PyMongoArrow

Release 0.2.0

Prashant Mital

Mar 02, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

PyMongoArrow is a PyMongo extension containing tools for loading MongoDB query result sets as Apache Arrow tables, Pandas and NumPy arrays. PyMongoArrow is the recommended way to materialize MongoDB query result sets as contiguous-in-memory, typed arrays suited for in-memory analytical processing applications. This documentation attempts to explain everything you need to know to use **PyMongoArrow**.

Installing / Upgrading Instructions on how to get the distribution.

Quick Start Start here for a quick overview.

Supported Types A list of BSON types that are supported by PyMongoArrow.

faq Frequently asked questions.

pymongoarrow – Tools for working with MongoDB and PyArrow The complete API documentation, organized by module.

CHAPTER

TWO

GETTING HELP

If you're having trouble or have questions about PyMongoArrow, ask your question on our MongoDB Community Forum. Once you get an answer, it'd be great if you could work it back into this documentation and contribute!

CHAPTER

THREE

ISSUES

All issues should be reported (and can be tracked / voted for / commented on) at the main MongoDB JIRA bug tracker, in the "Python Driver" project.

CHAPTER

FOUR

FEATURE REQUESTS / FEEDBACK

Use our feedback engine to send us feature requests and general feedback about PyMongoArrow.

CHAPTER

FIVE

CONTRIBUTING

Contributions to **PyMongoArrow** are encouraged. To contribute, fork the project on GitHub and send a pull request.

See also *Developer Guide*.

CHAPTER

SIX

CHANGES

See the *Changelog* for a full list of changes to PyMongoArrow.

CHAPTER

SEVEN

ABOUT THIS DOCUMENTATION

This documentation is generated using the Sphinx documentation generator. The source files for the documentation are located in the *docs/* directory of the **PyMongoArrow** distribution. To generate the docs locally run the following command from the root directory of the **PyMongoArrow** source:

```
$ cd docs && make html
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

8.1 Installing / Upgrading

8.1.1 System Compatibility

PyMongoArrow is regularly built and tested on macOS and Linux (Ubuntu 20.04).

8.1.2 Python Compatibility

PyMongoArrow is currently compatible with CPython 3.6, 3.7, 3.8 and 3.9.

8.1.3 Using Pip

PyMongoArrow is available on PyPI. We recommend using pip to install `pymongoarrow` on all platforms:

```
$ python -m pip install pymongoarrow
```

To get a specific version of pymongo:

```
$ python -m pip install pymongoarrow==0.1.1
```

To upgrade using pip:

```
$ python -m pip install --upgrade pymongoarrow
```

Attention: Installing PyMongoArrow from wheels on macOS Big Sur requires `pip` >= 20.3. To upgrade `pip` run:

```
$ python -m pip install --upgrade pip
```

We currently distribute wheels for macOS and Linux on x86_64 architectures.

Dependencies

PyMongoArrow requires:

- PyMongo>=3.11,<4
- PyArrow>=3,<3.1

To use PyMongoArrow with a PyMongo feature that requires an optional dependency, users must install PyMongo with the given dependency manually.

Note: PyMongo's optional dependencies are detailed here.

For example, to use PyMongoArrow with MongoDB Atlas' `mongodb+srv://` URIs users must install PyMongo with the `srv` extra in addition to installing PyMongoArrow:

```
$ python -m pip install 'pymongo[srv]<4' pymongoarrow
```

Applications intending to use PyMongoArrow APIs that return query result sets as `pandas.DataFrame` instances (e.g. *`find_pandas_all()`*) must also have `pandas` installed:

```
$ python -m pip install pandas
```

8.1.4 Installing from source

See *Installing from source*.

8.2 Quick Start

This tutorial is intended as an introduction to working with **PyMongoArrow**. The reader is assumed to be familiar with basic PyMongo and MongoDB concepts.

8.2.1 Prerequisites

Before we start, make sure that you have the **PyMongoArrow** distribution *installed*. In the Python shell, the following should run without raising an exception:

```
import pymongoarrow as pma
```

This tutorial also assumes that a MongoDB instance is running on the default host and port. Assuming you have downloaded and installed MongoDB, you can start it like so:

```
$ mongod
```

Extending PyMongo

The *pymongoarrow.monkey* module provides an interface to patch PyMongo, in place, and add **PyMongoArrow**'s functionality directly to `Collection` instances:

```
from pymongoarrow.monkey import patch_all
patch_all()
```

After running *patch_all()*, new instances of `Collection` will have PyMongoArrow's APIs, e.g. *find_pandas_all()*.

Note: Users can also directly use any of **PyMongoArrow**'s APIs by importing them from *pymongoarrow.api*. The only difference in usage would be the need to manually pass the instance of `Collection` on which the operation is to be run as the first argument when directly using the API method.

Test data

Before we begein, we must first add some data to our cluster that we can query. We can do so using **PyMongo**:

```
from datetime import datetime
from pymongo import MongoClient
client = MongoClient()
client.db.data.insert_many([
    {'_id': 1, 'amount': 21, 'last_updated': datetime(2020, 12, 10, 1, 3, 1)},
    {'_id': 2, 'amount': 16, 'last_updated': datetime(2020, 7, 23, 6, 7, 11)},
    {'_id': 3, 'amount': 3, 'last_updated': datetime(2021, 3, 10, 18, 43, 9)},
    {'_id': 4, 'amount': 0, 'last_updated': datetime(2021, 2, 25, 3, 50, 31)}])
```

8.2.2 Defining the schema

PyMongoArrow relies upon a **user-specified** data schema to marshall query result sets into tabular form. Users can define the schema by instantiating *pymongoarrow.api.Schema* using a mapping of field names to type-specifiers, e.g.:

```
from pymongoarrow.api import Schema
schema = Schema({'_id': int, 'amount': float, 'last_updated': datetime})
```

There are multiple permissible type-identifiers for each supported BSON type. For a full-list of supported types and associated type-identifiers see *Supported Types*.

8.2.3 Find operations

We are now ready to query our data. Let's start by running a `find` operation to load all records with a non-zero `amount` as a `pandas.DataFrame`:

```
df = client.db.data.find_pandas_all({'amount': {'$gt': 0}}, schema=schema)
```

We can also load the same result set as a `pyarrow.Table` instance:

```
arrow_table = client.db.data.find_arrow_all({'amount': {'$gt': 0}}, schema=schema)
```

Or as `numpy.ndarray` instances:

```
ndarrays = client.db.data.find_numpy_all({'amount': {'$gt': 0}}, schema=schema)
```

In the NumPy case, the return value is a dictionary where the keys are field names and values are the corresponding arrays.

8.2.4 Aggregate operations

Running `aggregate` operations is similar to `find`. Here is an example of an aggregation that loads all records with an `amount` less than 10:

```
# pandas
df = client.db.data.aggregate_pandas_all([{'$match': {'amount': {'$lte': 10}}}],␣
↪schema=schema)
# arrow
arrow_table = client.db.data.aggregate_arrow_all([{'$match': {'amount': {'$lte': 10}}}],␣
↪schema=schema)
# numpy
ndarrays = client.db.data.aggregate_numpy_all([{'$match': {'amount': {'$lte': 10}}}],␣
↪schema=schema)
```

8.2.5 Writing to other formats

Result sets that have been loaded as Arrow's `Table` type can be easily written to one of the formats supported by PyArrow. For example, to write the table referenced by the variable `arrow_table` to a Parquet file `example.parquet`, run:

```
import pyarrow.parquet as pq
pq.write_table(arrow_table, 'example.parquet')
```

Pandas also supports writing `DataFrame` instances to a variety of formats including CSV, and HDF. For example, to write the data frame referenced by the variable `df` to a CSV file `out.csv`, run:

```
df.to_csv('out.csv', index=False)
```

8.3 Supported Types

PyMongoArrow currently supports a small subset of all BSON types. Support for additional types will be added in subsequent releases.

Note: For more information about BSON types, see the BSON specification.

BSON Type	Type Identifiers
64-bit binary floating point	`py.float`, an instance of `pyarrow.float64()`
32-bit integer	an instance of `pyarrow.int32()`
64-bit integer	`int`, `bson.int64.Int64`, an instance of `pyarrow.int64()`
UTC datetime	an instance of `timestamp` with `ms` resolution, `py.datetime.datetime`

Type identifiers can be used to specify that a field is of a certain type during *pymongoarrow.api.Schema* declaration. For example, if your data has fields 'f1' and 'f2' bearing types 32-bit integer and UTC datetime respectively, your schema can be defined as:

```
schema = Schema({'f1': pyarrow.int32(), 'f2': pyarrow.timestamp('ms')})
```

8.4 pymongoarrow – Tools for working with MongoDB and PyArrow

Sub-modules:

8.4.1 api – PyMongoArrow APIs

class pymongoarrow.api.**Schema**(*schema*)

A mapping of field names to data types.

To create a schema, provide its constructor a mapping of field names to their expected types, e.g.:

```
schema1 = Schema({'field_1': int, 'field_2': float})
```

Each key in `schema` is a field name and its corresponding value is the expected type of the data contained in the named field.

Data types can be specified as pyarrow type instances (e.g. an instance of `pyarrow.int64`), bson types (e.g. `bson.Int64`), or python type-identifiers (e.g. `int`, `float`). To see a complete list of supported data types and their corresponding type-identifiers, see *Supported Types*.

pymongoarrow.api.**aggregate_arrow_all**(*collection*, *pipeline*, *, *schema*, ***kwargs*)

Method that returns the results of an aggregation pipeline as a `pyarrow.Table` instance.

Parameters

- *collection*: Instance of `Collection`. against which to run the `aggregate` operation.
- *pipeline*: A list of aggregation pipeline stages.
- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `aggregate` operation.

Returns An instance of class:*pyarrow.Table*.

pymongoarrow.api.**aggregate_numpy_all**(*collection*, *pipeline*, *, *schema*, ***kwargs*)

Method that returns the results of an aggregation pipeline as a `dict` instance whose keys are field names and values are `ndarray` instances bearing the appropriate dtype.

Parameters

- *collection*: Instance of `Collection`. against which to run the `find` operation.
- *query*: A mapping containing the query to use for the find operation.

- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `aggregate` operation.

This method attempts to create each NumPy array as a view on the Arrow data corresponding to each field in the result set. When this is not possible, the underlying data is copied into a new NumPy array. See `pyarrow.Array.to_numpy()` for more information.

NumPy arrays returned by this method that are views on Arrow data are not writable. Users seeking to modify such arrays must first create an editable copy using `numpy.copy()`.

Returns An instance of `dict`.

pymongoarrow.api.**aggregate_pandas_all**(*collection*, *pipeline*, *, *schema*, ***kwargs*)

Method that returns the results of an aggregation pipeline as a `pandas.DataFrame` instance.

Parameters

- *collection*: Instance of `Collection`. against which to run the `find` operation.
- *pipeline*: A list of aggregation pipeline stages.
- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `aggregate` operation.

Returns An instance of class:*pandas.DataFrame*.

pymongoarrow.api.**find_arrow_all**(*collection*, *query*, *, *schema*, ***kwargs*)

Method that returns the results of a find query as a `pyarrow.Table` instance.

Parameters

- *collection*: Instance of `Collection`. against which to run the `find` operation.
- *query*: A mapping containing the query to use for the find operation.
- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `find` operation.

Returns An instance of class:*pyarrow.Table*.

pymongoarrow.api.**find_numpy_all**(*collection*, *query*, *, *schema*, ***kwargs*)

Method that returns the results of a find query as a `dict` instance whose keys are field names and values are `ndarray` instances bearing the appropriate dtype.

Parameters

- *collection*: Instance of `Collection`. against which to run the `find` operation.
- *query*: A mapping containing the query to use for the find operation.
- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `find` operation.

This method attempts to create each NumPy array as a view on the Arrow data corresponding to each field in the result set. When this is not possible, the underlying data is copied into a new NumPy array. See `pyarrow.Array.to_numpy()` for more information.

NumPy arrays returned by this method that are views on Arrow data are not writable. Users seeking to modify such arrays must first create an editable copy using `numpy.copy()`.

Returns An instance of `dict`.

pymongoarrow.api.**find_pandas_all**(*collection*, *query*, *, *schema*, ***kwargs*)

Method that returns the results of a find query as a `pandas.DataFrame` instance.

Parameters

- *collection*: Instance of `Collection`. against which to run the `find` operation.
- *query*: A mapping containing the query to use for the find operation.
- *schema*: Instance of *Schema*.

Additional keyword-arguments passed to this method will be passed directly to the underlying `find` operation.

Returns An instance of class:*pandas.DataFrame*.

8.4.2 monkey – Add PyMongoArrow APIs to PyMongo

Add PyMongoArrow APIs to PyMongo.

pymongoarrow.monkey.**patch_all**()

Patch all PyMongoArrow methods into PyMongo.

Calling this method equips the `pymongo.collection.Collection` classes returned by PyMongo with PyMongoArrow's API methods. When using a patched method, users can omit the first argument which is passed implicitly. For example:

```
# Example of direct usage
df = find_pandas_all(coll.db.test, {'amount': {'$gte': 20}}, schema=schema)

# Example of patched usage
df = coll.db.test.find_pandas_all({'amount': {'$gte': 20}}, schema=schema)
```

8.5 Changelog

8.5.1 Changes in Version 0.2.0

- Support for PyMongo 4.0.
- Support for Python 3.10.
- Support for Windows.
- `find_arrow_all` now accepts a user-provided `projection`.
- `find_raw_batches` now accepts a `session` object.
- Note: The supported version of `pyarrow` is now `>=6,<6.1`.

8.5.2 Changes in Version 0.1.1

- Fixed a bug that caused Linux wheels to be created without the appropriate `manylinux` platform tags.

8.5.3 Changes in Version 0.1.0

- Support for efficiently converting find and aggregate query result sets into Arrow/Pandas/Numpy data structures.
- Support for patching PyMongo's APIs using *patch_all()*
- Support for loading the following BSON types:
  - 64-bit binary floating point
  - 32-bit integer
  - 64-bit integer
  - Timestamp

8.6 Developer Guide

Technical guide for contributors to PyMongoArrow.

8.6.1 Installing from source

System Requirements

On macOS, you need a working modern XCode installation with the XCode Command Line Tools. Additionally, you need CMake and pkg-config:

```
$ xcode-select --install
$ brew install cmake
$ brew install pkg-config
```

On Linux, you require gcc 4.8, CMake and pkg-config.

Windows is not yet supported.

Environment Setup

First, clone the mongo-arrow git repository:

```
$ git clone https://github.com/mongodb-labs/mongo-arrow.git
$ cd mongo-arrow/bindings/python
```

Additionally, create a virtualenv in which to install `pymongoarrow` from sources:

```
$ virtualenv pymongoarrow
$ source ./pymongoarrow/bin/activate
```

libbson

PyMongoArrow uses libbson. Detailed instructions for building/installing `libbson` can be found here.

On macOS, users can install the latest `libbson` via Homebrew:

```
$ brew install mongo-c-driver
```

Alternatively, you can use the provided *build-libbson.sh* script to build it:

```
$ LIBBSON_INSTALL_DIR=$(pwd)/libbson ./build-libbson.sh
```

Build

In the previously created virtualenv, install PyMongoArrow and its test dependencies in editable mode:

```
(pymongoarrow) $ pip install -v -e ".[test]"
```

If you built libbson using the *build-libbson* script then use the same *LIBBSON_INSTALL_DIR* as above:

(pymongoarrow) $ LIBBSON_INSTALL_DIR=$(pwd)/libbson pip install -v -e ".[test]"

Test

To run the test suite, you will need a MongoDB instance running on `localhost` using port `27017`. To run the entire test suite, do:

```
(pymongoarrow) $ python -m unittest discover test
```

PYTHON MODULE INDEX

p

INDEX