pymapd Documentation

Release 0.24.1

Tom Augspurger

Jul 28, 2020

CONTENTS:

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 cudf GPU DataFrame format for efficient data interchange.

```
>>> from pymapd import connect
>>> con = connect(user="admin", password="HyperInteractive", host="localhost",
...               dbname="omnisci")
>>> 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
```

CHAPTER

ONE

5-MINUTE QUICKSTART

pymapd follows the python DB API 2.0, so experience with other Python database clients will feel similar to pymapd.

Note: This tutorial assumes you have an OmniSci server running on localhost:6274 with the default logins and databases, and have loaded the example flights_2008_10k dataset. This dataset can be obtained from the insert_sample_data script included in the OmniSci install directory.

1.1 Installing pymapd

1.1.1 pymapd

pymapd can be installed with conda using conda-forge or pip.

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

# pip
pip install pymapd
```

If you have an NVIDIA GPU in the same machine where your pymapd code will be running, you'll want to install cudf as well to return results sets into GPU memory as a cudf GPU DataFrame:

1.1.2 cudf via conda

```
# CUDA 9.2
conda install -c nvidia -c rapidsai -c numba -c conda-forge -c defaults cudf

# CUDA 10.0
conda install -c nvidia/label/cuda10.0 -c rapidsai/label/cuda10.0 -c numba \
    -c conda-forge -c defaults cudf
```

1.1.3 cudf via PyPI/pip

```
# CUDA 9.2
pip install cudf-cuda92

# CUDA 10.0
pip install cudf-cuda100
```

1.2 Connecting

1.2.1 Self-Hosted Install

For self-hosted OmniSci installs, use `protocol='binary'` (this is the default) to connect with OmniSci, as this will have better performance than using `protocol='http'` or `protocol='https'`.

To create a *`Connection`* using the `connect()` method along with `user`, `password`, `host` and `dbname`:

```
>>> from pymapd import connect
>>> con = connect(user="admin", password="HyperInteractive", host="localhost",
...               dbname="omnisci")
>>> con
Connection(mapd://admin:***@localhost:6274/omnisci?protocol=binary)
```

Alternatively, you can pass in a SQLAlchemy-compliant connection string to the `connect()` method:

```
>>> uri = "mapd://admin:HyperInteractive@localhost:6274/omnisci?protocol=binary"
>>> con = connect(uri=uri)
Connection(mapd://admin:***@localhost:6274/omnisci?protocol=binary)
```

1.2.2 OmniSci Cloud

When connecting to OmniSci Cloud, the two methods are the same as above, however you can only use `protocol='https'`. For a step-by-step walk-through with screenshots, please see this blog post.

1.3 Querying

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

1. Into GPU Memory via cudf (*`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 be slower than the GPU or CPU shared memory methods. The shared memory methods require that your OmniSci database is running on the same machine.

Note: We currently support `Timestamp(0|3|6)` data types i.e. seconds, milliseconds, and microseconds granularity. Support for nanoseconds, `Timestamp(9)` is in progress.

1.3.1 GPU Shared Memory

Use *Connection.select_ipc_gpu()* to select data into a GpuDataFrame, provided by cudf. To use this method, **the Python code must be running on the same machine as the OmniSci installation AND you must have an NVIDIA GPU installed.**

```
>>> 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
```

1.3.2 CPU Shared Memory

Use *Connection.select_ipc()* to select data into a pandas DataFrame using CPU shared memory to avoid unnecessary intermediate copies. To use this method, **the Python code must be running on the same machine as the OmniSci installation.**

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

1.3.3 pandas.read_sql()

With a *Connection* defined, you can use pandas.read_sql() to read your data in a pandas DataFrame. This will be slower than using *Connection.select_ipc()*, but works regardless of where the Python code is running (i.e. select_ipc() must be on the same machine as the OmniSci install, pandas.read_sql() works everywhere):

```
>>> from pymapd import connect
>>> import pandas as pd
>>> con = connect(user="admin", password="HyperInteractive", host="localhost",
...               dbname="omnisci")
>>> df = pd.read_sql("SELECT depdelay, arrdelay FROM flights_2008_10k limit 100", con)
```

1.3.4 Cursors

After connecting to OmniSci, 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)]
```

1.4 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`.

- lists of tuples are always loaded with *Connection.load_table_rowwise()*
- A `pandas.DataFrame` or `pyarrow.Table` will be loaded using *Connection.load_table_arrow()*
- If upload fails using the arrow method, a `pandas.DataFrame` can be loaded using *Connection.load_table_columnar()*

1.5 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),
 ...
```

1.6 Runtime User-Defined Functions

Connection instance is callable, it can be used as a decorator to Python functions to define these as Runtime UDFs:

```
>>> @con('int32(int32, int32)')
... def totaldelay(dep, arr):
...     return dep + arr
...
>>> query = ("SELECT depdelay, arrdelay, totaldelay(depdelay, arrdelay)"
...          " FROM flights_2008_10k limit 100")
>>> df = con.select_ipc(query)
>>> df.head()
   depdelay  arrdelay  EXPR$2
0         8       -14      -6
1        19         2      21
2         8        14      22
```

(continues on next page)

(continued from previous page)

```
3        -4        -6     -10
4        34        34      68
```

Note: Runtime UDFs can be defined if the OmniSci server has enabled its support (see `--enable-runtime-udf` option of `omnisci_server`) and rbc package is installed. This is still experimental functionality, and currently it does not work on the Windows operating system.

CHAPTER TWO

API REFERENCE

class pymapd.**Connection**(*uri=None*, *user=None*, *password=None*, *host=None*, *port=6274*, *dbname=None*, *protocol='binary'*, *sessionid=None*, *bin_cert_validate=None*, *bin_ca_certs=None*, *idpurl=None*, *idpformusernamefield='username'*, *idpformpasswordfield='password'*, *idpsslverify=True*)

Connect to your OmniSci database.

close()

Disconnect from the database unless created with sessionid

commit()

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

Implemented to comply with the DBI 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_ids: int GPU which contains TDataFrame

duplicate_dashboard(*dashboard_id*, *new_name=None*, *source_remap=None*)

Duplicate an existing dashboard, returning the new dashboard id.

Parameters

dashboard_id: int The id of the dashboard to duplicate

new_name: str The name for the new dashboard

source_remap: dict EXPERIMENTAL A dictionary remapping table names. The old table name(s) should be keys of the dict, with each value being another dict with a 'name' key holding the new table value. This structure can be used later to support changing column names.

Examples

```
>>> source_remap = {'oldtablename1': {'name': 'newtablename1'}, 'oldtablename2
↪': {'name': 'newtablename2'}}
>>> newdash = con.duplicate_dashboard(12345, "new dash", source_remap)
```

execute (*operation*, *parameters=None*)

Execute a SQL statement

Parameters

operation: str A SQL statement to exucute

Returns

c: Cursor

get_dashboard (*dashboard_id*)

Return the dashboard object of a specific dashboard

Examples

```
>>> con.get_dashboard(123)
```

get_dashboards ()

List all the dashboards in the database

Examples

```
>>> con.get_dashboards()
```

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, encoding='DICT'),
 ColumnDetails(name='trans', type='STR', nullable=True, precision=0,
               scale=0, comp_param=32, encoding='DICT'),
 ...
]
```

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', 'arrow'} 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

Notes

Use `pymapd >= 0.11.0` while running with `omnisci >= 4.6.0` in order to avoid loading inconsistent values into DATE column.

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)
```

`register_runtime_udfs`()

Register any bending Runtime UDF functions in OmniSci server.

If no Runtime UDFs have been defined, the call to this method is noop.

`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 the Python code to be executed on the same machine where OmniSci running.

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.

This method requires the Python code to be executed on the same machine where OmniSci running.

class pymapd.**Cursor**(*connection*)

A database cursor.

property arraysize

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

See also:

fetchmany

close()

Close this cursor.

property 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

`pymapd.`**`connect`**(*uri=None*, *user=None*, *password=None*, *host=None*, *port=6274*, *dbname=None*, *protocol='binary'*, *sessionid=None*, *bin_cert_validate=None*, *bin_ca_certs=None*, *idpurl=None*, *idpformusernamefield='username'*, *idpformpasswordfield='password'*, *idpsslverify=True*)

Create a new Connection.

Parameters

uri: str

user: str

password: str

host: str

port: int

dbname: str

protocol: {'binary', 'http', 'https'}

sessionid: str

bin_cert_validate: bool, optional, binary encrypted connection only Whether to continue if there is any certificate error

bin_ca_certs: str, optional, binary encrypted connection only Path to the CA certificate file

idpurl [str] EXPERIMENTAL Enable SAML authentication by providing the logon page of the SAML Identity Provider.

idpformusernamefield: str The HTML form ID for the username, defaults to 'username'.

idpformpasswordfield: str The HTML form ID for the password, defaults to 'password'.

idpsslverify: str Enable / disable certificate checking, defaults to True.

Returns

conn: Connection

Examples

You can either pass a string `uri`, all the individual components, or an existing sessionid excluding user, password, and database

```
>>> connect('mapd://admin:HyperInteractive@localhost:6274/omnisci?'
...         'protocol=binary')
Connection(mapd://mapd:***@localhost:6274/mapd?protocol=binary)
```

```
>>> connect(user='admin', password='HyperInteractive', host='localhost',
...         port=6274, dbname='omnisci')
```

```
>>> connect(user='admin', password='HyperInteractive', host='localhost',
...         port=443, idpurl='https://sso.localhost/logon',
            protocol='https')
```

```
>>> connect(sessionid='XihlkjhdasfsadSDoasdllMweieisdpo', host='localhost',
...         port=6273, protocol='http')
```

2.1 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.`**`DatabaseError`**
Raised when the database encounters an error.

exception `pymapd.exceptions.`**`Error`**
Base class for all pymapd errors.

exception `pymapd.exceptions.`**`IntegrityError`**
Raised when the relational integrity of the database is affected.

exception `pymapd.exceptions.`**`InterfaceError`**
Raised whenever you use pymapd interface incorrectly.

exception `pymapd.exceptions.`**`InternalError`**
Raised for errors internal to the database, e.g. and invalid cursor.

exception `pymapd.exceptions.`**`NotSupportedError`**
Raised when an API not supported by the database is used.

exception `pymapd.exceptions.`**`OperationalError`**
Raised for non-programmer related database errors, e.g. an unexpected disconnect.

exception `pymapd.exceptions.`**`ProgrammingError`**
Raised for programming errors, e.g. syntax errors, table already exists.

CHAPTER

THREE

CONTRIBUTING TO PYMAPD

As an open-source company, OmniSci welcomes contributions to all of its open-source repositories, including pymapd. All discussion and development takes place via the pymapd GitHub repository.

It is suggested, but not required, that you create a GitHub issue before contributing a feature or bug fix. This is so that other developers 1) know that you are working on the feature/issue and 2) that internal OmniSci experts can help you navigate any database-specific logic that may not be obvious within pymapd. All patches should be submitted as pull requests, and upon passing the test suite and review by OmniSci, will be merged to master for release as part of the next package release cycle.

3.1 Development Environment Setup

pymapd is written in plain Python 3 (i.e. no Cython), and as such, doesn't require any specialized development environment outside of installing the dependencies. However, we do suggest creating a new conda development enviornment with the provided conda *environment.yml* file to ensure that your changes work without relying on unspecified system-level Python packages.

Two development environment files are provided: one to provide the packages needed to develop on CPU only, and the other to provide *GPU* development packages. Only one is required, but you may decide to use both in order to run *pytest* against a CPU or GPU environment.

A *pymapd* development environment can be setup with the following:

3.1.1 CPU Environment

3.1.2 GPU Environment

At this point, you have everything you need to develop pymapd. However, to run the test suite, you need to be running an instance of OmniSci on the same machine you are devloping on. OmniSci provides Docker images that work great for this purpose.

3.2 Docker Environment Setup

3.2.1 OmniSci Core CPU-only

Unless you are planning on developing GPU-specific functionality in pymapd, using the CPU image is enough to run the test suite:

```
docker run \
  -d \
  --name omnisci \
  -p 6274:6274 \
  -p 6278:6278 \
  --ipc=host \
  -v /home/<username>/omnisci-storage:/omnisci-storage \
  omnisci/core-os-cpu
```

With the above code, we:

- create/run an instance of OmniSci Core CPU as a daemon (i.e. running in the background until stopped)
- forward ports `6274` (binary connection) and `6278` (http connection).
- set `ipc=host` for testing shared memory/IPC functionality
- point to a local directory to store data loaded to OmniSci. This allows our container to be ephemeral.

To run the test suite, call `pytest` from the top-level pymapd folder:

```
(pymapd_dev) laptop:~/github_work/pymapd$ pytest
```

`pytest` will run through the test suite, running the tests against the Docker container. Because we are using CPU-only, the test suite skips the GPU tests, and you can expect to see the following messages at the end of the test suite run:

```
=============================================== short test summary info␣
↪================================================
SKIPPED [4] tests/test_data_no_nulls_gpu.py:15: No GPU available
SKIPPED [1] tests/test_deallocate.py:34: No GPU available
SKIPPED [1] tests/test_deallocate.py:54: deallocate non-functional in recent distros
SKIPPED [1] tests/test_deallocate.py:67: No GPU available
SKIPPED [1] tests/test_deallocate.py:80: deallocate non-functional in recent distros
SKIPPED [1] tests/test_deallocate.py:92: No GPU available
SKIPPED [1] tests/test_deallocate.py:105: deallocate non-functional in recent distros
SKIPPED [2] tests/test_integration.py:207: No GPU available
SKIPPED [1] tests/test_integration.py:238: No GPU available
================================== 69 passed, 13 skipped, 1 warnings in 19.40 seconds␣
↪==================================
```

3.2.2 OmniSci Core GPU-enabled

To run the pymapd test suite with the GPU tests, the workflow is pretty much the same as CPU-only, except with the OmniSci Core GPU-enabled container:

```
docker run \
  --runtime=nvidia \
  -d \
  --name omnisci \
  -p 6274:6274 \
  -p 6278:6278 \
  --ipc=host \
  -v /home/<username>/omnisci-storage:/omnisci-storage \
  omnisci/core-os-cuda
```

You also need to install cudf in your development environment. Because cudf is in active development, and requires attention to the specific version of CUDA installed, we recommend checking the cudf documentation to get the most up-to-date installation instructions.

3.3 Updating Apache Thrift Bindings

When the upstream mapd-core project updates its Apache Thrift definition file, the bindings shipped with `pymapd` need to be regenerated. Note that the *omniscidb* repository must be cloned locally.

3.4 Updating the Documentation

The documentation for pymapd is generated by ReadTheDocs on each commit. Some pages (such as this one) are manually created, others such as the API Reference is generated by the docstrings from each method.

If you are planning on making non-trival changes to the documentation and want to preview the result before making a commit, you need to install sphinx and sphinx-rtd-theme into your development environment:

```
pip install sphinx sphinx-rtd-theme
```

Once you have sphinx installed, to build the documentation switch to the `pymapd/docs` directory and run `make html`. This will update the documentation in the `pymapd/docs/build/html` directory. From that directory, running `python -m http.server` will allow you to preview the site on `localhost:8000` in the browser. Run `make html` each time you save a file to see the file changes in the documentation.

3.5 Publishing a new package version

pymapd doesn't currently follow a rigid release schedule; rather, when enough functionality is deemed to be "enough" for a new version to be released, or a sufficiently serious bug/issue is fixed, we will release a new version. pymapd is distributed via PyPI and conda-forge.

Prior to submitting to PyPI and/or conda-forge, create a new release tag on GitHub (with notes), then run `git pull` to bring this tag to your local pymapd repository folder.

3.5.1 PyPI

To publish to PyPI, we use the twine package via the CLI. twine only allows for submitting to PyPI by registered users (currently, internal OmniSci employees):

```
conda install twine
python setup.py sdist
twine upload dist/*
```

Publishing a package to PyPI is near instantaneous after runnning `twine upload dist/*`. Before running `twine upload`, be sure the `dist` directory only has the current version of the package you are intending to upload.

3.5.2 conda-forge

The release process for conda-forge is triggered via creating a new version number on the pymapd GitHub repository. Given the volume of packages released on conda-forge, it can take several hours for the bot to open a PR on pymapd-feedstock. There is nothing that needs to be done to speed this up, just be patient.

When the conda-forge bot opens a PR on the pymapd-feedstock repo, one of the feedstock maintainers needs to validate the correctness of the PR, check the accuracy of the package versions on the meta.yaml recipe file, and then merge once the CI tests pass.

CHAPTER

FOUR

RELEASE NOTES

The release notes for pymapd are managed on the GitHub repository in the Releases tab. Since pymapd releases try to track new features in the main OmniSci Core project, it's highly recommended that you check the Releases tab any time you install a new version of pymapd or upgrade OmniSci so that you understand any breaking changes that may have been made during a new pymapd release.

Some notable breaking changes include:

Release	Breaking Change
0.17	Added preliminary support for Runtime User-Defined Functions
0.15	Support for binary TLS Thrift connections
0.14	Updated Thrift bindings to 4.8
0.13	Changed context manager to return `self`, not next level down
0.12	Updated Thrift to 4.6.1 bindings
0.11	Dropped Python 3.5 support
0.11	Modified `load_table_columnar` to support OmniSci 4.6 backend Dates change
0.10	`Int8` when using Arrow for data upload no longer mutates input DataFrame
0.10	`load_table*` methods set string columns to `TEXT ENCODING DICT(32)`
0.9	Removed ability to specify `columnar` keyword on `Cursor`
0.9	Lower bounds for pandas, numpy, sqlalchemy and pytest increased
0.8	Default ports changed in connect statement from 9092 to 6274
0.8	Python 2 support dropped
0.7	Support for Python 3.4 dropped, support for Python 3.7 added
0.7	First release supporting cudf (removing option to use pygdf)
0.6	NumPy, pyarrow and pandas now hard dependencies instead of optional

CHAPTER

FIVE

FAQ AND KNOWN LIMITATIONS

This page contains information that doesn't fit into other pages or is important enough to be called out separately. If you have a question or tidbit of information that you feel should be included here, please create an issue and/or pull request to get it added to this page.

Note: While we strive to keep this page updated, bugfixes and new features are being added regularly. If information on this page conflicts with your experience, please open an issue or drop by our Community forum to get clarification.

5.1 FAQ

Q Why do `select_ipc()` and `select_ipc_gpu()` give me errors, but `execute()` works fine?

A Both `select_ipc()` and `select_ipc_gpu()` require running the pymapd code on the same machine where OmniSci is running. This also implies that these two methods will not work on Windows machines, just Linux (CPU and GPU) and OSX (CPU-only).

Q Why do geospatial data get uploaded as `TEXT ENCODED DICT(32)`?

A When using `load_table` with `create=True` or `create='infer'`, data where type cannot be easily inferred will default to `TEXT ENCODED DICT(32)`. To solve this issue, create the table definition before loading the data.

5.2 Helpful Hints

- **Convert your timestamps to UTC** OmniSci stores timestamps as UTC. When loading data to OmniSci, plain Python `datetime` objects are assumed to be UTC. If the `datetime` object has localization, only `datetime64[ns, UTC]` is supported.
- **When loading data, hand-create table schema if performance is critical** While the `load_table()` does provide a keyword argument `create` to auto-create the table before attempting to load to OmniSci, this functionality is for *convenience purposes only*. The user is in a much better position to know the exact data types of the input data than the heuristics used by pymapd.

  Additionally, pymapd does not attempt to use the smallest possible column width to represent your data. For example, significant reductions in disk storage and a larger amount of 'hot data' can be realized if your data fits in a `TINYINT` column vs storing it as an `INTEGER`.

5.3 Known Limitations

- **OmniSci `BIGINT` is 64-bit** Be careful using pymapd on 32-bit systems, as we do not check for integer overflow when returning a query.
- **`DECIMAL` types returned as Python `float`** OmniSci stores and performs `DECIMAL` calculations within the database at the column-definition level of precision. However, the results are currently returned back to Python as float. We are evaluating how to change this behavior, so that exact decimal representations is consistent on the server and in Python.

PYTHON MODULE INDEX

INDEX