
pymapd Documentation

Release 0.7.2.dev6+g941c872

Tom Augspurger

Feb 08, 2019

Contents:

1	Install	3
2	Usage	5
2.1	Connecting	5
2.2	Querying	6
2.3	Cursors	6
2.4	Loading Data	7
2.5	Database Metadata	8
3	API Reference	9
3.1	Exceptions	15
4	Development Setup	17
5	Thrift Binding	19
6	Indices and tables	21
	Python Module Index	23

The pymapd client interface provides a python DB API 2.0-compliant [OmniSci](#) interface (formerly MapD). In addition, it provides methods to get results in the [Apache Arrow](#) -based GDF format for efficient data interchange.

```
>>> from pymapd import connect
>>> con = connect(user="mapd", password="HyperInteractive", host="localhost",
...              dbname="mapd")
>>> query = "SELECT depdelay, arrdelay FROM flights_2008_10k limit 100"
>>> df = con.select_ipc_gpu("SELECT depdelay, arrdelay "
...                        "FROM flights_2008_10k "
...                        "LIMIT 100")
>>> df.head()
   depdelay  arrdelay
0         -2        -13
1         -1        -13
2         -3         1
3          4         -3
4         12         7
```


This describes how to install the python package. To setup an OmniSci (formerly MapD) server, see [here](#).

`pymapd` can be installed with conda using [conda-forge](#) or `pip`.

```
# conda
conda install -c conda-forge pymapd

# pip
pip install pymapd
```

This actually installs two packages, `pymapd`, the pythonic interface to OmniSci, and `mapd`, the Apache thrift bindings for OmniSci.

There are several optional dependencies that may be useful. To return results sets into GPU memory as a `Gpu-DataFrame`, you'll need [cudf](#). To return results into CPU shared memory, using the [Apache Arrow](#) memory layout, you'll need [pyarrow](#) and its dependencies.

pymapd follows the python DB API 2.0, so this may already be familiar to you.

Note: This assumes you have an OmniSci server running on `localhost:9091` with the default logins and databases, and have loaded the example “flights_2008_10k” dataset.

2.1 Connecting

Create a *Connection* with

```
>>> from pymapd import connect
>>> con = connect(user="mapd", password="HyperInteractive", host="localhost",
...              dbname="mapd")
>>> con
Connection(mapd://mapd:***@localhost:9091/mapd?protocol=binary)
```

or by passing in a connection string

```
>>> uri = "mapd://mapd:HyperInteractive@localhost:9091/mapd?protocol=binary"
>>> con = connect(uri=uri)
Connection(mapd://mapd:***@localhost:9091/mapd?protocol=binary)
```

See the [SQLAlchemy](#) documentation on what makes up a connection string. The components are:

```
dialect+driver://username:password@host:port/database
```

For pymapd, the `dialect+driver` will always be `mapd`, and we look for a `protocol` argument in the optional query parameters (everything following the `?` after database).

2.2 Querying

A few options are available for getting the results of a query into your Python process.

1. Into GPU Memory via `cuda` (`Connection.select_ipc_gpu()`)
2. Into CPU shared memory via Apache Arrow and pandas (`Connection.select_ipc()`)
3. Into python objects via Apache Thrift (`Connection.execute()`)

The best option depends on the hardware you have available, your connection to the database, and what you plan to do with the returned data. In general, the third method, using Thrift to serialize and deserialize the data, will slower than the GPU or CPU shared memory methods. The shared memory methods require that your OmniSci database is running on the same machine.

2.2.1 GPU Select

Use `Connection.select_ipc_gpu()` to select data into a `GpuDataFrame`, provided by `cuda`

```
>>> query = "SELECT depdelay, arrdelay FROM flights_2008_10k limit 100"
>>> df = con.select_ipc_gpu(query)
>>> df.head()
   depdelay  arrdelay
0         -2        -13
1         -1        -13
2         -3         1
3          4         -3
4         12         7
```

2.2.2 CPU Shared Memory Select

Use `Connection.select_ipc()` to select data into a pandas `DataFrame` using CPU shared memory to avoid unnecessary intermediate copies.

```
>>> df = con.select_ipc(query)
>>> df.head()
   depdelay  arrdelay
0         -2        -13
1         -1        -13
2         -3         1
3          4         -3
4         12         7
```

2.3 Cursors

A cursor can be created with `Connection.cursor()`

```
>>> c = con.cursor()
>>> c
<pymapd.cursor.Cursor at 0x110fe6438>
```

Or by using a context manager:

```
>>> with con as c:
...     print(c)
<pymapd.cursor.Cursor object at 0x1041f9630>
```

Arbitrary SQL can be executed using `Cursor.execute()`.

```
>>> c.execute("SELECT depdelay, arrdelay FROM flights_2008_10k limit 100")
<pymapd.cursor.Cursor at 0x110fe6438>
```

This will set the `rowcount` property, with the number of returned rows

```
>>> c.rowcount
100
```

The `description` attribute contains a list of `Description` objects, a namedtuple with the usual attributes required by the spec. There's one entry per returned column, and we fill the `name`, `type_code` and `null_ok` attributes.

```
>>> c.description
[Description(name='depdelay', type_code=0, display_size=None, internal_size=None,
↳ precision=None, scale=None, null_ok=True),
 Description(name='arrdelay', type_code=0, display_size=None, internal_size=None,
↳ precision=None, scale=None, null_ok=True)]
```

Cursors are iterable, returning a list of tuples of values

```
>>> result = list(c)
>>> result[:5]
[(38, 28), (0, 8), (-4, 9), (1, -1), (1, 2)]
```

2.4 Loading Data

The fastest way to load data is `Connection.load_table_arrow()`. Internally, this will use `pyarrow` and the `Apache Arrow` format to exchange data with the `OmniSci` database.

```
>>> import pyarrow as pa
>>> import pandas as pd
>>> df = pd.DataFrame({"A": [1, 2], "B": ['c', 'd']})
>>> table = pa.Table.from_pandas(df)
>>> con.load_table_arrow("table_name", table)
```

This accepts either a `pyarrow.Table`, or a `pandas.DataFrame`, which will be converted to a `pyarrow.Table` before loading.

You can also load a `pandas.DataFrame` using `Connection.load_table()` or `Connection.load_table_columnar()` methods.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": ["c", "d"]})
>>> con.load_table_columnar("table_name", df, preserve_index=False)
```

If you aren't using `arrow` or `pandas` you can pass list of tuples to `Connection.load_table_rowwise()`.

```
>>> data = [(1, "c"), (2, "d")]
>>> con.load_table_rowwise("table_name", data)
```

The high-level `Connection.load_table()` method will choose the fastest method available based on the type of data and whether or not `pyarrow` is installed.

- lists of tuples are always loaded with `Connection.load_table_rowwise()`
- If `pyarrow` is installed, a `pandas.DataFrame` or `pyarrow.Table` will be loaded using `Connection.load_table_arrow()`
- If `pyarrow` is not installed, a `pandas.DataFrame` will be loaded using `Connection.load_table_columnar()`

2.5 Database Metadata

Some helpful metadata are available on the `Connection` object.

1. Get a list of tables with `Connection.get_tables()`

```
>>> con.get_tables()
['flights_2008_10k', 'stocks']
```

2. Get column information for a table with `Connection.get_table_details()`

```
>>> con.get_table_details('stocks')
[ColumnDetails(name='date_', type='STR', nullable=True, precision=0,
                scale=0, comp_param=32),
 ColumnDetails(name='trans', type='STR', nullable=True, precision=0,
                scale=0, comp_param=32),
 ...]
```

`pymapd.connect` (*uri=None, user=None, password=None, host=None, port=9091, dbname=None, protocol='binary'*)

Crate a new Connection.

Parameters

uri [str]

user [str]

password [str]

host [str]

port [int]

dbname [str]

protocol [{'binary', 'http', 'https'}]

Returns

conn [Connection]

Examples

You can either pass a string `uri` or all the individual components

```
>>> connect('mapd://mapd:HyperInteractive@localhost:9091/mapd?'
...         'protocol=binary')
Connection(mapd://mapd:***@localhost:9091/mapd?protocol=binary)
```

```
>>> connect(user='mapd', password='HyperInteractive', host='localhost',
...         port=9091, dbname='mapd')
```

class pymapd.**Connection** (*uri=None, user=None, password=None, host=None, port=9091, db-name=None, protocol='binary'*)

Connect to your OmniSci database.

close ()

Disconnect from the database

commit ()

This is a noop, as OmniSci does not provide transactions.

Implementing to comply with the specification.

create_table (*table_name, data, preserve_index=False*)

Create a table from a pandas.DataFrame

Parameters

table_name [str]

data [DataFrame]

preserve_index [bool, default False] Whether to create a column in the table for the DataFrame index

cursor ()

Create a new *Cursor* object attached to this connection.

deallocate_ipc (*df, device_id=0*)

Deallocate a DataFrame using CPU shared memory.

Parameters

device_id [int] GPU which contains TDataFrame

deallocate_ipc_gpu (*df, device_id=0*)

Deallocate a DataFrame using GPU memory.

Parameters

device_id [int] GPU which contains TDataFrame

execute (*operation, parameters=None*)

Execute a SQL statement

Parameters

operation [str] A SQL statement to execute

Returns

c [Cursor]

get_table_details (*table_name*)

Get the column names and data types associated with a table.

Parameters

table_name [str]

Returns

details [List[tuples]]

Examples

```
>>> con.get_table_details('stocks')
[ColumnDetails(name='date_', type='STR', nullable=True, precision=0,
                scale=0, comp_param=32),
 ColumnDetails(name='trans', type='STR', nullable=True, precision=0,
                scale=0, comp_param=32),
 ...
]
```

`get_tables()`

List all the tables in the database

Examples

```
>>> con.get_tables()
['flights_2008_10k', 'stocks']
```

`load_table(table_name, data, method='infer', preserve_index=False, create='infer')`

Load data into a table

Parameters

table_name [str]

data [pyarrow.Table, pandas.DataFrame, or iterable of tuples]

method [{"infer", "columnar", "rows"}] Method to use for loading the data. Three options are available

1. pyarrow and Apache Arrow loader
2. columnar loader
3. row-wise loader

The Arrow loader is typically the fastest, followed by the columnar loader, followed by the row-wise loader. If a DataFrame or pyarrow.Table is passed and pyarrow is installed, the Arrow-based loader will be used. If arrow isn't available, the columnar loader is used. Finally, data is an iterable of tuples the row-wise loader is used.

preserve_index [bool, default False] Whether to keep the index when loading a pandas DataFrame

create [{"infer", True, False}] Whether to issue a CREATE TABLE before inserting the data.

- infer : check to see if the table already exists, and create a table if it does not
- True : attempt to create the table, without checking if it exists
- False : do not attempt to create the table

See also:

load_table_arrow, load_table_columnar

`load_table_arrow(table_name, data, preserve_index=False)`

Load a pandas.DataFrame or a pyarrow Table or RecordBatch to the database using Arrow columnar format for interchange

Parameters

table_name [str]

data [pandas.DataFrame, pyarrow.RecordBatch, pyarrow.Table]

preserve_index [bool, default False] Whether to include the index of a pandas DataFrame when writing.

See also:

[load_table](#), [load_table_columnar](#), [load_table_rowwise](#)

Examples

```
>>> df = pd.DataFrame({"a": [1, 2, 3], "b": ['d', 'e', 'f']})
>>> con.load_table_arrow('foo', df, preserve_index=False)
```

load_table_columnar (*table_name*, *data*, *preserve_index=False*, *chunk_size_bytes=0*, *col_names_from_schema=False*)

Load a pandas DataFrame to the database using OmniSci's Thrift-based columnar format

Parameters

table_name [str]

data [DataFrame]

preserve_index [bool, default False] Whether to include the index of a pandas DataFrame when writing.

chunk_size_bytes [integer, default 0] Chunk the loading of columns to prevent large Thrift requests. A value of 0 means do not chunk and send the dataframe as a single request

col_names_from_schema [bool, default False] Read the existing table schema to determine the column names. This will read the schema of an existing table in OmniSci and match those names to the column names of the dataframe. This is for user convenience when loading from data that is unordered, especially handy when a table has a large number of columns.

See also:

[load_table](#), [load_table_arrow](#), [load_table_rowwise](#)

Examples

```
>>> df = pd.DataFrame({"a": [1, 2, 3], "b": ['d', 'e', 'f']})
>>> con.load_table_columnar('foo', df, preserve_index=False)
```

load_table_rowwise (*table_name*, *data*)

Load data into a table row-wise

Parameters

table_name [str]

data [Iterable of tuples] Each element of *data* should be a row to be inserted

See also:

[load_table](#), [load_table_arrow](#), [load_table_columnar](#)

Examples

```
>>> data = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> con.load_table('bar', data)
```

render_vega (*vega, compression_level=1*)

Render vega data on the database backend, returning the image as a PNG.

Parameters

vega [dict] The vega specification to render.

compression_level: int The level of compression for the rendered PNG. Ranges from 0 (low compression, faster) to 9 (high compression, slower).

select_ipc (*operation, parameters=None, first_n=-1, release_memory=True*)

Execute a SELECT operation using CPU shared memory

Parameters

operation [str] A SQL select statement

parameters [dict, optional] Parameters to insert for a parametrized query

first_n [int, optional] Number of records to return

release_memory: bool, optional Call `self.deallocate_ipc(df)` after DataFrame created

Returns

df [pandas.DataFrame]

Notes

This method requires pyarrow to be installed.

select_ipc_gpu (*operation, parameters=None, device_id=0, first_n=-1, release_memory=True*)

Execute a SELECT operation using GPU memory.

Parameters

operation [str] A SQL statement

parameters [dict, optional] Parameters to insert into a parametrized query

device_id [int] GPU to return results to

first_n [int, optional] Number of records to return

release_memory: bool, optional Call `self.deallocate_ipc_gpu(df)` after DataFrame created

Returns

gdf [cudf.GpuDataFrame]

Notes

This method requires cudf and libcudf to be installed. An ImportError is raised if those aren't available.

class `pymapd.Cursor` (*connection, columnar=True*)

A database cursor.

arraysize

The number of rows to fetch at a time with *fetchmany*. Default 1.

See also:

fetchmany

close()

Close this cursor.

description

Read-only sequence describing columns of the result set. Each column is an instance of *Description* describing

- name
- type_code
- display_size
- internal_size
- precision
- scale
- null_ok

We only use name, type_code, and null_ok; The rest are always None

execute (*operation, parameters=None*)

Execute a SQL statement.

Parameters

operation [str] A SQL query

parameters [dict] Parameters to substitute into operation.

Returns

self [Cursor]

Examples

```
>>> c = conn.cursor()
>>> c.execute("select symbol, qty from stocks")
>>> list(c)
[('RHAT', 100.0), ('IBM', 1000.0), ('MSFT', 1000.0), ('IBM', 500.0)]
```

Passing in parameters:

```
>>> c.execute("select symbol qty from stocks where qty <= :max_qty",
...           parameters={"max_qty": 500})
[('RHAT', 100.0), ('IBM', 500.0)]
```

executemany (*operation, parameters*)

Execute a SQL statement for many sets of parameters.

Parameters

operation [str]

parameters [list of dict]

Returns

results [list of lists]

fetchmany (*size=None*)

Fetch *size* rows from the results set.

fetchone ()

Fetch a single row from the results set

3.1 Exceptions

Define exceptions as specified by the DB API 2.0 spec.

Includes some helper methods for translating thrift exceptions to the ones defined here.

exception `pymapd.exceptions.Error`

Base class for all pymapd errors.

exception `pymapd.exceptions.InterfaceError`

Raised whenever you use pymapd interface incorrectly.

exception `pymapd.exceptions.DatabaseError`

Raised when the database encounters an error.

exception `pymapd.exceptions.OperationalError`

Raised for non-programmer related database errors, e.g. an unexpected disconnect.

exception `pymapd.exceptions.IntegrityError`

Raised when the relational integrity of the database is affected.

exception `pymapd.exceptions.InternalError`

Raised for errors internal to the database, e.g. and invalid cursor.

exception `pymapd.exceptions.ProgrammingError`

Raised for programming errors, e.g. syntax errors, table already exists.

exception `pymapd.exceptions.NotSupportedError`

Raised when an API not supported by the database is used.

CHAPTER 4

Development Setup

TODO

CHAPTER 5

Thrift Binding

When the upstream [mapd-core](#) project updates its Thrift definition, we have to regenerate the bindings we ship with pymapd. From the root of the pymapd repository, run

```
python scripts/generate_accelerated_bindings.py </path/to/mapd-core>/mapd.thrift
```

This requires that Thrift is installed and on your PATH. Running it will update two files, `mapd/MapD.py` and `mapd/ttypes.py`, which can be committed to the repository.

CHAPTER 6

Indices and tables

- `genindex`
- `modindex`
- `search`

p

`pymapd`, 9

`pymapd.exceptions`, 15

A

arraysize (*pymapd.Cursor attribute*), 14

C

close() (*pymapd.Connection method*), 10

close() (*pymapd.Cursor method*), 14

commit() (*pymapd.Connection method*), 10

connect() (*in module pymapd*), 9

Connection (*class in pymapd*), 9

create_table() (*pymapd.Connection method*), 10

Cursor (*class in pymapd*), 13

cursor() (*pymapd.Connection method*), 10

D

DatabaseError, 15

deallocate_ipc() (*pymapd.Connection method*), 10

deallocate_ipc_gpu() (*pymapd.Connection method*), 10

description (*pymapd.Cursor attribute*), 14

E

Error, 15

execute() (*pymapd.Connection method*), 10

execute() (*pymapd.Cursor method*), 14

executemany() (*pymapd.Cursor method*), 14

F

fetchmany() (*pymapd.Cursor method*), 15

fetchone() (*pymapd.Cursor method*), 15

G

get_table_details() (*pymapd.Connection method*), 10

get_tables() (*pymapd.Connection method*), 11

I

IntegrityError, 15

InterfaceError, 15

InternalError, 15

L

load_table() (*pymapd.Connection method*), 11

load_table_arrow() (*pymapd.Connection method*), 11

load_table_columnar() (*pymapd.Connection method*), 12

load_table_rowwise() (*pymapd.Connection method*), 12

N

NotSupportedError, 15

O

OperationalError, 15

P

ProgrammingError, 15

pymapd (*module*), 9

pymapd.exceptions (*module*), 15

R

render_vega() (*pymapd.Connection method*), 13

S

select_ipc() (*pymapd.Connection method*), 13

select_ipc_gpu() (*pymapd.Connection method*), 13