turbodbc Documentation

Release master

Michael König

Feb 14, 2022

Contents

1 Introduction 3

2 Getting started 7

3 Advanced usage 13

4 ODBC configuration 21

5 Databases configuration and performance 25

6 Version history / changelog 33

7 Troubleshooting 43

8 Frequently asked questions 47

9 Contributing 49

10 API reference 53

11 Indices and tables 59

Index 61

Turbodbc is a Python module to access relational databases via the Open Database Connectivity (ODBC) interface. Its primary target audience are data scientist that use databases for which no efficient native Python drivers are available.

For maximum compatibility, turbodbc complies with the Python Database API Specification 2.0 (PEP 249). For maximum performance, turbodbc offers built-in NumPy and Apache Arrow support and internally relies on batched data transfer instead of single-record communication as other popular ODBC modules do.

Turbodbc is free to use (MIT license), open source (GitHub), works with Python 3.6+, and is available for Linux, macOS, and Windows.

Turbodbc is routinely tested with MySQL, PostgreSQL, EXASOL, and MSSQL, but probably also works with your database.

CHAPTER 1

Introduction

Turbodbc is a Python module to access relational databases via the Open Database Connectivity (ODBC) interface. In addition to complying with the Python Database API Specification 2.0, turbodbc offers built-in NumPy support. Don't wait minutes for your results, just blink.

1.1 Features

- Bulk retrieval of result sets
- Built-in NumPy support
- Built-in Apache Arrow support
- Bulk transfer of query parameters
- Asynchronous I/O for result sets
- Automatic conversion of decimal type to integer, float, and string as appropriate
- Supported data types for both result sets and parameters: `int`, `float`, `str`, `bool`, `datetime.date`, `datetime.datetime`
- Also provides a high-level C++11 database driver under the hood
- Tested with Python 3.8, 3.9 and 3.10
- Tested on 64 bit versions of Linux, OSX, and Windows (Python 3.8+).

1.2 Why should I use turbodbc instead of other ODBC modules?

Short answer: turbodbc is faster.

Slightly longer answer: turbodbc is faster, *much* faster if you want to work with NumPy.

Medium-length answer: I have tested turbodbc and pyodbc (probably the most popular Python ODBC module) with various databases (Exasol, PostgreSQL, MySQL) and corresponding ODBC drivers. I found turbodbc to be consistently faster.

For retrieving result sets, I found speedups between 1.5 and 7 retrieving plain Python objects. For inserting data, I found speedups of up to 100.

Is this completely scientific? Not at all. I have not told you about which hardware I used, which operating systems, drivers, database versions, network bandwidth, database layout, SQL queries, what is measured, and how I performed was measured.

All I can tell you is that if I exchange pyodbc with turbodbc, my benchmarks took less time, often approaching one (reading) or two (writing) orders of magnitude. Give it a spin for yourself, and tell me if you liked it.

1.3 Smooth. What is the trick?

Turbodbc exploits buffering.

- Turbodbc implements both sending parameters and retrieving result sets using buffers of multiple rows/parameter sets. This avoids round trips to the ODBC driver and (depending how well the ODBC driver is written) to the database.
- Multiple buffers are used for asynchronous I/O. This allows to interleave Python object conversion and direct database interaction (see performance options below).
- Buffers contain binary representations of data. NumPy arrays contain binary representations of data. Good thing they are often the same, so instead of converting we can just copy data.

1.4 Supported environments

- 64 bit operating systems (32 bit not supported)
- Linux (successfully built on Ubuntu 12, Ubuntu 14, Debian 7, Debian 8)
- OSX (successfully built on Sierra a.k.a. 10.12 and El Capitan a.k.a. 10.11)
- Windows (successfully built on Windows 10)
- Python 3.7, 3.8, 3.9
- More environments probably work as well, but these are the versions that are regularly tested on Travis or local development machines.

1.5 Supported databases

Turbodbc uses suites of unit and integration tests to ensure quality. Every time turbodbc's code is updated on GitHub, turbodbc is automatically built from scratch and tested with the following databases:

- PostgreSQL (Linux, OSX, Windows)
- MySQL (Linux, OSX, Windows)
- MSSQL (Windows, with official MS driver)

During development, turbodbc is tested with the following database:

- Exasol (Linux, OSX)

Releases will not be made if any (implemented) test fails for any of the databases listed above. The following databases/driver combinations are tested on an irregular basis:

- MSSQL with FreeTDS (Linux, OSX)
- MSSQL with Microsoft's official ODBC driver (Linux)

There is a good chance that turbodbc will work with other, totally untested databases as well. There is, however, an equally good chance that you will encounter compatibility issues. If you encounter one, please take the time to report it so turbodbc can be improved to work with more real-world databases. Thanks!

CHAPTER 2

Getting started

2.1 Installation

2.1.1 Conda

We generally recommend using conda to install *turbodbc*. This will fetch the pre-compiled binaries and all dependencies for your platform:

```
conda install -c conda-forge turbodbc
```

2.1.2 Pip

Linux and OSX

To install turbodbc on Linux and OSX using pip, use the following command:

```
pip install turbodbc
```

This will trigger a source build that requires compiling C++ code. Please make sure the following prerequisites are met:

Requirement	Linux (`apt-get install`)	OSX (`brew install`)
C++11 compiler	G++-4.8 or higher	clang with OSX 10.9+
Boost library + headers (1)	`libboost-all-dev`	`boost`
ODBC library + headers	`unixodbc-dev`	`unixodbc`
Python headers	`python-dev`	use `pyenv` to install

Please `pip install numpy` before installing turbodbc, because turbodbc will search for the `numpy` Python package at installation/compile time. If NumPy is not installed, turbodbc will not compile the *optional NumPy support*

features. Similarly, please `pip install pyarrow` before installing turbodbc if you would like to use the *optional Apache Arrow support*.

(1) The minimum viable Boost setup requires the libraries `variant`, `optional`, `datetime`, and `locale`.

Windows

To install turbodbc on Windows, please use the following command:

```
pip install turbodbc
```

This will download and install a binary wheel, no compilation required. You still need to meet the following prerequisites, though:

Requirement	Windows
OS Bitness	64-bit
Python	3.5 or 3.6, 64-bit
Runtime	MSVS 2015 Update 3 Redistributable, 64 bit

If you wish to install turbodbc with a later (64-bit) version of Python, you must manually install the Boost C++ libraries first:

1. The Boost libraries must be compiled, hence if you don't have a suitable C++ compiler installed already, download the "Build Tools for Visual Studio 2019" from Microsoft Visual Studio, and install the "C++ build tools" Workload.
2. Download Boost from https://www.boost.org/ (click on the "Current Release" version link, e.g. "Version 1.72.0", then download the Windows zip file).
3. Unzip the zipfile somewhere on your PC, e.g. the Downloads folder.
4. In an "x64 Native Tools Command Prompt for VS 2019" command prompt (or equivalent), navigate to the unzipped Boost top-level directory.
5. Run `.\bootstrap.bat` (this generates the `b2` executable).
6. Run `.\b2` (this generates the `stage` directory and contents, takes a few minutes to run).

At this point, Boost is available but you must set the BOOST_ROOT environment variable before installing turbodbc. Set BOOST_ROOT to the directory that contains the `bootstrap.bat` file referenced above, e.g.:

```
set BOOST_ROOT=C:\your path to\boost_1_72_0
pip install turbodbc
```

If you require NumPy support, please

```
pip install numpy
```

sometime after installing turbodbc. Apache Arrow support is not yet available on Windows.

2.2 Basic usage

Turbodbc follows the specification of the Python database API v2 (PEP 249). Here is a short summary, including the parts not specified by the PEP.

2.2.1 Establish a connection with your database

All ODBC appications, including turbodbc, use connection strings to establish connections with a database. If you know how the connection string for your database looks like, use the following lines to establish a connection:

```
>>> from turbodbc import connect
>>> connection = connect(connection_string='Driver={PostgreSQL};Server=IP address;
↪Port=5432;Database=myDataBase;Uid=myUsername;Pwd=myPassword;')
```

If you do not specify the `connection_string` keyword argument, turbodbc will create a connection string based on the keyword arguments you pass to `connect`:

```
>>> from turbodbc import connect
>>> connection = connect(dsn='My data source name as defined by your ODBC␣
↪configuration')
```

The `dsn` is the data source name of your connection. Data source names uniquely identify connection settings that shall be used to connect with a database. Data source names are part of your *ODBC configuration* and you need to set them up yourself. Once set up, however, all ODBC applications can use the same data source name to refer to the same set of connection options, typically including the host, database, driver settings, and sometimes even credentials. If your ODBC environment is set up properly, just using the `dsn` option should be sufficient.

You can add extra options besides the `dsn` to overwrite or add settings:

```
>>> from turbodbc import connect
>>> connection = connect(dsn='my dsn', user='my user has precedence')
>>> connection = connect(dsn='my dsn', username='field names depend on the driver')
```

Last but not least, you can also do without a `dsn` and just specify all required configuration options directly:

```
>>> from turbodbc import connect
>>> connection = connect(driver="PostgreSQL",
...                      server="hostname",
...                      port="5432",
...                      database="myDataBase",
...                      uid="myUsername",
...                      pwd="myPassword")
```

2.2.2 Executing SQL queries and retrieving results

To execute a query, you need to create a `cursor` object first:

```
>>> cursor = connection.cursor()
```

This cursor object lets you execute SQL commands and queries. Here is how to execute a `SELECT` query:

```
>>> cursor.execute('SELECT 42')
```

You have multiple options to retrieve the generated result set. For example, you can iterate over the cursor:

```
>>> for row in cursor:
...     print row
[42L]
```

Alternatively, you can fetch all results as a list of rows:

```
>>> cursor.fetchall()
[[42L]]
```

You can also retrieve result sets as NumPy arrays or Apache Arrow tables, see *Advanced usage*.

2.2.3 Executing manipulating SQL queries

As before, you need to create a `cursor` object first:

```
>>> cursor = connection.cursor()
```

You can now execute a basic `INSERT` query:

```
>>> cursor.execute("INSERT INTO TABLE my_integer_table VALUES (42, 17)")
```

This will insert two values, `42` and `17`, in a single row of table `my_integer_table`. Inserting values like this is impractical, because it requires putting the values into the actual SQL string.

To avoid this, you can pass parameters to `execute()`:

```
>>> cursor.execute("INSERT INTO TABLE my_integer_table VALUES (?, ?)",
...                [42, 17])
```

Please note the question marks `?` in the SQL string that marks two parameters. Adding single rows at a time is not efficient. You can add more than just a single row to a table efficiently by using `executemany()`:

```
>>> parameter_sets = [[42, 17],
...                   [23, 19],
...                   [314, 271]]
>>> cursor.executemany("INSERT INTO TABLE my_integer_table VALUES (?, ?)",
...                    parameter_sets)
```

If you already have parameters stored as NumPy arrays, check the *Using NumPy arrays as query parameters* section to use them even more efficiently.

2.2.4 Transactions

By default, turbodbc does not enable automatic commits (`autocommit`). To commit your changes to the database, please use the following command:

```
>>> connection.commit()
```

If you want to roll back your changes, use the following command:

```
>>> connection.rollback()
```

If you prefer `autocommit` for your workflow or your database does not support transactions at all, you can use the *autocommit* option.

2.3 Supported data types

Turbodbc supports the most common data types data scientists are interested in. The following table shows which database types are converted to which Python types:

Database type(s)	Python type
Integers, `DECIMAL(<19,0)`	`int`
`DOUBLE`, `DECIMAL(<19, >0)`	`float`
`DOUBLE`, `DECIMAL(>18, 0)`	`unicode` (`str`) or `int` *
`DOUBLE`, `DECIMAL(>18, >0)`	`unicode` (`str`) or `float` *
`BIT`, boolean-like	`bool`
`TIMESTAMP`, `DATETIME`	`datetime.datetime`
`DATE`	`datetime.date`
`VARCHAR`, strings	`unicode` (`str`)
`TIME`	(not supported)
`UUID`, `GUID`	(not supported)

*) The conversion depends on turbodbc's `large_decimals_as_64_bit_types` *option*.

When using parameters with `execute()` and `executemany()`, the table is basically reversed. The first type in the "database type(s)" column denotes the type used to transfer back data. For integers, 64-bit integers are transferred. For strings, the length of the transferred `VARCHAR` depends on the length of the transferred strings.

2.4 What to read next

Continue with the *advanced usage* section. Besides general *tuning parameters* it also discusses how to leverage *NumPy* or *Apache Arrow* for even better performance.

CHAPTER 3

Advanced usage

3.1 Performance, compatibility, and behavior options

Turbodbc offers a way to adjust its behavior to tune performance and to achieve compatibility with your database. The basic usage is this:

```
>>> from turbodbc import connect, make_options
>>> options = make_options()
>>> connect(dsn="my_dsn", turbodbc_options=options)
```

This will connect with your database using the default options. To use non-default options, supply keyword arguments to `make_options()`:

```
>>> from turbodbc import Megabytes
>>> options = make_options(read_buffer_size=Megabytes(100),
...                        parameter_sets_to_buffer=1000,
...                        varchar_max_character_limit=10000,
...                        use_async_io=True,
...                        prefer_unicode=True,
...                        autocommit=True,
...                        large_decimals_as_64_bit_types=True,
...                        limit_varchar_results_to_max=True)
```

3.1.1 Read buffer size

`read_buffer_size` affects how many result set rows are retrieved per batch of results. Set the attribute to `turbodbc.Megabytes(42)` to have turbodbc determine the optimal number of rows per batch so that the total buffer amounts to 42 MB. This is recommended for most users and databases. You can also set the attribute to `turbodbc.Rows(13)` if you would like to fetch results in batches of 13 rows. By default, turbodbc fetches results in batches of 20 MB.

Please note that sometimes a single row of a result set may exceed the specified buffer size. This can happen if large fields such as `VARCHAR(8000000)` or `TEXT` are part of the result set. In this case, results are fetched in batches of

single rows that exceed the specified size. Buffer sizes for large text fields can be controlled with the *VARCHAR(max) character limit* and XXX options.

3.1.2 Buffered parameter sets

Similarly, `parameter_sets_to_buffer` changes the number of parameter sets which are transferred per batch of parameters (e.g., as sent with `executemany()`). Please note that it is not (yet) possible to use the *Megabytes* and *Rows* classes here.

3.1.3 VARCHAR(max) character limit

The `varchar_max_character_limit` specifies the buffer size for result set columns of types `VARCHAR(max)`, `NVARCHAR(max)`, or similar types your database supports. Small values increase the chance of truncation, large ones require more memory. Depending on your setting of `read_buffer_size`, this may increase the total memory consumption or reduce the number of rows fetched per batch, thus affecting performance. The default value is `65535` characters.

Note: This value does not affect fields of type `VARCHAR(n)` with `n > 0`, unless the option *Limit VARCHAR results to MAX* is set. Also, this option does not affect parameters that you may pass to the database.

3.1.4 Asynchronous input/output

If you set `use_async_io` to `True`, turbodbc will use asynchronous I/O operations (limited to result sets for the time being). Asynchronous I/O means that while the main thread converts result set rows retrieved from the database to Python objects, another thread fetches a new batch of results from the database in the background. This may yield a speedup of `2` if retrieving and converting are similarly fast operations.

Note: Asynchronous I/O is experimental and has to fully prove itself yet. Do not be afraid to give it a try, though.

3.1.5 Prefer unicode

Set `prefer_unicode` to `True` if your database does not fully support the UTF-8 encoding turbodbc prefers. With this option you can tell turbodbc to use two-byte character strings with UCS-2/UTF-16 encoding. Use this option if you wish to connect to Microsoft SQL Server (MSSQL).

3.1.6 Autocommit

Set `autocommit` to `True` if you want the database to `COMMIT` your changes automatically after each query or command. By default, `autocommit` is disabled and users are required to call `connection.commit()` to persist their changes.

Note: Some databases that do not support transactions may even require this option to be set to `True` in order to establish a connection at all.

3.1.7 Large decimals as 64 bit types

Set `large_decimals_as_64_bit_types` to `True` if you want to retrieve `Decimal` and `Numeric` types with more than `18` digits as 64 bit `integer` and `float` numbers. The default is to retrieve such fields as strings instead.

Please note that this option may lead to overflows or loss of precision. If, however, your data type is much larger than the data it is supposed to hold, this option is very useful to obtain numeric Python objects and *NumPy arrays*.

3.1.8 Limit VARCHAR results to MAX

Set `limit_varchar_results_to_max` to `True` if you want to limit *all* string-like fields (`VARCHAR(n)`, `NVARCHAR(n)`, etc. with `n > 0`) in result sets to a maximum of *VARCHAR(max) character limit* characters.

Please note that enabling this option can lead to truncation of string-like data when retrieving results. Parameters sent to the database are not affected by this option.

If not set or set to `False`, string-like result fields with a specific size will *always* be retrieved with a sufficiently large buffer so that no truncation occurs. String-like fields of indeterminate size (`VARCHAR(max)`, `TEXT`, etc. on some databases) are still subject to *VARCHAR(max) character limit*.

3.1.9 Extra capacity for unicode strings

Set `force_extra_capacity_for_unicode` to `True` if you find that strings retrieved from `VARCHAR(n)` or `NVARCHAR(n)` fields are being truncated. Some ODBC drivers report the length of the field and setting this option changes the way turbodbc allocates memory, so that retrieving these strings are not truncated. If `limit_varchar_results_to_max` is `True`, memory is allocated as if `n` is *VARCHAR(max) character limit*.

Please note that enabling this option leads to increased memory usage when retrieving string fields in result sets. Parameters sent to the database are not affected by this option.

3.1.10 Decoding wide character types as narrow types

Set `fetch_wchar_as_char` to `True` if you find that strings retrieved from `NVARCHAR(n)` fields are being corrupted. Some ODBC drivers place single byte encodings into `SQL_WCHAR` type strings and as a consequence are corrupted upon retrieval by turbodbc. Setting this option forces turbodbc to decode `SQL_WCHAR` as single byte encodings.

3.2 Controlling autocommit behavior at runtime

You can enable and disable autocommit mode after you have established a connection, and you can also check whether autocommit is currently enabled:

```
>>> from turbodbc import connect
>>> connection = connect(dsn="my DSN")
>>> connection.autocommit = True

[... more things happening ...]

>>> if not connection.autocommit:
...     connection.commit()
```

3.3 NumPy support

Note: Turbodbc's NumPy support requires the `numpy` package to be installed. For all source builds, Numpy needs to be installed before installing turbodbc. Please check the *installation instructions* for more details.

3.3.1 Obtaining NumPy result sets all at once

Here is how to use turbodbc to retrieve the full result set in the form of NumPy masked arrays:

```
>>> cursor.execute("SELECT A, B FROM my_table")
>>> cursor.fetchallnumpy()
OrderedDict([('A', masked_array(data = [42 --],
                                mask = [False True],
                                fill_value = 999999)),
             ('B', masked_array(data = [3.14 2.71],
                                mask = [False False],
                                fill_value = 1e+20))])
```

3.3.2 Obtaining NumPy result sets in batches

You can also fetch NumPy result sets in batches using an iterable:

```
>>> cursor.execute("SELECT A, B FROM my_table")
>>> batches = cursor.fetchnumpybatches()
>>> for batch in batches:
...     print(batch)
OrderedDict([('A', masked_array(data = [42 --],
                                mask = [False True],
                                fill_value = 999999)),
             ('B', masked_array(data = [3.14 2.71],
                                mask = [False False],
                                fill_value = 1e+20))])
```

The size of the batches depends on the `read_buffer_size` attribute set in the *performance options*.

3.3.3 Notes regarding NumPy result sets

- NumPy results are returned as an `OrderedDict` of column name/value pairs. The column order is the same as in your query.
- The column values are of type `MaskedArray`. Any `NULL` values you have in your database will show up as masked entries (`NULL` values in string-like columns will show up as `None` objects).

The following table shows how the most common data types data scientists are interested in are converted to NumPy columns:

Database type(s)	Python type
Integers, `DECIMAL(<19,0)`	`int64`
`DOUBLE`, `DECIMAL(<19, >0)`	`float64`
`DECIMAL(>18, 0)`	`object_` or `int64` *
`DECIMAL(>18, >0)`	`object_` or `float64` *
`BIT`, boolean-like	`bool_`
`TIMESTAMP`, `TIME`	`datetime64[µs]`
`DATE`	`datetime64[D]`
`VARCHAR`, strings	`object_`

*) The conversion depends on turbodbc's `large_decimals_as_64_bit_types` *option*.

3.3.4 Using NumPy arrays as query parameters

Here is how to use turbodbc to use values stored in NumPy arrays as query parameters with `executemanycolumns()`:

```
>>> from numpy import array
>>> from numpy.ma import MaskedArray
>>> normal_param = array([1, 2, 3], dtype='int64')
>>> masked_param = MaskedArray([3.14, 1.23, 4.56],
...                            mask=[False, True, False],
...                            dtype='float64')

>>> cursor.executemanycolumns("INSERT INTO my_table VALUES (?, ?)",
...                           [normal_param, masked_param])
# functionally equivalent, but much faster than:
# cursor.execute("INSERT INTO my_table VALUES (1, 3.14)")
# cursor.execute("INSERT INTO my_table VALUES (2, NULL)")
# cursor.execute("INSERT INTO my_table VALUES (3, 4.56)")

>>> cursor.execute("SELECT * FROM my_table").fetchall()
[[1L, 3.14], [2L, None], [3L, 4.56]]
```

- Columns must either be of type `MaskedArray` or `ndarray`.
- Each column must contain one-dimensional, contiguous data.
- All columns must have equal size.
- The `dtype` of each column must be supported, see the table below.
- Use `MaskedArray` and set the `mask` to `True` for individual elements to use `None` values.
- Data is transferred in batches (see *Buffered parameter sets*)

Supported NumPy type	Transferred as
`int64`	`BIGINT` (64 bits)
`float64`	`DOUBLE PRECISION` (64 bits)
`bool_`	`BIT`
`datetime64[µs]`	`TIMESTAMP`
`datetime64[ns]`	`TIMESTAMP`
`datetime64[D]`	`DATE`
`object_` (only `str`, `unicode`, and `None` objects supported)	`VARCHAR` (automatic sizing)

3.4 Apache Arrow support

Note: Turbodbc's Apache Arrow support requires the `pyarrow` package to be installed. For all source builds, Apache Arrow needs to be installed before installing turbodbc. Please check the *installation instructions* for more details.

Apache Arrow is a high-performance data layer that is built for cross-system columnar in-memory analytics using a data model designed to make the most of the CPU cache and vector operations.

Note: Apache Arrow support in turbodbc is still experimental and may not be as efficient as possible yet. Also, Apache Arrow support is not yet available for Windows and has some issues with Unicode fields. Stay tuned for upcoming improvements.

3.4.1 Obtaining Apache Arrow result sets

Here is how to use turbodbc to retrieve the full result set in the form of an Apache Arrow table:

```
>>> cursor.execute("SELECT A, B FROM my_table")
>>> table = cursor.fetchallarrow()
>>> table
pyarrow.Table
A: int64
B: string
>>> table[0].to_pylist()
[42]
>>> table[1].to_pylist()
[u'hello']
```

Looking at the data like this is not particularly useful. However, there is some really useful stuff you can do with an Apache Arrow table, for example, convert it to a Pandas dataframe like this:

```
>>> table.to_pandas()
    A      B
0  42  hello
```

As a performance optimisation for string columns, you can specify the parameter `strings_as_dictionary`. This will retrieve all string columns as Arrow `DictionaryArray`. The data will here be split into two arrays, one that stores all unique string values and one integer array that stores for each row the index in the dictionary. On conversion to Pandas, these columns will be turned into `pandas.Categorical`.

```
>>> cursor.execute("SELECT a, b FROM my_other_table")
>>> table = cursor.fetchallarrow(strings_as_dictionary=True)
>>> table
pyarrow.Table
a: int64
b: dictionary<values=binary, indices=int8, ordered=0>
  dictionary: [61, 62]
>>> table.to_pandas()
   a  b
0  1  a
1  2  b
```

(continues on next page)

(continued from previous page)

```
2  3  b
>>> table.to_pandas().info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 2 columns):
a    3 non-null int64
b    3 non-null category
dtypes: category(1), int64(1)
memory usage: 147.0 bytes
```

To further reduce the memory usage of the returned results, the Arrow based interface can return the integer columns as the minimal possible integer storage type. This type can be different from the integer type used and returned by the database. This mode can be activated by setting `adaptive_integers=True`.

```
>>> # Standard result retrieval
>>> cursor.execute("SELECT * FROM (VALUES(1), (2), (3))")
>>> table = cursor.fetchallarrow()
>>> table
pyarrow.Table
__COL0__: int64
>>> table.to_pandas()
   __COL0__
0         1
1         3
2         2
>>> table.to_pandas().info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 1 columns):
__COL0__    3 non-null int64
dtypes: int64(1)
memory usage: 96.0 bytes

>>> # With adaptive integer storage
>>> cursor.execute("SELECT * FROM (VALUES(1), (2), (3))")
>>> table = cursor.fetchallarrow(adaptive_integers=True)
>>> table
pyarrow.Table
__COL0__: int8
>>> table.to_pandas()
   __COL0__
0         1
1         3
2         2
>>> table.to_pandas().info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 1 columns):
__COL0__    3 non-null int8
dtypes: int8(1)
memory usage: 75.0 bytes
```

3.4.2 Obtaining Apache Arrow result sets in batches

Similar to the numpy support for fetching results as a batch, you can fetch a query result as an iterator of pyarrow tables.

```
>>> cursor.execute("SELECT A, B FROM my_table")
>>> batches = cursor.fetcharrowbatches()
>>> for batch in batches:
...     print(batch)
pyarrow.Table
```

The size of the batches depends on the `read_buffer_size` attribute set in the *performance options*.

3.4.3 Using Apache Arrow tables as query parameters

Here is how to use turbodbc to use values stored in Apache Arrow tables as query parameters with `executemanycolumns()`:

```
>>> import numpy as np
>>> import pyarrow as pa
>>> normal_param = pa.array([1, 2, 3], type=pa.int64())
>>> masked_param = pa.Array.from_pandas(np.array([3.14, 1.23, 4.56])
...                             mask=np.array([False, True, False])
...                             type=pa.float64())
>>> table = pa.Table.from_arrays([normal_param, masked_param], ['a', 'b'])

>>> cursor.executemanycolumns("INSERT INTO my_table VALUES (?, ?)",
...                           table)
# functionally equivalent, but much faster than:
# cursor.execute("INSERT INTO my_table VALUES (1, 3.14)")
# cursor.execute("INSERT INTO my_table VALUES (2, NULL)")
# cursor.execute("INSERT INTO my_table VALUES (3, 4.56)")

>>> cursor.execute("SELECT * FROM my_table").fetchall()
[[1L, 3.14], [2L, None], [3L, 4.56]]
```

- Tables must be of type `pyarrow.Table`.
- Each column must contain one-dimensional, contiguous data. There is no support for chunked arrays yet.
- All columns must have equal size.
- The `dtype` of each column must be supported, see the table below.
- Data is transferred in batches (see *Buffered parameter sets*)

Supported Apache Arrow type	Transferred as
`INT8`, `UINT8`, `INT16`, `UINT16`, `INT32`, `UINT32`, `INT64`	`BIGINT` (64 bits)
`DOUBLE`	`DOUBLE PRECISION` (64 bits)
`BOOL`	`BIT`
`TIMESTAMP[µs]`	`TIMESTAMP`
`TIMESTAMP[ns]`	`TIMESTAMP`
`DATE32`	`DATE`
`BINARY`	`VARCHAR` (automatic sizing)
`STRING`	`VARCHAR` (automatic sizing)

CHAPTER 4

ODBC configuration

ODBC configuration can be a real pain, in particular if you are new to ODBC. So here is a short primer of what ODBC is about.

4.1 ODBC basics

ODBC is the abbreviation for open database connectivity, a standard for interacting with relational databases that has been considerably influenced by Microsoft. The aim of the standard is that applications can work with multiple databases with little to no adjustments in code.

This is made possible by combining three components with each other:

- Database vendors supply *ODBC drivers*.
- An ODBC *driver manager* manages ODBC data sources.
- *Applications* use the ODBC driver manager to connect to data sources.

Turbodbc makes it easy to build applications that use the ODBC driver manager, but it still requires the driver manager to be configured correctly so that your databases are found.

4.2 ODBC concepts

4.2.1 ODBC drivers

ODBC drivers comply with the ODBC API, meaning that they offer a set of about 80 C functions with well-defined behavior that internally use database-specific commands to achieve the desired behavior. There is some wiggle room that allows ODBC drivers to implement certain things differently or even exclude support for some advanced usage patterns. But in essence, all ODBC drivers are born more or less equal.

ODBC drivers are easy to come by. Major database vendors offer ODBC drivers as free downloads (Microsoft SQL Server, Exasol, Teradata, etc). Open source databases provide ODBC databases as part of their projects (PostgreSQL,

Impala, MongoDB). Many ODBC drivers are also shipped with Linux distributions or are readily available via Homebrew for OSX. Last but not least, commercial ODBC drivers are available at Progress or easysoft, claiming better performance than their freely available counterparts.

4.2.2 ODBC driver manager

The driver manager is a somewhat odd centerpiece. It is a library that can be used just like any ODBC driver. It provides definitions for various data types, and actual ODBC drivers often rely on these definitions for compilation. The driver manager has a built-in configuration of data sources. A data source has a name (the data source name or DSN), is associated with an ODBC driver, contains configuration options such as the database host or the connection locale, and sometimes it also contains credentials for authentication with the database. Finally, the driver manager typically comes with a tool to edit data sources.

Driver managers are less numerous, but still easily available on all major platforms. Windows comes with a preinstalled ODBC database manager. On Linux and OSX, there are competing driver managers in unixodbc and iodbc.

Note: Turbodbc is tested with Windows's built-in driver manager and unixodbc on Linux and OSX.

4.2.3 ODBC applications

Applications finally use the ODBC API and link to the driver manager. Any time they open a connection, they need to specify the data source name that contains connection attributes that relate to the desired database. Alternatively, they can specify all necessary connection options directly.

Linking to the driver manager instead of the ODBC driver directly means that changing to another driver is as simple as exchanging the connection string at runtime instead of tediously linking to a new driver. Linking to the driver manager also means that the driver manager handles many capability and compatibility options by transparently using alternative functions and workarounds as required.

4.3 Driver manager configuration

The driver manager needs to know to which databases to connect with which ODBC drivers. This configuration needs to be maintained by the user.

4.3.1 Windows

Windows comes with a preinstalled driver manager that can be configured with the ODBC data source administrator. Please see Microsoft's official documentation for this. Besides adding your data sources, no special measures need to be done for your configuration to be found.

4.3.2 Unixodbc (Linux and OSX)

Unixodbc is a different beast. For one thing, you need to install it first. That is usually an easy task involving a simple `apt-get install unixodbc` (Linux) or `brew install unixodbc` (OSX with Homebrew).

However, unixodbc can be configured in many ways, both with and without graphical guidance. The official documentation is not always easy to follow, and finding what you are looking for may be more difficult than you planned for and may involve looking into unixodbc's source code.

The following primer assumes that no graphic tools are used (as is often common in server environments). It is not specific to turbodbc and based on information available at these locations:

- Unixodbc documentation "hub"
- Details on using unixodbc without a GUI
- Unixodbc man page

ODBC configuration files

Unixodbc's main configuration file is usually called `odbc.ini`. `odbc.ini` defines data sources that are available for connecting. It is a simple ini-style text file with the following layout:

```
[data source name]
Driver = /path/to/driver_library.so
Option1 = Value
Option2 = Other value

[other data source]
Driver = Identifier specified in odbcinst.ini file
OptionA = Value
```

The sections define data source names that can be used to connect with the respective database. Each section requires a `Driver` key. The value of this key may either contain the path to the database's *ODBC driver* or a key that identifies the driver in unixodbc's other configuration file `odbcinst.ini`. Each section may contain an arbitrary number of key-value pairs that specify further connection options. These connection options are driver-specific, so you need to refer to the ODBC driver's reference for that.

As mentioned before, unixodbc features a second (and optional) configuration file usually called `odbcinst.ini`. This file lists available ODBC drivers and labels them for convenient reference in `odbc.ini`. The file also follows the ini-style convention:

```
[driver A]
Driver = /path/to/driver_library.so
Threading = 2
Description = A driver to access ShinyDB databases

[driver B]
Driver = /some/other/driver/library.so
```

The sections define names that can be used as values for the `Driver` keys in `odbc.ini`. Each section needs to feature `Driver` keys themselves, where the values represent paths to the respective ODBC drivers. Some additional options are available such as the `Threading` level (see unixodbc's source code for details) or a `Description` field.

Configuration file placement options

Unixodbc has a few places where it looks for its configuration files:

- Global configuration files are found in `/etc/odbc.ini` and `/etc/odbcinst.ini`. Data sources defined in `/etc/odbc.ini` are available to all users of your computer. Drivers defined in `/etc/odbcinst.ini` can be used by all users of your computer.
- Users can define additional data sources by adding the file `~/.odbc.ini` to their home directory. It seems that a file called `~/.odbcinst.ini` has no effect.

- Users can add a folder in which to look for configuration files by setting the `ODBCSYSINI` environment variable:

```
> export ODBCSYSINI=/my/folder
```

  This will override the configuration files found at `/etc`. Place your configuration files at `/my/folder/odbc.ini` and `/my/folder/odbcinst.ini`.

- Users can override the path for the user-specific `odbc.ini` file by setting the `ODBCINI` environment variable:

```
> export ODBCINI=/full/path/to/odbc.ini
```

  If you set this option, unixodbc will no longer consider `~/.odbc.ini`.

Note: Do not expect the `ODBCINSTINI` environment variable to work just as `ODBCINI`. Instead, the `ODBCINSTINI` specifies the file name of `odbcinst.ini` relative to the value of the `ODBCSYSINI` variable. I suggest not to use this variable since it is outright confusing.

Configuration file placement recommendations

Here are a few typical scenarios:

- *First steps with unixodbc*: Create a new folder that contains `odbc.ini` and `odbcinst.ini`. Set the `ODBCSYSINI` variable to this folder.
- *Experimenting with a new database/driver*: Create a new folder that contains `odbc.ini` and `odbcinst.ini`. Set the `ODBCSYSINI` variable to this folder.
- *Provision a system with drivers*: Place an `odbcinst.ini` file at `/etc/odbcinst.ini`. Tell users to configure their databases using `~/odbc.ini` or setting `ODBCINI`.
- *Switching between multiple distinct configurations (test/production)*: Use the `ODBCSYSINI` variable if the configurations do not share common drivers. Otherwise, use the `ODBCINI` variable to switch between different `odbc.ini` files.

CHAPTER 5

Databases configuration and performance

As already outlined in the more general *ODBC configuration* section, connecting with your database via ODBC can be a real pain. Making matters worse, database performance may significantly depend on the configuration as well.

Well, this section tries to make life just a tad easier by providing recommended configurations for various databases. For some databases, comparisons with other database access modules are provided as well so that you know what kind of performance to expect.

Note: The quality of the *ODBC driver* for a given database heavily affects performance of all ODBC applications using this driver. Even though turbodbc was built to exploit buffering and what else the ODBC API has to offer, it cannot work wonders when the ODBC driver is not up to the task. In such circumstances, other, non-ODBC technologies may be available that outperform turbodbc by a considerable margin.

5.1 Exasol

Exasol is one of turbodbc's main development databases, and also provided the initial motivation for creating turbodbc. Here are the recommended settings for connecting to an Exasol database via ODBC using the turbodbc module for Python.

5.1.1 Recommended odbcinst.ini (Linux)

```
[Exasol driver]
Driver = /path/to/libexaodbc-uo2214lv1.so    # only when libodbc.so.2 is not present
Driver = /path/to/libexaodbc-uo2214lv2.so    # only when libodbc.so.2 is present
Threading = 2
```

- Exasol ships drivers for various versions of unixodbc. Any modern system should use the `uo2214` driver variants. Choose the `lv1` version if your system contains the file `libodbc.so.1`. If it does not, choose `lv2` instead.

- `Threading = 2` seems to be required to handle some thread issues with the driver.

5.1.2 Recommended odbcinst.ini (OSX)

```
[Exasol driver]
Driver = /path/to/libexaodbc-io418sys.dylib
Threading = 2
```

- The driver listed here is built with the iodbc library. All turbodbc tests work with this driver even though turbodbc uses unixodbc.
- `Threading = 2` seems to be required to handle some thread issues with the driver.

5.1.3 Recommended data source configuration

```
[Exasol]
DRIVER = Exasol driver
EXAHOST = <host>:<port_range>
EXAUID = <user>
EXAPWD = <password>
EXASCHEMA = <default_schema>
CONNECTIONLCALL = en_US.utf-8
```

- `CONNECTIONLCALL` is set to a locale with unicode support to avoid problems with retrieving Unicode characters.

5.1.4 Recommended turbodbc configuration

The default turbodbc connection options work fine for Exasol. You can probably tune the performance a little by increasing the read buffer size to 100 Megabytes. Exasol claims that their database works best with this setting.

See the *advanced options* for details.

```
>>> from turbodbc import connect, make_options, Megabytes
>>> options = make_options(read_buffer_size=Megabytes(100))
>>> connect(dsn="Exasol", turbodbc_options=options)
```

5.2 PostgreSQL

PostgreSQL is part of turbodbc's integration databases. That means that each commit in turbodbc's repository is automatically tested against PostgreSQL to ensure compatibility. Here are the recommended settings for connecting to a PostgreSQL database via ODBC using the turbodbc module for Python.

Note: PostgreSQL's free ODBC driver is not optimized for performance. Hence, there is not too much turbodbc can do to improve speed. You will achieve much better performance with psycopg2 or asyncpg.

5.2.1 Recommended odbcinst.ini (Linux)

```
[PostgreSQL Driver]
Driver      = /usr/lib/x86_64-linux-gnu/odbc/psqlodbcw.so
Threading   = 2
```

- `Threading = 2` is a safe choice to avoid potential thread issues with the driver, but you can also attempt using the driver without this option.

5.2.2 Recommended odbcinst.ini (OSX)

```
[PostgreSQL Driver]
Driver      = /usr/local/lib/psqlodbcw.so
Threading   = 2
```

- `Threading = 2` is a safe choice to avoid potential thread issues with the driver, but you can also attempt using the driver without this option.

5.2.3 Recommended data source configuration

```
[PostgreSQL]
Driver               = PostgreSQL Driver
Database             = <database name>
Servername           = <host>
UserName             = <user>
Password             = <password>
Port                 = <port, default is 5432>
Protocol             = 7.4
UseDeclareFetch      = 1
Fetch                = 10000
UseServerSidePrepare = 1
BoolsAsChar          = 0
ConnSettings         = set time zone 'UTC';
```

- `Protocol = 7.4` indicates version 3 of the PostgreSQL protocol is to be used, which is the latest one.
- `UseDeclareFetch = 1` means that the driver will only cache a few lines of the result set instead of the entire result set (which may easily eat up all available memory). The downside is that PostgreSQL will always cache exactly `Fetch` lines, no matter what the ODBC application (including turbodbc) actually requires.
- `Fetch = 10000` tells the PostgreSQL ODBC driver to fetch results from the database exactly in batches of 10,000 rows (no matter what turbodbc or any other ODBC application declares as the batch size). Using a high value may improve performance, but increases memory consumption in particular for tables with many columns. Low values reduces the memory footprint, but increases the number of database roundtrips, which may dominate performance for large result sets. The default value is 100.
- `UseServerSidePrepare = 1` means the server will prepare queries rather than the ODBC driver. This yields the most accurate results concerning result sets.
- `BoolsAsChar = 0` tells the driver to return boolean fields as booleans (`SQL_BIT` in ODBC-speak) instead of characters. Turbodbc can deal with booleans, so make sure to use them.
- `ConnSettings = set time zone 'UTC';` sets the session time zone to UTC. This will yield consistent values for fields of types `WITH TIME ZONE` information.

Note: ODBC configuration files generated by the PostgreSQL generation utility use the string `No` to deactivate options. It is recommended to replace all occurrences of `No` with `0`. The reason is that `Yes` will not work as expected, and also deactivate the option. Use `1` instead of `Yes` to get the desired result.

5.2.4 Recommended turbodbc configuration

The default turbodbc connection options work fine for PostgreSQL. As stated above, performance is not great due to the ODBC driver.

```
>>> from turbodbc import connect
>>> connect(dsn="PostgreSQL")
```

5.3 MySQL

MySQL is part of turbodbc's integration databases. That means that each commit in turbodbc's repository is automatically tested against MySQL to ensure compatibility. Here are the recommended settings for connecting to a MySQL database via ODBC using the turbodbc module for Python.

Note: You can use the MySQL ODBC driver to connect with databases that use the MySQL wire protocol. Examples for such databases are MariaDB, Amazon Aurora DB, or MemSQL.

5.3.1 Recommended odbcinst.ini (Linux)

```
[MySQL Driver]
Driver      = /usr/lib/x86_64-linux-gnu/odbc/libmyodbc.so
Threading   = 2
```

- `Threading = 2` is a safe choice to avoid potential thread issues with the driver, but you can also attempt using the driver without this option.

5.3.2 Recommended data source configuration

```
[MySQL]
Driver   = MySQL Driver
SERVER   = <host>
UID      = <user>
PASSWORD = <password>
DATABASE = <database name>
PORT     = <port, default is 3306>
INITSTMT = set session time_zone ='+00:00';
```

- `INITSTMT = set session time_zone ='+00:00';` sets the session time zone to UTC. This will yield consistent values for fields of type `TIMESTAMP`.

5.3.3 Recommended turbodbc configuration

The default turbodbc connection options work fine for MySQL.

```
>>> from turbodbc import connect
>>> connect(dsn="MySQL")
```

5.4 Microsoft SQL Server (MSSQL)

Microsoft SQL Server (MSSQL) is part of turbodbc's integration databases. That means that each commit in turbodbc's repository is automatically tested against MSSQL to ensure compatibility. Here are the recommended settings for connecting to a Microsoft SQL database via ODBC using the turbodbc module for Python.

5.4.1 Recommended odbcinst.ini (Linux)

On Linux, you have the choice between two popular drivers.

Official Microsoft ODBC driver

Microsoft offers an official ODBC driver for selected modern Linux distributions.

```
[MSSQL Driver]
Driver=/opt/microsoft/msodbcsql/lib64/libmsodbcsql-13.1.so.4.0
```

FreeTDS

FreeTDS is an open source ODBC driver that supports MSSQL. It is stable, has been around for well over decade and is actively maintained. However, it is not officially supported by Microsoft.

```
[FreeTDS Driver]
Driver = /usr/local/lib/libtdsodbc.so
```

5.4.2 Recommended odbcinst.ini (OSX)

FreeTDS seems to be the only available driver for OSX that can connect to MSSQL databases.

```
[FreeTDS Driver]
Driver = /usr/local/lib/libtdsodbc.so
```

5.4.3 Recommended data source configuration

Official Microsoft ODBC driver (Windows)

Put these values in your registry under the given key. Be sure to prefer the latest ODBC driver over any driver that may come bundled with your Windows version.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\MSSQL]
"Driver"="C:\\Windows\\system32\\msodbcsql17.dll"
"Server"="<host>"
"Database"="<database>"
```

Official Microsoft ODBC driver (Linux)

```
[MSSQL]
Driver          = MSSQL Driver
Server          = <host>,<port>
Database        = <database>
```

Note: You cannot specify credentials for MSSQL databases in `odbc.ini`.

FreeTDS data sources

```
[MSSQL]
Driver   = FreeTDS Driver
Server   = <host>
Port     = <post>
Database = <database>
```

Note: You cannot specify credentials for MSSQL databases in `odbc.ini`.

5.4.4 Recommended turbodbc configuration

The default turbodbc connection options have issues with Unicode strings on MSSQL. Please make sure to set the `prefer_unicode` *option*.

```
>>> from turbodbc import connect, make_options
>>> options = make_options(prefer_unicode=True)
>>> connect(dsn="MSSQL", turbodbc_options=options)
```

Warning: If you forget to set `prefer_unicode`, you may get anything from garbled up characters (e.g., `u'\xe2\x99\xa5'` instead of the unicode character `u'\u2665'`) or even ODBC error messages such as `[FreeTDS][SQL Server]Invalid cursor state`.

5.5 Netezza

Although IBM Netezza is not an integration tested database for turbodbc, some features have been added to support working with Netezza. Here are the recommended settings for connecting to a Netezza database via ODBC using the turbodbc module for Python.

Note: Probably due to the prefetch buffering, turbodbc seems to be about as fast as using pyodbc, however, fetching using NumPy or Arrow calls (`cursor.fetchallnumpy()` or `cursor.fetchallarrow()`) is approximately 3 times faster due to avoiding the conversion from SQL types to Python types to NumPy/Arrow types and just converting directly to NumPy/Arrow types.

5.5.1 Recommended odbcinst.ini (Linux)

```
[NZSQL]
Driver                    = /path/to/nz/lib/libnzsqlodbc3.so  # 32bit driver
Driver64                  = /path/to/nz/lib64/libnzodbc.so    # 64bit driver
UnicodeTranslationOption  = utf8
CharacterTranslationOption = all
PreFetch                  = 10000
Socket                    = 32000
```

5.5.2 Recommended data source configuration

```
[Netezza]
Driver                    = NZSQL
Description               = NetezzaSQL ODBC Connection
Servername                = <server_hostname>
Port                      = 5480
Database                  = <default_database>
Username                  = <username>
Password                  = <password>
```

5.5.3 Recommended turbodbc configuration

The default turbodbc connection options works for Netezza. However, `NVARCHAR` fields will be corrupted with the default settings. If you have `NVARCHAR` fields that you need to retrieve correctly you can set a few turbodbc options to support this, but this will cause turbodbc to use a lot more memory when retrieving result sets containing either `VARCHAR` or `NVARCHAR` fields (4x more per `VARCHAR` field and 2x more per `NVARCHAR` field).

See the *advanced options* for details.

```
>>> from turbodbc import connect, make_options, Megabytes
>>> options = make_options(read_buffer_size=Megabytes(100))
>>> connect(dsn="Netezza", turbodbc_options=options)
```

If retrieving string heavy datasets containing `NVARCHAR` fields:

```
>>> from turbodbc import connect, make_options, Megabytes
>>> options = make_options(read_buffer_size=Megabytes(250), fetch_wchar_as_char=True,␣
↪force_extra_capacity_for_unicode=True)
>>> connect(dsn="Netezza", turbodbc_options=options)
```

CHAPTER 6

Version history / changelog

From version 2.0.0, turbodbc adapts semantic versioning.

6.1 Version 4.4.0

- Bump pybind11 version to 2.9.1
- Bump dependency requirements to `pyarrow>=1.0` and `numpy>=1.19`
- Bump the used google test versions to 1.11.0

6.2 Version 4.3.1

- Update package requirements so that `pyarrow>0.17.1,<7.1` can be used.

6.3 Version 4.3.0

- Add Python 3.10 build
- Update package requirements so that `pyarrow>0.17.1,<6.1` can be used.
- Bump pybind11 version to 2.8.1

6.4 Version 4.2.1

- Update package requirements so that `pyarrow>0.17.1,<5.1` can be used.

6.5 Version 4.2.0

- Update package requirements so that `pyarrow>0.17.1,<4.1` can be used.
- Set minimal Python version to 3.7 following NEP 29.

6.6 Version 4.1.2

- Update package requirements so `pyarrow>0.17.1,<3.1` can be used.

6.7 Version 4.1.1

- Version bump as package version was not adapted.

6.8 Version 4.1.0

- Adjust unit tests to Apache Arrow master changes.
- Bump bundled pybind11 to 2.5.0

6.9 Version 4.0.0

- Minimal supported python version is 3.6.X now

6.10 Version 3.4.0

- Support Arrow 0.16.x and 0.17.x, require at least 0.15.x
- Minimal NumPy version was bumped to 1.16

6.11 Version 3.3.0

- Don't override but amend `CMAKE_CXX_FLAGS`
- Support Arrow 0.15.X

6.12 Version 3.2.0

- Release GIL while fetching batches using Apache Arrow engine
- Support Arrow 0.14.X

6.13 Version 3.1.1

- Correctly report odbc errors when freeing the statement handle as exceptions; see Github issue 153 (thanks @byjott)
- Support user-provided gmock/gtest, e.g. in conda environments via `conda install -c conda-forge gtest gmock`.
- Make source code compatible with Apache Arrow 0.13.0

6.14 Version 3.1.0

- Update to Apache Arrow 0.12
- Support the unicode datatype in the Arrow support. This primarily enables MS SQL support for the Arrow adapter.
- Windows support for the Arrow adapter.
- Add a new entry to the build matrix that tests Python 3.7 with conda and MS SQL on Linux.
- Big hands to @xhochy for making all these changes!

6.15 Version 3.0.0

- Adjust generators to conform to PEP-479
- Build wheels for Python 3.7 on Windows
- Drop support for Python 3.4
- Update to Apache Arrow 0.11

6.16 Version 2.7.0

- Added new keyword argument `fetch_wchar_as_char` to `make_options()`. If set to `True`, wide character types (`NVARCHAR`) are fetched and decoded as narrow character types for compatibility with certain databases/drivers (thanks @yaxxie).
- Added batched fetch support for Arrow as `fetcharrowbatches()` (thanks @mariusvniekerk).
- Support (u)int8, (u)int16, (u)int32 Arrow columns on `executemanycolumns()` (thanks @xhochy).

6.17 Version 2.6.0

- Added support for `with` blocks for `Cursor` and `Connection` objects. This makes turbodbc conform with PEP 343 (thanks @AtomBaf)
- Added new keyword argument `force_extra_capacity_for_unicode` to `make_options()`. If set to `True`, memory allocation is modified to operate under the assumption that the database driver reports field lengths in characters, rather than code units (thanks @yaxxie).
- Updated Apache Arrow support to work with both versions 0.8.0 and 0.9.0 (thanks @pacman82)

- Fixed a bug that led to `handle limit exceeded` error messages when `Cursor` objects were not closed *manually*. With this fix, cursors are garbage collected as expected.

6.18 Version 2.5.0

- Added an option to `fetchallarrow()` that fetches integer columns in the smallest possible integer type the retrieved values fit in. While this reduces the memory footprint of the resulting table, the schema of the table is now dependent on the data it contains.
- Updated Apache Arrow support to work with version 0.8.x

6.19 Version 2.4.1

- Fixed a memory leak on `fetchallarrow()` that increased the reference count of the returned table by one too much.

6.20 Version 2.4.0

- Added support for Apache Arrow `pyarrow.Table` objects as the input for `executemanycolumns()`. In addition to direct Arrow support, this should also help with more graceful handling of Pandas DataFrames as `pa.Table.from_pandas(...)` handles additional corner cases of Pandas data structures. Big thanks to @xhochy!

6.21 Version 2.3.0

- Added an option to `fetchallarrow()` that enables the fetching of string columns as dictionary-encoded string columns. In most cases, this increases performance and reduces RAM usage. Arrow columns of type `dictionary[string]` will result in `pandas.Categorical` columns on conversion.
- Updated pybind11 dependency to version 2.2+
- Fixed a symbol visibility issue when building Arrow unit tests on systems that hide symbols by default.

6.22 Version 2.2.0

- Added new keyword argument `large_decimals_as_64_bit_types` to `make_options()`. If set to `True`, decimals with more than `18` digits will be retrieved as 64 bit integers or floats as appropriate. The default retains the previous behavior of returning strings.
- Added support for `datetime64[ns]` data type for `executemanycolumns()`. This is particularly helpful when dealing with pandas `DataFrame` objects, since this is the type that contains time stamps.
- Added the keyword argument `limit_varchar_results_to_max` to `make_options()`. This allows to truncate `VARCHAR(n)` fields to `varchar_max_character_limit` characters, see the next item.
- Added possibility to enforce NumPy and Apache Arrow requirements using extra requirements during installation: `pip install turbodbc[arrow,numpy]`
- Updated Apache Arrow support to work with version 0.6.x

- Fixed an issue with retrieving result sets with `VARCHAR(max)` fields and similar types. The size of the buffer allocated for such fields can be controlled with the `varchar_max_character_limit` option to `make_options()`.
- Fixed an issue with some versions of Boost that lead to problems with `datetime64[us]` columns with `executemanycolumns()`. An overflow when converting microseconds since 1970 to a database-readable timestamp could happen, badly garbling the timestamps in the process. The issue was surfaced with Debian 7's Boost version (1.49), although the Boost issue was allegedly fixed with version 1.43.
- Fixed an issue that lead to undefined behavior when character sequences could not be decoded into Unicode code points. The new (and defined) behavior is to ignore the offending character sequences completely.

6.23 Version 2.1.0

- Added new method `cursor.executemanycolumns()` that accepts parameters in columnar fashion as a list of NumPy (masked) arrays.
- CMake build now supports `conda` environments
- CMake build offers `DISABLE_CXX11_ABI` option to fix linking issues with `pyarrow` on systems with the new C++11 compliant ABI enabled

6.24 Version 2.0.0

- Initial support for the arrow data format with the `Cursor.fetchallarrow()` method. Still in alpha stage, mileage may vary (Windows not yet supported, UTF-16 unicode not yet supported). Big thanks to @xhochy!
- `prefer_unicode` option now also affects column name rendering when gathering results from the database. This effectively enables support for Unicode column names for some databases.
- Added module version number `turbodbc.__version__`
- Removed deprecated performance options for `connect()`. Use `connect(..., turbodbc_options=make_options(...))` instead.

6.25 Earlier versions (not conforming to semantic versioning)

The following versions do not conform to semantic versioning. The meaning of the `major.minor.revision` versions is:

- Major: psychological ;-)
- Minor: If incremented, this indicates a breaking change
- Revision: If incremented, indicates non-breaking change (either feature or bug fix)

6.26 Version 1.1.2

- Added `autocommit` as a keyword argument to `make_options()`. As the name suggests, this allows you to enable automatic `COMMIT` operations after each operation. It also improves compatibility with databases that do not support transactions.

- Added `autocommit` property to `Connection` class that allows switching autocommit mode after the connection was created.
- Fixed bug with `cursor.rowcount` not being reset to `-1` when calls to `execute()` or `executemany()` raised exceptions.
- Fixed bug with `cursor.rowcount` not showing the correct value when manipulating queries were used without placeholders, i.e., with parameters baked into the query.
- Global interpreter lock (GIL) is released during some operations to facilitate basic multi-threading (thanks @chmp)
- Internal: The return code `SQL_SUCCESS_WITH_INFO` is now treated as a success instead of an error when allocating environment, connection, and statement handles. This may improve compatibility with some databases.

6.27 Version 1.1.1

- Windows is now _officially_ supported (64 bit, Python 3.5 and 3.6). From now on, code is automatically compiled and tested on Linux, OSX, and Windows (thanks @TWAC for support). Windows binary wheels are uploaded to pypi.
- Added supported for fetching results in batches of NumPy objects with `cursor.fetchnumpybatches()` (thanks @yaxxie)
- MSSQL is now part of the Windows test suite (thanks @TWAC)
- `connect()` now allows to specify a `connection_string` instead of individual arguments that are then compiles into a connection string (thanks @TWAC).

6.28 Version 1.1.0

- Added support for databases that require Unicode data to be transported in UCS-2/UCS-16 format rather than UTF-8, e.g., MSSQL.
- Added _experimental_ support for Windows source distribution builds. Windows builds are not fully (or automatically) tested yet, and still require significant effort on the user side to compile (thanks @TWAC for this initial version)
- Added new `cursor.fetchnumpybatches()` method which returns a generator to iterate over result sets in batch sizes as defined by buffer size or rowcount (thanks @yaxxie)
- Added `make_options()` function that take all performance and compatibility settings as keyword arguments.
- Deprecated all performance options (`read_buffer_size`, `use_async_io`, and `parameter_sets_to_buffer`) for `connect()`. Please move these keyword arguments to `make_options()`. Then, set `connect{}`'s new keyword argument `turbodbc_options` to the result of `make_options()`. This effectively separates performance options from options passed to the ODBC connection string.
- Removed deprecated option `rows_to_buffer` from `turbodbc.connect()` (see version 0.4.1 for details).
- The order of arguments for `turbodbc.connect()` has changed; this may affect you if you have not used keyword arguments.
- The behavior of `cursor.fetchallnumpy()` has changed a little. The `mask` attribute of a generated `numpy.MaskedArray` instance is shortened to `False` from the previous `[False, ..., False]` if the mask is `False` for all entries. This can cause problems when you access individual indices of the mask.

- Updated `pybind11` requirement to at least `2.1.0`.
- Internal: Some types have changed to accomodate for Linux/OSX/Windows compatibility. In particular, a few `long` types were converted to `intptr_t` and `int64_t` where appropriate. In particular, this affects the `field` type that may be used by C++ end users (so they exist).

6.29 Version 1.0.5

- Internal: Remove some `const` pointers to resolve some compile issues with xcode 6.4 (thanks @xhochy)

6.30 Version 1.0.4

- Added possibility to set unixodbc include and library directories in setup.py. Required for conda builds.

6.31 Version 1.0.3

- Improved compatibility with ODBC drivers (e.g. FreeTDS) that do not support ODBC's `SQLDescribeParam()` function by using a default parameter type.
- Used a default parameter type when the ODBC driver cannot determine a parameter's type, for example when using column expressions for `INSERT` statements.
- Improved compatibility with some ODBC drivers (e.g. Microsoft's official MSSQL ODBC driver) for setting timestamps with fractional seconds.

6.32 Version 1.0.2

- Added support for chaining operations to `Cursor.execute()` and `Cursor.executemany()`. This allows one-liners such as `cursor.execute("SELECT 42").fetchallnumpy()`.
- Right before a database connection is closed, any open transactions are explicitly rolled back. This improves compatibility with ODBC drivers that do not perform automatic rollbacks such as Microsoft's official ODBC driver.
- Improved stability of turbodbc when facing errors while closing connections, statements, and environments. In earlier versions, connection timeouts etc. could have lead to the Python process's termination.
- Source distribution now contains license, readme, and changelog.

6.33 Version 1.0.1

- Added support for OSX

6.34 Version 1.0.0

- Added support for Python 3. Python 2 is still supported as well. Tested with Python 2.7, 3.4, 3.5, and 3.6.

- Added `six` package as dependency
- Turbodbc uses pybind11 instead of Boost.Python to generate its Python bindings. pybind11 is available as a Python package and automatically installed when you install turbodbc. Other boost libraries are still required for other aspects of the code.
- A more modern compiler is required due to the pybind11 dependency. GCC 4.8 will suffice.
- Internal: Move remaining stuff depending on python to turbodbc_python
- Internal: Now requires CMake 2.8.12+ (get it with `pip install cmake`)

6.35 Version 0.5.1

- Fixed build issue with older numpy versions, e.g., 1.8 (thanks @xhochy)

6.36 Version 0.5.0

- Improved performance of parameter-based operations.
- Internal: Major modifications to the way parameters are handled.

6.37 Version 0.4.1

- The size of the input buffers for retrieving result sets can now be set to a certain amount of memory instead of using a fixed number of rows. Use the optional `read_buffer_size` parameter of `turbodbc.connect()` and set it to instances of the new top-level classes `Megabytes` and `Rows` (thanks @LukasDistel).
- The read buffer size's default value has changed from 1,000 rows to 20 MB.
- The parameter `rows_to_buffer` of `turbodbc.connect()` is _deprecated_. You can set the `read_buffer_size` to `turbodbc.Rows(1000)` for the same effect, though it is recommended to specify the buffer size in MB.
- Internal: Libraries no longer link `libpython.so` for local development (linking is already done by the Python interpreter). This was always the case for the libraries in the packages uploaded to PyPI, so no change was necessary here.
- Internal: Some modifications to the structure of the underlying C++ code.

6.38 Version 0.4.0

- NumPy support is introduced to turbodbc for retrieving result sets. Use `cursor.fetchallnumpy` to retrieve a result set as an `OrderedDict` of `column_name: column_data` pairs, where `column_data` is a NumPy `MaskedArray` of appropriate type.
- Internal: Single `turbodbc_intern` library was split up into three libraries to keep NumPy support optional. A few files were moved because of this.

6.39 Version 0.3.0

- turbodbc now supports asynchronous I/O operations for retrieving result sets. This means that while the main thread is busy converting an already retrieved batch of results to Python objects, another thread fetches an additional batch in the background. This may yield substantial performance improvements in the right circumstances (results are retrieved in roughly the same speed as they are converted to Python objects).

  Ansynchronous I/O support is experimental. Enable it with `turbodbc.connect('My data source name', use_async_io=True)`

6.40 Version 0.2.5

- C++ backend: `turbodbc::column` no longer automatically binds on construction. Call `bind()` instead.

6.41 Version 0.2.4

- Result set rows are returned as native Python lists instead of a not easily printable custom type.
- Improve performance of Python object conversion while reading result sets. In tests with an Exasol database, performance got about 15% better.
- C++ backend: `turbodbc::cursor` no longer allows direct access to the C++ `field` type. Instead, please use the `cursor`'s `get_query()` method, and construct a `turbodbc::result_sets::field_result_set` using the `get_results()` method.

6.42 Version 0.2.3

- Fix issue that only lists were allowed for specifying parameters for queries
- Improve parameter memory consumption when the database reports very large string parameter sizes
- C++ backend: Provides more low-level ways to access the result set

6.43 Version 0.2.2

- Fix issue that `dsn` parameter was always present in the connection string even if it was not set by the user's call to `connect()`
- Internal: First version to run on Travis.
- Internal: Use pytest instead of unittest for testing
- Internal: Allow for integration tests to run in custom environment
- Internal: Simplify integration test configuration

6.44 Version 0.2.1

- Internal: Change C++ test framework to Google Test

6.45 Version 0.2.0

- New parameter types supported: `bool`, `datetime.date`, `datetime.datetime`
- `cursor.rowcount` returns number of affected rows for manipulating queries
- `Connection` supports `rollback()`
- Improved handling of string parameters

6.46 Version 0.1.0

Initial release

CHAPTER 7

Troubleshooting

This section contains advice on how to troubleshoot ODBC connections. The advice contained here is not specific to turbodbc, but very related.

Note: This section currently assumes you are on a Linux/OSX machine that uses unixodbc as a *driver manager*. Windows users may find the contained information useful, but should expect some additional transfer work adjusting the advice to the Windows platform.

7.1 Testing your ODBC configuration

You can test your configuration with turbodbc, obviously, by creating a connection. It is preferable, however, to use the tool `isql` that is shipped together with `unixodbc`. It is a very simple program that does not try anything fancy and is perfectly suited for debugging. If you configuration does not work with `isql`, it will not work with turbodbc.

Note: Before you file an issue with turbodbc, please make sure that you can actually connect to your database using `isql`.

When you have selected an ODBC configuration as outlined above, enter the following command in a shell:

```
> isql "data source name" user password -v
```

Specifying user and password is optional. On success, this will output a shell such as this:

```
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
```

(continues on next page)

(continued from previous page)

```
|                                       |
+---------------------------------------+
SQL>
```

You can type in any SQL command you wish to test or leave the shell with the `quit` command. In case of errors, a hopefully somewhat helpful error message will be displayed.

7.2 Common configuration errors

```
[IM002][unixODBC][Driver Manager]Data source name not found, and no default driver␣
↪specified
[ISQL]ERROR: Could not SQLConnect
```

This usually means that the data source name could not be found because the configuration is not active. Troubleshooting:

- Check for typos in data source names in `odbc.ini` or your shell.
- Check if the correct `odbc.ini` file is used.
- Check the values of `$ODBCSYSINI` and `$ODBCINI` (usually only one should be set).
- Check if `$ODBCINSTINI` is set (usually it should not be set). Unset the variable.
- Check your data source has a `Driver` section.

```
[01000][unixODBC][Driver Manager]Can't open lib '/path/to/driver.so' : file not found
[ISQL]ERROR: Could not SQLConnect
```

This means the ODBC driver library cannot be opened. The suggested cause "file not found" may be misleading, however, as this message may be printed even if the file exists. Troubleshooting:

- Check whether the library exists at the specified location.
- Check whether you have permission to *read* the library.
- Check whether the library depends on other shared libraries that are not present:
  - On Linux, use `ldd /path/to/library.so`
  - On OSX, use `otool -L /path/to/library.dylib`
- Check whether any superfluous non-printable characters are present in your `odbc.ini` or `odbcinst.ini` near the `Driver` line. Been there, done that...

7.3 More subtle issues

There are still a few errors to make even when you can successfully establish a connection with your database. Here are a few common ones:

- *Unsuitable locale*: Some databases return data in a format dictated by your current locale settings. For example, unicode output may require a locale that supports UTF-8, such as `en-US.utf-8`. Otherwise, replacement characters appear instead of unicode characters. Set the locale via environment variables such as `LC_ALL` or check whether your driver supports setting a locale in its connection options.

- *Time zones*: ODBC does not feature a dedicated type that is aware of time zones or the distinction between local time and UTC. Some databases, however, feature separate types for, e.g., timestamps with and without time zone information. ODBC drivers now need to find a way to make such information available to ODBC applications. A usual way to do this is to convert (some) values into the session time zone. This may lead to conflicting information when sessions with different time zones access the same database. The recommendation would be to fix the session time zone to UTC whenever possible to keep things consistent.

CHAPTER 8

Frequently asked questions

8.1 Can I use turbodbc together with SQLAlchemy?

Using Turbodbc in combination with SQLAlchemy is possible for a (currently) limited number of databases:

- EXASOL: sqlalchemy_exasol
- MSSQL: sqlalchemy-turbodbc

All of the above packages are available on PyPI. There are also more experimental implementations available:

- Vertica: Unofficial fork of sqlalchemy-vertica

8.2 Where would I report issues concerning turbodbc?

In this case, please use turbodbc's issue tracker on GitHub.

8.3 Where can I ask questions regarding turbodbc?

Basically, you can ask them anywhere, chances to get a helpful answer may vary, though. I suggest asking questions either using turbodbc's issue tracker on GitHub or by heading over to stackoverflow.

8.4 Is there a guided tour through turbodbc's entrails?

Yes, there is! Check out these blog posts on the making of turbodbc:

- Part one: Wrestling with the side effects of a C API. This explains the C++ layer that is used to handle all calls to the ODBC API.

- Part two: C++ to Python This explains how concepts of the ODBC API are transformed into an API compliant with Python's database API, including making use of pybind11.

8.5 I love Twitter! Is turbodbc on Twitter?

Yes, it is! Just follow @turbodbc for the latest turbodbc talk and news about related technologies.

8.6 How can I find our more about turbodbc's latest developments?

There are a few options:

- Watch the turbodbc project on GitHub. This way, you will get mails for new issues, updates issues, and the like.
- Periodically read turbodbc's change log
- Follow @turbodbc on Twitter. There will be tweets for new releases.

CHAPTER 9

Contributing

9.1 Ways to contribute

Contributions to turbodbc are most welcome! There are many options how you can contribute, and not all of them require you to be an expert programmer:

- Ask questions and raise issues on GitHub. This may influence turbodbc's roadmap.
- If you like turbodbc, star/fork/watch the project on GitHub. This will improve visibility, and potentially attracts more contributors.
- Report performance comparisons between turbodbc and other means to access a database.
- Tell others about turbodbc on your blog, Twitter account, or at the coffee machine at work.
- Improve turbodbc's documentation by creating pull requests on GitHub.
- Improve existing features by creating pull requests on GitHub.
- Add new features by creating pull requests on GitHub.
- Implement dialects for SQLAlchemy that connect to databases using turbodbc.

9.2 Pull requests

Pull requests are appreciated in general, but not all pull requests will be accepted into turbodbc. Before starting to work on a pull request, please make sure your pull request is aligned with turbodbc's vision of creating fast ODBC database access for data scientists. The safest way is to ask on GitHub whether a certain feature would be appreciated.

After forking the project and making your modifications, you can create a new pull request on turbodbc's GitHub page. This will trigger an automatic build and, eventually, a code review. During code reviews, I try to make sure that the added code complies with clean code principles such as single level of abstraction, single responsibility principle, principle of least astonishment, etc.

If you do not know what all of this means, just try to keep functions small (up to five lines) and find meaningful names. If you feel like writing a comment, think about whether the comment would make a nice variable or function name, and refactor your code accordingly.

I am well aware that the current code falls short of clean code standards in one place or another. Please do not take criticism regarding your code personally. Any comments are purely directed to improve the quality of turbodbc's code base over its current state.

9.3 Development version

9.3.1 Earthly Setup

You can build turbodbc and run the full test suite with earthly.

See the install instructions on how to get earthly: https://earthly.dev/get-earthly.

Build and test turbodbc with the default arguments:

```
earthly --allow-privileged +test
```

You can use the interactive mode to get a shell when tests fail:

```
earthly -i --allow-privileged +test
```

Run a specific test setup with an export of the build package:

```
earthly -P +test-python3.8-arrow3.x.x
```

The coverage results as well as the build tar.gz for turbodbc can be found in the created *result* directory. To run other specific test targets and arguments, like different python or pyarrow versions, please see the Earthfile.

Run the full tests for different Python and package versions:

```
earthly --allow-privileged +test-all
```

Build and save a Docker image for development usage:

```
earthly +docker
```

It will create and tag a docker image *turbodbc:latest* which can be used for IDEs or to run builds with less overhead and with a mounted source code and build directory:

```
docker run -v $PWD:/src -it turbodbc:latest
```

This will give you a shell in a running container, there you can execute the different build steps as you wish:

- build setup: *cmake -DBOOST_ROOT=$CONDA_PREFIX -DBUILD_COVERAGE=ON -DCMAKE_INSTALL_PREFIX=./dist -DPYTHON_EXECUTABLE=/miniconda/envs/turbodbc-dev/bin/python -GNinja ..*
- compile: *ninja*
- install: *cmake –build . –target install*
- unit tests: *ctest -E turbodbc_integration_test –verbose*

To run the integration tests in this setup, it is required that the docker-compose setup is running:

```
docker-compose -f earthly/docker-compose.yml up

# to stop it
docker-compose -f earthly/docker-compose.yml down
```

Start the container within the host network stack:

```
docker run -v $PWD:/src --network=host -it turbodbc:latest
```

There you can run the integration tests:

```
# the mssql database has to be created
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'StrongPassword1' -Q 'CREATE␣
↪DATABASE test_db'

ctest --verbose
```

9.3.2 Manual Host Native Setup

For developing new features or just sampling the latest version of turbodbc directly on your host do the following:

1. Make sure your development environment meets the prerequisites mentioned in the *getting started guide*.
2. Create development environment depending on your Python package manager.
   - For a pip-based workflow, create a virtual environment, activate it, and install the necessary packages numpy, pyarrow, pytest, and mock:

     ```
     pip install numpy pytest pytest-cov mock pyarrow
     ```

     Make sure you have a recent version of `cmake` installed. For some operating systems, binary wheels are available in addition to the package your operating system offers:

     ```
     pip install cmake
     ```

   - If you're using `conda` to manage your python packages, you can install the dependencies from conda-forge:

     ```
     conda create -y -q -n turbodbc-dev pyarrow numpy pybind11 boost-cpp \
         pytest pytest-cov mock cmake unixodbc gtest gmock -c conda-forge
     source activate turbodbc-dev
     ```

3. Clone turbodbc into the virtual environment somewhere:

   ```
   git clone https://github.com/blue-yonder/turbodbc.git
   ```

4. `cd` into the git repo and get `pybind11`

   ```
   wget -q https://github.com/pybind/pybind11/archive/v2.9.1.tar.gz
   tar xvf v2.9.1.tar.gz
   ```

5. Create a build directory somewhere and `cd` into it.
6. Execute the following command:

   ```
   cmake -DCMAKE_INSTALL_PREFIX=./dist -DPYTHON_EXECUTABLE=`which python` /path/to/
   ↪turbodbc
   ```

where the final path parameter is the directory to the turbodbc git repo, specifically the directory containing `setup.py`. This `cmake` command will prepare the build directory for the actual build step.

Note: The `-DPYTHON_EXECUTABLE` flag is not strictly necessary, but it helps `pybind11` to detect the correct Python version, in particular when using virtual environments.

---

7. Run `make`. This will build (compile) the source code.

Note: Some Linux distributions with very modern C++ compilers, e.g., Fedora 24+, may yield linker error messages such as

```
arrow_result_set_test.cpp:168: undefined reference to␣
↪`arrow::Status::ToString[abi:cxx11]() const'
```

This error is caused because some Linux distributions use a C++11 compliant ABI version of the standard library, while the `pyarrow` manylinux wheel does not. In this case, throw away your build directory and use

```
cmake -DDISABLE_CXX11_ABI=ON -DCMAKE_INSTALL_PREFIX=./dist -DPYTHON_
↪EXECUTABLE=`which python` /path/to/turbodbc
```

in place of the CMake command in the previous step.

---

8. At this point you can run the test suite. First, make a copy of the relevant json documents from the turbodbc `python/turbodbc_test` directory, there's one for each database. Then edit your copies with the relevant credentials. Next, set the environment variable `TURBODBC_TEST_CONFIGURATION_FILES` as a comma-separated list of the json files you've just copied and run the test suite, as follows:

```
export TURBODBC_TEST_CONFIGURATION_FILES="<Postgres json file>,<MySql json file>,
↪<MS SQL json file>"
ctest --output-on-failure
```

9. Finally, to create a Python source distribution for `pip` installation, run the following from the build directory:

```
make install
cd dist
python setup.py sdist
```

This will create a `turbodbc-x.y.z.tar.gz` file locally which can be used by others to install turbodbc with `pip install turbodbc-x.y.z.tar.gz`.

CHAPTER 10

API reference

turbodbc.**connect**(**args*, ***kwds*)

Create a connection with the database identified by the `dsn` or the `connection_string`.

Parameters

- **dsn** – Data source name as given in the (unix) odbc.ini file or (Windows) ODBC Data Source Administrator tool.
- **turbodbc_options** – Options that control how turbodbc interacts with the database. Create such a struct with *turbodbc.make_options()* or leave this blank to take the defaults.
- **connection_string** – Preformatted ODBC connection string. Specifying this and dsn or kwargs at the same time raises ParameterError.
- ****kwargs** – You may specify additional options as you please. These options will go into the connection string that identifies the database. Valid options depend on the specific database you would like to connect with (e.g. *user* and *password*, or *uid* and *pwd*)

Returns A connection to your database

turbodbc.**make_options**(*read_buffer_size=None*, *parameter_sets_to_buffer=None*, *varchar_max_character_limit=None*, *prefer_unicode=None*, *use_async_io=None*, *autocommit=None*, *large_decimals_as_64_bit_types=None*, *limit_varchar_results_to_max=None*, *force_extra_capacity_for_unicode=None*, *fetch_wchar_as_char=None*)

Create options that control how turbodbc interacts with a database. These options affect performance for the most part, but some options may require adjustment so that all features work correctly with certain databases.

If a parameter is set to *None*, this means the default value is used.

Parameters

- **read_buffer_size** – Affects performance. Controls the size of batches fetched from the database when reading result sets. Can be either an instance of `turbodbc.Megabytes` (recommended) or `turbodbc.Rows`.

- **parameter_sets_to_buffer** – Affects performance. Number of parameter sets (rows) which shall be transferred to the server in a single batch when `executemany()` is called. Must be an integer.
- **varchar_max_character_limit** – Affects behavior/performance. If a result set contains fields of type `VARCHAR(max)` or `NVARCHAR(max)` or the equivalent type of your database, buffers will be allocated to hold the specified number of characters. This may lead to truncation. The default value is `65535` characters. Please note that large values reduce the risk of truncation, but may affect the number of rows in a batch of result sets (see `read_buffer_size`). Please note that this option only relates to retrieving results, not sending parameters to the database.
- **use_async_io** – Affects performance. Set this option to `True` if you want to use asynchronous I/O, i.e., while Python is busy converting database results to Python objects, new result sets are fetched from the database in the background.
- **prefer_unicode** – May affect functionality and performance. Some databases do not support strings encoded with UTF-8, leading to UTF-8 characters being misinterpreted, misrepresented, or downright rejected. Set this option to `True` if you want to transfer character data using the UCS-2/UCS-16 encoding that use (multiple) two-byte instead of (multiple) one-byte characters.
- **autocommit** – Affects behavior. If set to `True`, all queries and commands executed with `cursor.execute()` or `cursor.executemany()` will be succeeded by an implicit `COMMIT` operation, persisting any changes made to the database. If not set or set to `False`, users has to take care of calling `cursor.commit()` themselves.
- **large_decimals_as_64_bit_types** – Affects behavior. If set to `True`, `DECIMAL(x, y)` results with `x > 18` will be rendered as 64 bit integers (`y == 0`) or 64 bit floating point numbers (`y > 0`), respectively. Use this option if your decimal data types are larger than the data they actually hold. Using this data type can lead to overflow errors and loss of precision. If not set or set to `False`, large decimals are rendered as strings.
- **limit_varchar_results_to_max** – Affects behavior/performance. If set to `True`, any text-like fields such as `VARCHAR(n)` and `NVARCHAR(n)` will be limited to a maximum size of `varchar_max_character_limit` characters. This may lead to values being truncated, but reduces the amount of memory required to allocate string buffers, leading to larger, more efficient batches. If not set or set to `False`, strings can exceed `varchar_max_character_limit` in size if the database reports them this way. For fields such as `TEXT`, some databases report a size of 2 billion characters. Please note that this option only relates to retrieving results, not sending parameters to the database.
- **force_extra_capacity_for_unicode** – Affects behavior/performance. Some ODBC drivers report the length of the `VARCHAR`/`NVARCHAR` field rather than the number of code points for which space is required to be allocated, resulting in string truncations. Set this option to `True` to increase the memory allocated for `VARCHAR` and `NVARCHAR` fields and prevent string truncations. Please note that this option only relates to retrieving results, not sending parameters to the database.
- **fetch_wchar_as_char** – Affects behavior. Some ODBC drivers retrieve single byte encoded strings into `NVARCHAR` fields of result sets, which are decoded incorrectly by turbodbc default settings, resulting in corrupt strings. Set this option to `True` to have turbodbc treat `NVARCHAR` types as narrow character types when decoding the fields in result sets. Please note that this option only relates to retrieving results, not sending parameters to the database.

Returns An option struct that is suitable to pass to the `turbodbc_options` parameter of

turbodbc.connect()

class turbodbc.connection.**Connection**(*impl*)

autocommit
This attribute controls whether changes are automatically committed after each execution or not.

close()
Close the connection and all associated cursors. This will implicitly roll back any uncommitted operations.

commit(**kwds*)
Commits the current transaction

cursor(**kwds*)
Create a new Cursor instance associated with this Connection

Returns A new Cursor instance

rollback(**kwds*)
Roll back all changes in the current transaction

class turbodbc.cursor.**Cursor**(*impl*)
This class allows you to send SQL commands and queries to a database and retrieve associated result sets.

close()
Close the cursor.

description
Retrieve a description of the columns in the current result set

Returns

A tuple of seven elements. Only some elements are meaningful:

- Element #0 is the name of the column
- Element #1 is the type code of the column
- Element #6 is true if the column may contain NULL values

execute(**kwds*)
Execute an SQL command or query

Parameters

- **sql** – A (unicode) string that contains the SQL command or query. If you would like to use parameters, please use a question mark ? at the location where the parameter shall be inserted.
- **parameters** – An iterable of parameter values. The number of values must match the number of parameters in the SQL string.

Returns The Cursor object to allow chaining of operations.

executemany(**kwds*)
Execute an SQL command or query with multiple parameter sets passed in a row-wise fashion. This function is part of PEP-249.

Parameters

- **sql** – A (unicode) string that contains the SQL command or query. If you would like to use parameters, please use a question mark ? at the location where the parameter shall be inserted.

- **parameters** – An iterable of iterable of parameter values. The outer iterable represents separate parameter sets. The inner iterable contains parameter values for a given parameter set. The number of values of each set must match the number of parameters in the SQL string.

Returns The `Cursor` object to allow chaining of operations.

executemanycolumns(***kwds*)

Execute an SQL command or query with multiple parameter sets that are passed in a column-wise fashion as opposed to the row-wise parameters in `executemany()`. This function is a turbodbc-specific extension to PEP-249.

Parameters

- **sql** – A (unicode) string that contains the SQL command or query. If you would like to use parameters, please use a question mark `?` at the location where the parameter shall be inserted.
- **columns** – An iterable of NumPy MaskedArrays. The Arrays represent the columnar parameter data,

Returns The `Cursor` object to allow chaining of operations.

fetchall(***kwds*)

Fetches a list of all rows in the active result set generated with `execute()` or `executemany()`.

Returns A list of rows

fetchallarrow(*strings_as_dictionary=False*, *adaptive_integers=False*)

Fetches all rows in the active result set generated with `execute()` or `executemany()`.

Parameters

- **strings_as_dictionary** – If true, fetch string columns as dictionary[string] instead of a plain string column.
- **adaptive_integers** – If true, instead of the integer type returned by the database (driver), this produce integer columns with the smallest possible integer type in which all values can be stored. Be aware that here the type depends on the resulting data.

Returns `pyarrow.Table`

fetchallnumpy()

Fetches all rows in the active result set generated with `execute()` or `executemany()`.

Returns An `OrderedDict` of *columns*, where the keys of the dictionary are the column names. The columns are of NumPy's `MaskedArray` type, where the optimal data type for each result set column is chosen automatically.

fetcharrowbatches(*strings_as_dictionary=False*, *adaptive_integers=False*)

Fetches rows in the active result set generated with `execute()` or `executemany()` as an iterable of arrow tables.

Parameters

- **strings_as_dictionary** – If true, fetch string columns as dictionary[string] instead of a plain string column.
- **adaptive_integers** – If true, instead of the integer type returned by the database (driver), this produce integer columns with the smallest possible integer type in which all values can be stored. Be aware that here the type depends on the resulting data.

Returns generator of `pyarrow.Table`

fetchmany(***kwds*)
Fetches a batch of rows in the active result set generated with `execute()` or `executemany()`.

Parameters **size** – Controls how many rows are returned. The default `None` means that the value of Cursor.arraysize is used.

Returns A list of rows

fetchnumpybatches()
Returns an iterator over all rows in the active result set generated with `execute()` or `executemany()`.

Returns An iterator you can use to iterate over batches of rows of the result set. Each batch consists of an `OrderedDict` of NumPy `MaskedArray` instances. See `fetchallnumpy()` for details.

fetchone(***kwds*)
Returns a single row of a result set. Requires an active result set on the database generated with `execute()` or `executemany()`.

Returns Returns `None` when no more rows are available in the result set

setinputsizes(*sizes*)
Has no effect since turbodbc automatically picks appropriate return types and sizes. Method exists since PEP-249 requires it.

setoutputsize(*size*, *column=None*)
Has no effect since turbodbc automatically picks appropriate input types and sizes. Method exists since PEP-249 requires it.

class turbodbc.exceptions.**Error**
turbodbc's basic error class

class turbodbc.exceptions.**InterfaceError**
An error that is raised whenever you use turbodbc incorrectly

class turbodbc.exceptions.**DatabaseError**
An error that is raised when the database encounters an error while processing your commands and queries

class turbodbc.exceptions.**ParameterError**
An error that is raised when you use connection arguments that are supposed to be mutually exclusive

CHAPTER 11

Indices and tables

- genindex
- modindex

Index

A

autocommit (*turbodbc.connection.Connection attribute*), 55

C

close() (*turbodbc.connection.Connection method*), 55
close() (*turbodbc.cursor.Cursor method*), 55
commit() (*turbodbc.connection.Connection method*), 55
connect() (*in module turbodbc*), 53
Connection (*class in turbodbc.connection*), 55
Cursor (*class in turbodbc.cursor*), 55
cursor() (*turbodbc.connection.Connection method*), 55

D

DatabaseError (*class in turbodbc.exceptions*), 57
description (*turbodbc.cursor.Cursor attribute*), 55

E

Error (*class in turbodbc.exceptions*), 57
execute() (*turbodbc.cursor.Cursor method*), 55
executemany() (*turbodbc.cursor.Cursor method*), 55
executemanycolumns() (*turbodbc.cursor.Cursor method*), 56

F

fetchall() (*turbodbc.cursor.Cursor method*), 56
fetchallarrow() (*turbodbc.cursor.Cursor method*), 56
fetchallnumpy() (*turbodbc.cursor.Cursor method*), 56
fetcharrowbatches() (*turbodbc.cursor.Cursor method*), 56
fetchmany() (*turbodbc.cursor.Cursor method*), 56
fetchnumpybatches() (*turbodbc.cursor.Cursor method*), 57
fetchone() (*turbodbc.cursor.Cursor method*), 57

I

InterfaceError (*class in turbodbc.exceptions*), 57

M

make_options() (*in module turbodbc*), 53

P

ParameterError (*class in turbodbc.exceptions*), 57

R

rollback() (*turbodbc.connection.Connection method*), 55

S

setinputsizes() (*turbodbc.cursor.Cursor method*), 57
setoutputsize() (*turbodbc.cursor.Cursor method*), 57