorts --sslPEMKeyFile=test/certificates/server.pem
         --sslCAFile=test/certificates/ca.pem --sslWeakCertificateValidation
```

First we're going to put some sample data into our DB. Make sure you are using a `mongo` shell that has been compiled with SSL:

```
$ mongo --ssl
  > for (var i = 0; i < 5; i++) { db.ssl.insert({x1:NumberInt(i) }) }
  WriteResult({ "nInserted" : 1 })
```

The sample certificate files can be downloaded from here. Collect the certificate files to be passed to Monary:

```
>>> import os
>>> import monary
>>> cert_path = os.path.join("test", "certificates")
>>> client_pem = os.path.join(cert_path, 'client.pem')
>>> ca_pem = os.path.join(cert_path, 'ca.pem')
```

Connect with Monary:

```
>>> with monary.Monary("mongodb://localhost:27017/?ssl=true",
...                    pem_file=client_pem,
...                    ca_file=ca_pem,
...                    ca_dir=cert_path,
...                    weak_cert_validation=False) as m:
...     arrays = m.query("test", "ssl", {}, ["x1"], ["float64"])
>>> arrays
[masked_array(data = [1.0 0.0 1.0 2.0 3.0 4.0],
        mask = [False False False False False False],
  fill_value = 1e+20)
]
```

Password Protected PEM Files

To run `mongod` with a password-protected PEM file, you need to run mongo with the following options:

```
$ mongod --sslMode requireSSL --sslPEMKeyFile=<password-protected-pem> --sslPEMKeyPassword "qwerty"
```

For example, with the test/certificates files:

```
$ mongod --sslMode requireSSL --sslPEMKeyFile=test/certificates/password_protected.pem --sslPEMKeyPas
```

To connect with an SSL-protected MongoDB instance with Monary, you just need to specify the SSL parameters:

```
>>> pwd_pem = os.path.join(cert_path, 'password_protected.pem')
>>> with monary.Monary("mongodb://localhost:27017/?ssl=true",
...                    pem_file=pwd_pem,
...                    pem_pwd='qwerty',
...                    ca_file=ca_pem,
...                    weak_cert_validation=True) as monary:
...     arrays = monary.query("test", "ssl", {}, ["x1"], ["float64"])
>>> arrays
[masked_array(data = [0.0 1.0 2.0 3.0 4.0],
        mask = [False False False False False],
  fill_value = 1e+20)
]
```

Alternatively, if you do not want to specify the password directly, you can connect without the `pem_pwd` parameter. You will be prompted for the password.

7.4 Type Reference

Monary converts values between BSON and NumPy. The following data types can be stored in NumPy arrays:

- `id`: ObjectID
- `bool`: boolean
- `int8`: signed two's compliment 8-bit integer
- `int16`: signed two's compliment 16-bit integer
- `int32`: signed two's compliment 32-bit integer
- `int64`: signed two's compliment 64-bit integer
- `uint8`: unsigned 8-bit integer
- `uint16`: unsigned 16-bit integer
- `uint32`: unsigned 32-bit integer
- `uint64`: unsigned 64-bit integer
- `float32`: IEEE 754 single-precision (32-bit) floating point value
- `float64`: IEEE 754 single-precision (64-bit) floating point value
- `date`: UTC datetime
- `timestamp`: Timestamp
- `string`: UTF-8 string
- `binary`: binary data
- `bson`: BSON document
- `type`: <see below>
- `size`: <see below>
- `length`: <see below>

When values are retrieved from MongoDB they are converted from BSON types to the NumPy types you specify. See query, block_query, and aggregate. All types are implemented in C, thus type conversions follow the rules of the C standard.

When values are inserted into MongoDB they are converted from NumPy types to BSON types. In most cases the BSON type can be inferred from the input NumPy type, except for types `id`, `date`, `timestamp`, `string`, `binary`, and `bson`.

See also:

The official BSON Specification for more information about how BSON is stored in binary format.

7.4.1 BSON Types

Integers

BSON only stores signed 32- and 64-bit integers. Specifying an unsigned integer or an integer size of 8- or 16-bits causes a cast. Casting a negative number to an unsigned integer or casting to a smaller integer size with overflow is implementation-defined, depending on the C compiler for your platform.

Floating-point numbers can be cast to integers. In the case of overflow, the result is undefined.

Note that signed integers are kept in two's complement format.

Floating-Point

BSON only stores doubles; that is, 64-bit IEEE 754 floating point numbers. Specifying float32 will cause a cast with possible loss of precision.

Integers can be cast to floating-point. If the original value is outside of the range of the destination type, the result is undefined.

Datetimes

This datetime is a 64-bit integer representing milliseconds since the epoch, which is January 1, 1970. Dates before the epoch are expressed as negative milliseconds. Monary provides helper functions for converting MongoDB dates into Python datetime objects.

Timestamps

A BSON timestamp is a special type used internally by MongoDB. It is a 64-bit integer, where the first four bytes represent an increment and the second four represent a timestamp. For storing arbitrary times, use datetime instead.

See also:

Timestamp Example for an example of using timestamps.

Binary

Binary data retrieved from MongoDB is accessed via

See also:

Binary Data Example for an example of using binary data.

Strings

All strings in MongoDB are encoded in UTF-8. When performing a find query on strings, you must also input the lengths of the strings in bytes. For a regular ASCII string, the length is the number of characters. Characters with higher-order UTF-8 encodings may occupy more space. You can use Monary to query for the strings' actual size in bytes to determine what size to use.

Find queries return lists of `numpy.string_` objects.

See also:

Strings Example for an example of using strings.

Subdocuments

Documents are retrieved as BSON. Each value is a NumPy void pointer to the binary data.

7.4.2 Monary-Specific Types

Type

"Type" refers to a field's BSON type code. For integers, the type code returned will be either an int32 (type code 16) or int64 (type code 18), depending on how it is stored in MongoDB.

Here is a list of selected type codes, as per the specification:

- 1 : double
- 2 : string
- 3 : (sub)document
- 4 : array
- 5 : binary
- 7 : ObjectID
- 8 : boolean
- 9 : UTC datetime
- 16 : 32-bit integer
- 17 : timestamp
- 18 : 64-bit integer

See also:

Why do my integers have a "double" type code?

Size

For UTF-8 strings, JavaScript code, binary values, BSON subdocuments, and arrays, "size" is defined as the size of the object in bytes. All other types do not have a defined Monary size.

Length

For ASCII/UTF-8 strings and Javascript code, "length" refers to the string length (the same as `len` on a string); for arrays, the number of elements; and for documents, the number of key-value pairs. No other types have a defined Monary length.

7.5 Write Concern Reference

The Monary WriteConcern object allows users to specify the type of write Monary will perform. This object will be converted into a C struct and used for `insert`, `remove`, and `update`. The parameters to the constructor mimic the MongoDB Write Concern options.

See also:

The MongoDB manual entry on Write Concern

7.5.1 wtimeout

This option specifies a time limit, in milliseconds, for the write concern. `wtimeout` is only applicable for `w` values greater than 1.

`wtimeout` causes write operations to return with an error after the specified limit, even if the required write concern is not fulfilled. When these write operations return, MongoDB does not undo successful data modifications performed before the write concern exceeded the wtimeout time limit.

If you do not specify the `wtimeout` option and the level of write concern is unachievable, the write operation will block indefinitely. Specifying a `wtimeout` value of 0 is equivalent to a write concern without the `wtimeout` option.

7.5.2 wjournal

The `wjournal` option confirms that the mongod instance has written the data to the on-disk journal. This ensures that data is not lost if the mongod instance shuts down unexpectedly. Set to `True` to enable.

7.5.3 wtag

By specifying a `wtag`, you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.

See also:

The MongoDB tag set configuration tutorial

7.6 Frequently Asked Questions

Contents

7.6.1 Can Monary do Removes, Updates, and/or Upserts?

Though there will soon be support for bulk removes, updates, and upserts from arrays into MongoDB, for now Monary can only retrieve and store data. It cannot perform any updates or removals. In the meantime, you can use PyMongo.

7.6.2 Why does my array contain masked values?

Typically, a value is masked if the data type you specify for a field is incompatible with the actual type retrieved from the document in MongoDB, or if the specified field is absent in some of your documents.

Alternatively, it could be that the documents do not contain the field that you requested.

If the entire array is masked, there are no documents in the collection that contain that field, or all of the matching fields in the database have an incompatible type.

If there are only some masked values in the result array, then some of the documents have fields with the specified name but not of the specified type.

Consider, for example, inserting the following two documents at the mongo shell:

```
> db.foo.insert({ a : NumberInt(1), sequence : 1 });
WriteResult({ "nInserted" : 1 })

> db.foo.insert({ a : "hello", sequence : 2 })
WriteResult({ "nInserted" : 1 })
```

Because there is a type mismatch for the field "a", some values will be masked depending on what type the query asks for:

```
>>> from monary import Monary()
>>> m = Monary()
>>> m.query("test", "foo", {}, ["a"], ["int32"], sort="sequence")
[masked_array(data = [1 --],
              mask = [False  True],
        fill_value = 99999)
]
>>> m.query("test", "foo", {}, ["a"], ["string:5"], sort="sequence")
[masked_array(data = [-- 'hello'],
              mask = [ True False],
        fill_value = N/A)
]
```

7.6.3 How does monary deal with ObjectIds?

When querying for `_id`'s or when inserting documents, you might end up with ObejectIds stored in a NumPy array. The data type of the array is `"<V12"`, and each individual element is of type `numpy.ma.core.mvoid`. However it may end up looking like this:

```
masked_array(data = [ array([ 83, -18,  62, -28,  97,  21,  83, -51, -21, 106, -54,  11], dtype=int8)
 array([ 83, -18,  62, -28,  97,  21,  83, -51, -21, 106, -54,  12], dtype=int8)
 array([ 83, -18,  62, -28,  97,  21,  83, -51, -21, 106, -54,  13], dtype=int8)
 ...,
 array([  83,  -18,   63,  -63,   97,   21,   83,  -51,  -21, -104, -112,
        -56], dtype=int8)
 array([  83,  -18,   63,  -63,   97,   21,   83,  -51,  -21, -104, -112,
        -55], dtype=int8)
 array([  83,  -18,   63,  -63,   97,   21,   83,  -51,  -21, -104, -112,
        -54], dtype=int8)],
             mask = [False False False ..., False False False],
       fill_value = ???)
```

Or it may look like this:

```
[masked_array(data = [<read-write buffer ptr 0x7f93718a3600, size 12 at 0x1094e1df0>
<read-write buffer ptr 0x7f93718a360c, size 12 at 0x1094e1d70>
<read-write buffer ptr 0x7f93718a3618, size 12 at 0x1094e1f70> ...,
<read-write buffer ptr 0x7f93718b4f1c, size 12 at 0x1097b92b0>
<read-write buffer ptr 0x7f93718b4f28, size 12 at 0x1097b92f0>
<read-write buffer ptr 0x7f93718b4f34, size 12 at 0x1097b9330>],
         mask = [False False False ..., False False False],
   fill_value = ???)
]
```

If you would like this as a `bson.objectid.ObjectId`, it can be done like this:

```
>>> from monary import Monary
>>> with Monary() as m:
...     ids = m.query("db", "col", {}, ["_id"], ["id"])

>>> from monary.monary import mvoid_to_bson_id
>>> id_vals = ids[0] # Depends on the type of query.
>>> oids = list(map(mvoid_to_bson_id, id_vals))
>>> oids[0]
ObjectId('53dba51e61155374af671dc1')
```

7.6.4 What if I don't know what type of data I want from MongoDB?

MongoDB has very flexible schemas; a consequence of this is that documents in the same collection can have fields of different types. To determine the type of data for a certain field name, specify the type "type":

```
>>> from monary import Monary
>>> m = Monary()
>>> m.query("test", "foo", {}, ["a"], ["type"])
[masked_array(data = [16 2]
             mask = [False False],
       fill_value = 999999)
]
```

This returns an 8-bit integer containing the BSON type code for the object.

See also:

The BSON specification for the BSON type codes.

7.6.5 How do I retrieve string data using Monary?

Internally, all strings are C strings. To specify a string type, you must also indicate the size of the string (not including the terminating `NUL` character):

```
>>> m.query("test", "foo", {}, ["mystr"], ["string:3"])
[masked_array(data = ['foo' 'bar' 'baz'],
              mask = [False False False],
       fill_value = N/A)
]
```

Ideally, the size specified should be the least upper bound of the sizes of strings you are expecting to receive.

See also:

Strings Example

7.6.6 When should I use a block query?

Block query can be used to read through many documents while only storing a specified amount of documents in memory at a time. This can save memory and decrease initial latency by processing documents in batches. This can also be used in combination with insert to perform operations on all of your data and store the processed results in a new collection.

See also:

Block Query and Insert Example

7.6.7 Why do my integers have a "double" type code?

Though the numbers look like integers, they are being stored internally as doubles. This most commonly happens at the mongo shell:

```
> use test
> db.foo.insert({ a : 22 })
WriteResult({ "nInserted" : 1 })
```

The BSON type code for double is 1, so this results in:

```
>>> m.query("test", "foo", {}, ["a"], ["type"])
[masked_array(data = [1],
              mask = [False],
       fill_value = N/A)
]
```

Because the mongo shell is a JavaScript interpreter, it follows the rules of JavaScript: all numbers are floating-point. If you'd like to insert strictly integers into MongoDB, use `NumberInt`:

```
> use test
> db.foo.insert({ b : NumberInt(1) })
WriteResult({ "nInserted" : 1 })
```

This yields the expected type code:

```
>>> m.query("test", "foo", {}, ["b"], ["type"])
[masked_array(data = [16],
              mask = [False],
       fill_value = N/A)
]
```

See also:

ECMAScript Number Type

7.7 Changelog

7.7.1 Changes in Version 0.4.0

- Remove vendoring of libmongoc - users **must** install libmongoc 1.0 or later independently.
- Improved error handling and error reporting.
- Improved Testing: tests can be run through `setup.py test`; added skiptest; removed dependency on Nose.
- Inserts.
- Connection over SSL.
- Various bugfixes.

7.7.2 Changes in Version 0.3.0

Version 0.3.0 is a major overhaul of the backend code.

- Upgrade to latest version of the MongoDB C driver (0.98.0).
- `monary_connect` now takes a MongoDB URI or hostname and port. See the connection string documentation for more information.
- Monary can now freely cast between integer and floating-point values.
- Debug messages are suppressed by default.
- `datehelper` now allows negative timedeltas and time values to represent dates before the epoch.
- Monary objects no longer support the `authenticate()` method, which is a breaking change. If your code relied on `authenticate()`, you must now include the username and password in the MongoDB URI passed into the Monary constructor. Authentication now occurs when a connection is made.

Issues Resolved

The new connection format fixes a bug where connection failures or invoking the constructor with `monary.Monary("localhost")` would cause a segmentation fault.

7.7.3 Changes in Version 0.2.3

Bugfix release.

Issues Resolved

Fixed a bug with query sorts.

7.7.4 Changes in Version 0.1.4

Upgraded to the latest version of the MongoDB C driver - changed the signature for the `mongo_connect()` function.

7.7.5 Changes in Version 0.1.3

Added support for sorting queries and providing hints - see `Monary.query`.

Added simple unit tests for `Monary.authenticate`.

7.7.6 Changes in Version 0.1.2

Added support for a "date" type which populates an array of in64 values from a BSON date. The date value is milliseconds since January 1, 1970.

Column tests improved.

Strict argument checks added to datehelper functions.

Issues Resolved

Fixed a minor bug in `datehelper.mongodelta_to_timedelta()`, which was not accepting a `numpy.int64` instance as the date value. (Now we simply convert the argument to a Python int.)

7.7.7 Changes in Version 0.1.1

Added support for int8, int16, int64 and float32 column types. Also added basic tests for all column types (requires `nosetests`).

To run the tests, first obtain `nosetests`:

```
$ pip install nose
```

Then, to test:

```
$ nosetests
```

Issues Resolved

Fixed issue with ObjectIDs containing NULL bytes. (ObjectIDs now use a 12-byte 'void' array type in numeric Python.)

7.7.8 Changes in Version 0.1.0

Initial release.

7.8 Contributors

Author: David J. C. Beach (djcbeach)

Below is an alphabetical list of people who have contributed to **Monary**. If you are missing here, please let us know!

- A. Jesse Jiryu Davis (ajdavis)
- Anna Herlihy (aherlihy)
- itaii
- Jason Carey (hanumantmk)
- Kyle Suarez (ksuarz)
- Matt Cotter (machyne)