asyncpg Documentation

Release 0.27.0

<See AUTHORS file>

Oct 29, 2022

CONTENTS

asyncpg is a database interface library designed specifically for PostgreSQL and Python/asyncio. asyncpg is an efficient, clean implementation of PostgreSQL server binary protocol for use with Python's `asyncio` framework.

asyncpg requires Python 3.7 or later and is supported for PostgreSQL versions 9.5 to 15. Older PostgreSQL versions or other databases implementing the PostgreSQL protocol *may* work, but are not being actively tested.

CHAPTER

ONE

CONTENTS

1.1 Installation

asyncpg has no external dependencies and the recommended way to install it is to use **pip**:

```
$ pip install asyncpg
```

Note: It is recommended to use **pip** version **8.1** or later to take advantage of the precompiled wheel packages. Older versions of pip will ignore the wheel packages and install asyncpg from the source package. In that case a working C compiler is required.

1.1.1 Building from source

If you want to build **asyncpg** from a Git checkout you will need:

- To have cloned the repo with *–recurse-submodules*.
- A working C compiler.
- CPython header files. These can usually be obtained by installing the relevant Python development package: **python3-dev** on Debian/Ubuntu, **python3-devel** on RHEL/Fedora.

Once the above requirements are satisfied, run the following command in the root of the source checkout:

```
$ pip install -e .
```

A debug build containing more runtime checks can be created by setting the `ASYNCPG_DEBUG` environment variable when building:

```
$ env ASYNCPG_DEBUG=1 pip install -e .
```

1.1.2 Running tests

If you want to run tests you must have PostgreSQL installed.

To execute the testsuite run:

```
$ python setup.py test
```

1.2 asyncpg Usage

The interaction with the database normally starts with a call to *connect()*, which establishes a new database session and returns a new *Connection* instance, which provides methods to run queries and manage transactions.

```
import asyncio
import asyncpg
import datetime

async def main():
    # Establish a connection to an existing database named "test"
    # as a "postgres" user.
    conn = await asyncpg.connect('postgresql://postgres@localhost/test')
    # Execute a statement to create a new table.
    await conn.execute('''
        CREATE TABLE users(
            id serial PRIMARY KEY,
            name text,
            dob date
        )
    ''')

    # Insert a record into the created table.
    await conn.execute('''
        INSERT INTO users(name, dob) VALUES($1, $2)
    ''', 'Bob', datetime.date(1984, 3, 1))

    # Select a row from the table.
    row = await conn.fetchrow(
        'SELECT * FROM users WHERE name = $1', 'Bob')
    # *row* now contains
    # asyncpg.Record(id=1, name='Bob', dob=datetime.date(1984, 3, 1))

    # Close the connection.
    await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Note: asyncpg uses the native PostgreSQL syntax for query arguments: `$n`.

1.2.1 Type Conversion

asyncpg automatically converts PostgreSQL types to the corresponding Python types and vice versa. All standard data types are supported out of the box, including arrays, composite types, range types, enumerations and any combination of them. It is possible to supply codecs for non-standard types or override standard codecs. See *Custom Type Conversions* for more information.

The table below shows the correspondence between PostgreSQL and Python types.

PostgreSQL Type	Python Type
`anyarray`	`list`
`anyenum`	`str`
`anyrange`	*`asyncpg.Range`*, `tuple`
`anymultirange`	`list[`*`asyncpg.Range`*`]`, `list[tuple]`[1]
`record`	*`asyncpg.Record`*, `tuple`, `Mapping`
`bit`, `varbit`	*`asyncpg.BitString`*
`bool`	`bool`
`box`	*`asyncpg.Box`*
`bytea`	`bytes`
`char`, `name`, `varchar`, `text`, `xml`	`str`
`cidr`	`ipaddress.IPv4Network`, `ipaddress.IPv6Network`
`inet`	`ipaddress.IPv4Interface`, `ipaddress.IPv6Interface`, `ipaddress.IPv4Address`, `ipaddress.IPv6Address`[2]
`macaddr`	`str`
`circle`	*`asyncpg.Circle`*
`date`	`datetime.date`
`time`	offset-naïve `datetime.time`
`time with time zone`	offset-aware `datetime.time`
`timestamp`	offset-naïve `datetime.datetime`
`timestamp with time zone`	offset-aware `datetime.datetime`
`interval`	`datetime.timedelta`
`float`, `double precision`	`float`[3]
`smallint`, `integer`, `bigint`	`int`
`numeric`	`Decimal`
`json`, `jsonb`	`str`
`line`	*`asyncpg.Line`*
`lseg`	*`asyncpg.LineSegment`*
`money`	`str`
`path`	*`asyncpg.Path`*
`point`	*`asyncpg.Point`*
`polygon`	*`asyncpg.Polygon`*
`uuid`	`uuid.UUID`
`tid`	`tuple`

All other types are encoded and decoded as text by default.

[1] Since version 0.25.0

[2] Prior to version 0.20.0, asyncpg erroneously treated `inet` values with prefix as `IPvXNetwork` instead of `IPvXInterface`.

[3] Inexact single-precision `float` values may have a different representation when decoded into a Python float. This is inherent to the implementation of limited-precision floating point types. If you need the decimal representation to match, cast the expression to `double` or `numeric` in your query.

1.2.2 Custom Type Conversions

asyncpg allows defining custom type conversion functions both for standard and user-defined types using the *Connection.set_type_codec()* and *Connection.set_builtin_type_codec()* methods.

Example: automatic JSON conversion

The example below shows how to configure asyncpg to encode and decode JSON values using the `json` module.

```
import asyncio
import asyncpg
import json


async def main():
    conn = await asyncpg.connect()

    try:
        await conn.set_type_codec(
            'json',
            encoder=json.dumps,
            decoder=json.loads,
            schema='pg_catalog'
        )

        data = {'foo': 'bar', 'spam': 1}
        res = await conn.fetchval('SELECT $1::json', data)

    finally:
        await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Example: automatic conversion of PostGIS types

The example below shows how to configure asyncpg to encode and decode the PostGIS `geometry` type. It works for any Python object that conforms to the geo interface specification and relies on Shapely, although any library that supports reading and writing the WKB format will work.

```
import asyncio
import asyncpg

import shapely.geometry
import shapely.wkb
from shapely.geometry.base import BaseGeometry


async def main():
    conn = await asyncpg.connect()

    try:
        def encode_geometry(geometry):
```

(continues on next page)

(continued from previous page)

```
            if not hasattr(geometry, '__geo_interface__'):
                raise TypeError('{g} does not conform to '
                                'the geo interface'.format(g=geometry))
            shape = shapely.geometry.asShape(geometry)
            return shapely.wkb.dumps(shape)

        def decode_geometry(wkb):
            return shapely.wkb.loads(wkb)

        await conn.set_type_codec(
            'geometry',  # also works for 'geography'
            encoder=encode_geometry,
            decoder=decode_geometry,
            format='binary',
        )

        data = shapely.geometry.Point(-73.985661, 40.748447)
        res = await conn.fetchrow(
            '''SELECT 'Empire State Building' AS name,
                      $1::geometry            AS coordinates
            ''',
            data)

        print(res)

    finally:
        await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Example: decoding numeric columns as floats

By default asyncpg decodes numeric columns as Python `Decimal` instances. The example below shows how to instruct asyncpg to use floats instead.

```
import asyncio
import asyncpg


async def main():
    conn = await asyncpg.connect()

    try:
        await conn.set_type_codec(
            'numeric', encoder=str, decoder=float,
            schema='pg_catalog', format='text'
        )

        res = await conn.fetchval("SELECT $1::numeric", 11.123)
        print(res, type(res))

```

(continues on next page)

(continued from previous page)

```
    finally:
        await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Example: decoding hstore values

hstore is an extension data type used for storing key/value pairs. asyncpg includes a codec to decode and encode hstore values as `dict` objects. Because `hstore` is not a builtin type, the codec must be registered on a connection using *`Connection.set_builtin_type_codec()`*:

```
import asyncpg
import asyncio

async def run():
    conn = await asyncpg.connect()
    # Assuming the hstore extension exists in the public schema.
    await conn.set_builtin_type_codec(
        'hstore', codec_name='pg_contrib.hstore')
    result = await conn.fetchval("SELECT 'a=>1,b=>2,c=>NULL'::hstore")
    assert result == {'a': '1', 'b': '2', 'c': None}

asyncio.get_event_loop().run_until_complete(run())
```

1.2.3 Transactions

To create transactions, the *`Connection.transaction()`* method should be used.

The most common way to use transactions is through an `async with` statement:

```
async with connection.transaction():
    await connection.execute("INSERT INTO mytable VALUES(1, 2, 3)")
```

Note: When not in an explicit transaction block, any changes to the database will be applied immediately. This is also known as *auto-commit*.

See the *Transactions* API documentation for more information.

1.2.4 Connection Pools

For server-type type applications, that handle frequent requests and need the database connection for a short period time while handling a request, the use of a connection pool is recommended. asyncpg provides an advanced pool implementation, which eliminates the need to use an external connection pooler such as PgBouncer.

To create a connection pool, use the *`asyncpg.create_pool()`* function. The resulting *`Pool`* object can then be used to borrow connections from the pool.

Below is an example of how **asyncpg** can be used to implement a simple Web service that computes the requested power of two.

```python
import asyncio
import asyncpg
from aiohttp import web


async def handle(request):
    """Handle incoming requests."""
    pool = request.app['pool']
    power = int(request.match_info.get('power', 10))

    # Take a connection from the pool.
    async with pool.acquire() as connection:
        # Open a transaction.
        async with connection.transaction():
            # Run the query passing the request argument.
            result = await connection.fetchval('select 2 ^ $1', power)
            return web.Response(
                text="2 ^ {} is {}".format(power, result))


async def init_app():
    """Initialize the application server."""
    app = web.Application()
    # Create a database connection pool
    app['pool'] = await asyncpg.create_pool(database='postgres',
                                            user='postgres')
    # Configure service routes
    app.router.add_route('GET', '/{power:\d+}', handle)
    app.router.add_route('GET', '/', handle)
    return app


loop = asyncio.get_event_loop()
app = loop.run_until_complete(init_app())
web.run_app(app)
```

See *Connection Pools* API documentation for more information.

1.3 API Reference

1.3.1 Connection

`async connect`(*dsn=None*, ***, *host=None*, *port=None*, *user=None*, *password=None*, *passfile=None*, *database=None*, *loop=None*, *timeout=60*, *statement_cache_size=100*, *max_cached_statement_lifetime=300*, *max_cacheable_statement_size=15360*, *command_timeout=None*, *ssl=None*, *direct_tls=False*, *connection_class=<class 'asyncpg.connection.Connection'>*, *record_class=<class 'asyncpg.Record'>*, *server_settings=None*)

A coroutine to establish a connection to a PostgreSQL server.

The connection parameters may be specified either as a connection URI in *dsn*, or as specific keyword arguments, or both. If both *dsn* and keyword arguments are specified, the latter override the corresponding values parsed from the connection URI. The default values for the majority of arguments can be specified using environment variables.

Returns a new `Connection` object.

Parameters

- **`dsn`** – Connection arguments specified using as a single string in the libpq connection URI format: `postgres://user:password@host:port/database?option=value`. The following options are recognized by asyncpg: `host`, `port`, `user`, `database` (or `dbname`), `password`, `passfile`, `sslmode`, `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl`. Unlike libpq, asyncpg will treat unrecognized options as server settings to be used for the connection.

  Note: The URI must be *valid*, which means that all components must be properly quoted with `urllib.parse.quote()`, and any literal IPv6 addresses must be enclosed in square brackets. For example:

  ```
  postgres://dbuser@[fe80::1ff:fe23:4567:890a%25eth0]/dbname
  ```

- **`host`** – Database host address as one of the following:
  - an IP address or a domain name;
  - an absolute path to the directory containing the database server Unix-domain socket (not supported on Windows);
  - a sequence of any of the above, in which case the addresses will be tried in order, and the first successful connection will be returned.

  If not specified, asyncpg will try the following, in order:
  - host address(es) parsed from the *dsn* argument,
  - the value of the `PGHOST` environment variable,
  - on Unix, common directories used for PostgreSQL Unix-domain sockets: `"/run/postgresql"`, `"/var/run/postgresl"`, `"/var/pgsql_socket"`, `"/private/tmp"`, and `"/tmp"`,
  - `"localhost"`.

- **`port`** – Port number to connect to at the server host (or Unix-domain socket file extension). If multiple host addresses were specified, this parameter may specify a sequence of port numbers of the same length as the host sequence, or it may specify a single port number to be used for all host addresses.

  If not specified, the value parsed from the *dsn* argument is used, or the value of the `PGPORT` environment variable, or `5432` if neither is specified.

- **`user`** – The name of the database role used for authentication.

  If not specified, the value parsed from the *dsn* argument is used, or the value of the `PGUSER` environment variable, or the operating system name of the user running the application.

- **`database`** – The name of the database to connect to.

  If not specified, the value parsed from the *dsn* argument is used, or the value of the `PGDATABASE` environment variable, or the computed value of the *user* argument.

- **password** – Password to be used for authentication, if the server requires one. If not specified, the value parsed from the *dsn* argument is used, or the value of the `PGPASSWORD` environment variable. Note that the use of the environment variable is discouraged as other users and applications may be able to read it without needing specific privileges. It is recommended to use *passfile* instead.

  Password may be either a string, or a callable that returns a string. If a callable is provided, it will be called each time a new connection is established.

- **passfile** – The name of the file used to store passwords (defaults to `~/.pgpass`, or `%APPDATA%\postgresql\pgpass.conf` on Windows).
- **loop** – An asyncio event loop instance. If `None`, the default event loop will be used.
- **timeout** (*float*) – Connection timeout in seconds.
- **statement_cache_size** (*int*) – The size of prepared statement LRU cache. Pass `0` to disable the cache.
- **max_cached_statement_lifetime** (*int*) – The maximum time in seconds a prepared statement will stay in the cache. Pass `0` to allow statements be cached indefinitely.
- **max_cacheable_statement_size** (*int*) – The maximum size of a statement that can be cached (15KiB by default). Pass `0` to allow all statements to be cached regardless of their size.
- **command_timeout** (*float*) – The default timeout for operations on this connection (the default is `None`: no timeout).
- **ssl** – Pass `True` or an ssl.SSLContext instance to require an SSL connection. If `True`, a default SSL context returned by ssl.create_default_context() will be used. The value can also be one of the following strings:
  - `'disable'` - SSL is disabled (equivalent to `False`)
  - `'prefer'` - try SSL first, fallback to non-SSL connection if SSL connection fails
  - `'allow'` - try without SSL first, then retry with SSL if the first attempt fails.
  - `'require'` - only try an SSL connection. Certificate verification errors are ignored
  - `'verify-ca'` - only try an SSL connection, and verify that the server certificate is issued by a trusted certificate authority (CA)
  - `'verify-full'` - only try an SSL connection, verify that the server certificate is issued by a trusted CA and that the requested server host name matches that in the certificate.

  The default is `'prefer'`: try an SSL connection and fallback to non-SSL connection if that fails.

  Note: *ssl* is ignored for Unix domain socket communication.

  Example of programmatic SSL context configuration that is equivalent to `sslmode=verify-full&sslcert=..&sslkey=..&sslrootcert=..`:

```
>>> import asyncpg
>>> import asyncio
>>> import ssl
>>> async def main():
...     # Load CA bundle for server certificate verification,
```

(continues on next page)

(continued from previous page)

```
...     # equivalent to sslrootcert= in DSN.
...     sslctx = ssl.create_default_context(
...         ssl.Purpose.SERVER_AUTH,
...         cafile="path/to/ca_bundle.pem")
...     # If True, equivalent to sslmode=verify-full, if False:
...     # sslmode=verify-ca.
...     sslctx.check_hostname = True
...     # Load client certificate and private key for client
...     # authentication, equivalent to sslcert= and sslkey= in
...     # DSN.
...     sslctx.load_cert_chain(
...         "path/to/client.cert",
...         keyfile="path/to/client.key",
...     )
...     con = await asyncpg.connect(user='postgres', ssl=sslctx)
...     await con.close()
>>> asyncio.run(run())
```

Example of programmatic SSL context configuration that is equivalent to `sslmode=require` (no server certificate or host verification):

```
>>> import asyncpg
>>> import asyncio
>>> import ssl
>>> async def main():
...     sslctx = ssl.create_default_context(
...         ssl.Purpose.SERVER_AUTH)
...     sslctx.check_hostname = False
...     sslctx.verify_mode = ssl.CERT_NONE
...     con = await asyncpg.connect(user='postgres', ssl=sslctx)
...     await con.close()
>>> asyncio.run(run())
```

- **direct_tls** (*bool*) – Pass `True` to skip PostgreSQL STARTTLS mode and perform a direct SSL connection. Must be used alongside `ssl` param.
- **server_settings** (*dict*) – An optional dict of server runtime parameters. Refer to PostgreSQL documentation for a list of supported options.
- **connection_class** (*type*) – Class of the returned connection object. Must be a subclass of *Connection*.
- **record_class** (*type*) – If specified, the class to use for records returned by queries on this connection object. Must be a subclass of *Record*.

Returns
A *Connection* instance.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     types = await con.fetch('SELECT * FROM pg_type')
```

(continues on next page)

(continued from previous page)

```
...     print(types)
...
>>> asyncio.get_event_loop().run_until_complete(run())
[<Record typname='bool' typnamespace=11 ...
```

New in version 0.10.0: Added `max_cached_statement_use_count` parameter.

Changed in version 0.11.0: Removed ability to pass arbitrary keyword arguments to set server settings. Added a dedicated parameter `server_settings` for that.

New in version 0.11.0: Added `connection_class` parameter.

New in version 0.16.0: Added `passfile` parameter (and support for password files in general).

New in version 0.18.0: Added ability to specify multiple hosts in the *dsn* and *host* arguments.

Changed in version 0.21.0: The *password* argument now accepts a callable or an async function.

Changed in version 0.22.0: Added the *record_class* parameter.

Changed in version 0.22.0: The *ssl* argument now defaults to `'prefer'`.

Changed in version 0.24.0: The `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl` options are supported in the *dsn* argument.

Changed in version 0.25.0: The `sslpassword`, `ssl_min_protocol_version`, and `ssl_max_protocol_version` options are supported in the *dsn* argument.

Changed in version 0.25.0: Default system root CA certificates won't be loaded when specifying a particular sslmode, following the same behavior in libpq.

Changed in version 0.25.0: The `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl` options in the *dsn* argument now have consistent default values of files under `~/.postgresql/` as libpq.

Changed in version 0.26.0: Added the *direct_tls* parameter.

class Connection(*protocol*, *transport*, *loop*, *addr*, *config: ConnectionConfiguration*, *params: ConnectionParameters*)

A representation of a database session.

Connections are created by calling `connect()`.

coroutine add_listener(*channel*, *callback*)

Add a listener for Postgres notifications.

Parameters

- **channel** (*str*) – Channel to listen on.
- **callback** (*callable*) – A callable or a coroutine function receiving the following arguments: **connection**: a Connection the callback is registered with; **pid**: PID of the Postgres server that sent the notification; **channel**: name of the channel the notification was sent to; **payload**: the payload.

Changed in version 0.24.0: The `callback` argument may be a coroutine function.

add_log_listener(*callback*)

Add a listener for Postgres log messages.

It will be called when asyncronous NoticeResponse is received from the connection. Possible message types are: WARNING, NOTICE, DEBUG, INFO, or LOG.

Parameters
`callback` (*callable*) – A callable or a coroutine function receiving the following arguments: **connection**: a Connection the callback is registered with; **message**: the *exceptions.PostgresLogMessage* message.

New in version 0.12.0.

Changed in version 0.24.0: The `callback` argument may be a coroutine function.

`add_termination_listener`(*callback*)

Add a listener that will be called when the connection is closed.

Parameters
`callback` (*callable*) – A callable or a coroutine function receiving one argument: **connection**: a Connection the callback is registered with.

New in version 0.21.0.

Changed in version 0.24.0: The `callback` argument may be a coroutine function.

`coroutine close`(**, timeout=None*)

Close the connection gracefully.

Parameters
`timeout` (*float*) – Optional timeout value in seconds.

Changed in version 0.14.0: Added the *timeout* parameter.

`coroutine copy_from_query`(*query, *args, output, timeout=None, format=None, oids=None, delimiter=None, null=None, header=None, quote=None, escape=None, force_quote=None, encoding=None*)

Copy the results of a query to a file or file-like object.

Parameters

- **query** (*str*) – The query to copy the results of.
- **args** – Query arguments.
- **output** – A path-like object, or a file-like object, or a coroutine function that takes a `bytes` instance as a sole argument.
- **timeout** (*float*) – Optional timeout value in seconds.

The remaining keyword arguments are `COPY` statement options, see COPY statement documentation for details.

Returns
The status string of the COPY command.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     result = await con.copy_from_query(
...         'SELECT foo, bar FROM mytable WHERE foo > $1', 10,
...         output='file.csv', format='csv')
...     print(result)
...
```

(continues on next page)

(continued from previous page)

```
>>> asyncio.get_event_loop().run_until_complete(run())
'COPY 10'
```

New in version 0.11.0.

coroutine copy_from_table(*table_name*, ***, *output*, *columns=None*, *schema_name=None*, *timeout=None*, *format=None*, *oids=None*, *delimiter=None*, *null=None*, *header=None*, *quote=None*, *escape=None*, *force_quote=None*, *encoding=None*)

Copy table contents to a file or file-like object.

Parameters

- **table_name** (*str*) – The name of the table to copy data from.
- **output** – A path-like object, or a file-like object, or a coroutine function that takes a `bytes` instance as a sole argument.
- **columns** (*list*) – An optional list of column names to copy.
- **schema_name** (*str*) – An optional schema name to qualify the table.
- **timeout** (*float*) – Optional timeout value in seconds.

The remaining keyword arguments are `COPY` statement options, see COPY statement documentation for details.

Returns
The status string of the COPY command.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     result = await con.copy_from_table(
...         'mytable', columns=('foo', 'bar'),
...         output='file.csv', format='csv')
...     print(result)
...
>>> asyncio.get_event_loop().run_until_complete(run())
'COPY 100'
```

New in version 0.11.0.

coroutine copy_records_to_table(*table_name*, ***, *records*, *columns=None*, *schema_name=None*, *timeout=None*)

Copy a list of records to the specified table using binary COPY.

Parameters

- **table_name** (*str*) – The name of the table to copy data to.
- **records** – An iterable returning row tuples to copy into the table. Asynchronous iterables are also supported.
- **columns** (*list*) – An optional list of column names to copy.
- **schema_name** (*str*) – An optional schema name to qualify the table.
- **timeout** (*float*) – Optional timeout value in seconds.

Returns
The status string of the COPY command.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     result = await con.copy_records_to_table(
...         'mytable', records=[
...             (1, 'foo', 'bar'),
...             (2, 'ham', 'spam')])
...     print(result)
...
>>> asyncio.get_event_loop().run_until_complete(run())
'COPY 2'
```

Asynchronous record iterables are also supported:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     async def record_gen(size):
...         for i in range(size):
...             yield (i,)
...     result = await con.copy_records_to_table(
...         'mytable', records=record_gen(100))
...     print(result)
...
>>> asyncio.get_event_loop().run_until_complete(run())
'COPY 100'
```

New in version 0.11.0.

Changed in version 0.24.0: The `records` argument may be an asynchronous iterable.

coroutine copy_to_table(*table_name*, ***, *source*, *columns=None*, *schema_name=None*, *timeout=None*, *format=None*, *oids=None*, *freeze=None*, *delimiter=None*, *null=None*, *header=None*, *quote=None*, *escape=None*, *force_quote=None*, *force_not_null=None*, *force_null=None*, *encoding=None*)

Copy data to the specified table.

Parameters

- **table_name** (*str*) – The name of the table to copy data to.
- **source** – A path-like object, or a file-like object, or an asynchronous iterable that returns `bytes`, or an object supporting the buffer protocol.
- **columns** (*list*) – An optional list of column names to copy.
- **schema_name** (*str*) – An optional schema name to qualify the table.
- **timeout** (*float*) – Optional timeout value in seconds.

The remaining keyword arguments are `COPY` statement options, see COPY statement documentation for details.

Returns
The status string of the COPY command.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     result = await con.copy_to_table(
...         'mytable', source='datafile.tbl')
...     print(result)
...
>>> asyncio.get_event_loop().run_until_complete(run())
'COPY 140000'
```

New in version 0.11.0.

cursor(*query*, **args*, *prefetch=None*, *timeout=None*, *record_class=None*)

Return a *cursor factory* for the specified query.

Parameters

- **args** – Query arguments.
- **prefetch** (*int*) – The number of rows the *cursor iterator* will prefetch (defaults to `50`.)
- **timeout** (*float*) – Optional timeout in seconds.
- **record_class** (*type*) – If specified, the class to use for records returned by this cursor. Must be a subclass of *Record*. If not specified, a per-connection *record_class* is used.

Returns
A `CursorFactory` object.

Changed in version 0.22.0: Added the *record_class* parameter.

coroutine execute(*query: str*, **args*, *timeout: float = None*) → str

Execute an SQL command (or commands).

This method can execute many SQL commands at once, when no arguments are provided.

Example:

```
>>> await con.execute('''
...     CREATE TABLE mytab (a int);
...     INSERT INTO mytab (a) VALUES (100), (200), (300);
... ''')
INSERT 0 3

>>> await con.execute('''
...     INSERT INTO mytab (a) VALUES ($1), ($2)
... ''', 10, 20)
INSERT 0 2
```

Parameters

- **args** – Query arguments.
- **timeout** (*float*) – Optional timeout value in seconds.

Return str
Status of the last SQL command.

Changed in version 0.5.4: Made it possible to pass query arguments.

coroutine executemany(*command: str*, *args*, *, *timeout: float = None*)

Execute an SQL *command* for each sequence of arguments in *args*.

Example:

```
>>> await con.executemany('''
...     INSERT INTO mytab (a) VALUES ($1, $2, $3);
... ''', [(1, 2, 3), (4, 5, 6)])
```

Parameters

- **command** – Command to execute.
- **args** – An iterable containing sequences of arguments.
- **timeout** (*float*) – Optional timeout value in seconds.

Return None
This method discards the results of the operations.

New in version 0.7.0.

Changed in version 0.11.0: *timeout* became a keyword-only parameter.

Changed in version 0.22.0: `executemany()` is now an atomic operation, which means that either all executions succeed, or none at all. This is in contrast to prior versions, where the effect of already-processed iterations would remain in place when an error has occurred, unless `executemany()` was called in a transaction.

coroutine fetch(*query*, **args*, *timeout=None*, *record_class=None*) → list

Run a query and return the results as a list of `Record`.

Parameters

- **query** (*str*) – Query text.
- **args** – Query arguments.
- **timeout** (*float*) – Optional timeout value in seconds.
- **record_class** (*type*) – If specified, the class to use for records returned by this method. Must be a subclass of *Record*. If not specified, a per-connection *record_class* is used.

Return list
A list of *Record* instances. If specified, the actual type of list elements would be *record_class*.

Changed in version 0.22.0: Added the *record_class* parameter.

coroutine fetchrow(*query*, **args*, *timeout=None*, *record_class=None*)

Run a query and return the first row.

Parameters

- **query** (*str*) – Query text
- **args** – Query arguments
- **timeout** (*float*) – Optional timeout value in seconds.

- **record_class** (*type*) – If specified, the class to use for the value returned by this method. Must be a subclass of `Record`. If not specified, a per-connection *record_class* is used.

Returns
The first row as a `Record` instance, or None if no records were returned by the query. If specified, *record_class* is used as the type for the result value.

Changed in version 0.22.0: Added the *record_class* parameter.

coroutine fetchval(*query*, **args*, *column=0*, *timeout=None*)

Run a query and return a value in the first row.

Parameters

- **query** (*str*) – Query text.
- **args** – Query arguments.
- **column** (*int*) – Numeric index within the record of the value to return (defaults to 0).
- **timeout** (*float*) – Optional timeout value in seconds. If not specified, defaults to the value of `command_timeout` argument to the `Connection` instance constructor.

Returns
The value of the specified column of the first record, or None if no records were returned by the query.

get_server_pid()

Return the PID of the Postgres server the connection is bound to.

get_server_version()

Return the version of the connected PostgreSQL server.

The returned value is a named tuple similar to that in `sys.version_info`:

```
>>> con.get_server_version()
ServerVersion(major=9, minor=6, micro=1,
              releaselevel='final', serial=0)
```

New in version 0.8.0.

get_settings()

Return connection settings.

Returns
`ConnectionSettings`.

is_closed()

Return `True` if the connection is closed, `False` otherwise.

Return bool
`True` if the connection is closed, `False` otherwise.

is_in_transaction()

Return True if Connection is currently inside a transaction.

Return bool
True if inside transaction, False otherwise.

New in version 0.16.0.

coroutine prepare(*query*, ***, *name=None*, *timeout=None*, *record_class=None*)

Create a *prepared statement* for the specified query.

Parameters

- **query** (*str*) – Text of the query to create a prepared statement for.
- **name** (*str*) – Optional name of the returned prepared statement. If not specified, the name is auto-generated.
- **timeout** (*float*) – Optional timeout value in seconds.
- **record_class** (*type*) – If specified, the class to use for records returned by the prepared statement. Must be a subclass of `Record`. If not specified, a per-connection *record_class* is used.

Returns

A `PreparedStatement` instance.

Changed in version 0.22.0: Added the *record_class* parameter.

Changed in version 0.25.0: Added the *name* parameter.

coroutine reload_schema_state()

Indicate that the database schema information must be reloaded.

For performance reasons, asyncpg caches certain aspects of the database schema, such as the layout of composite types. Consequently, when the database schema changes, and asyncpg is not able to gracefully recover from an error caused by outdated schema assumptions, an `OutdatedSchemaCacheError` is raised. To prevent the exception, this method may be used to inform asyncpg that the database schema has changed.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> async def change_type(con):
...     result = await con.fetch('SELECT id, info FROM tbl')
...     # Change composite's attribute type "int"=>"text"
...     await con.execute('ALTER TYPE custom DROP ATTRIBUTE y')
...     await con.execute('ALTER TYPE custom ADD ATTRIBUTE y text')
...     await con.reload_schema_state()
...     for id_, info in result:
...         new = (info['x'], str(info['y']))
...         await con.execute(
...             'UPDATE tbl SET info=$2 WHERE id=$1', id_, new)
...
>>> async def run():
...     # Initial schema:
...     # CREATE TYPE custom AS (x int, y int);
...     # CREATE TABLE tbl(id int, info custom);
...     con = await asyncpg.connect(user='postgres')
...     async with con.transaction():
...         # Prevent concurrent changes in the table
...         await con.execute('LOCK TABLE tbl')
...         await change_type(con)
...
>>> asyncio.get_event_loop().run_until_complete(run())
```

New in version 0.14.0.

coroutine remove_listener(*channel*, *callback*)

Remove a listening callback on the specified channel.

remove_log_listener(*callback*)

Remove a listening callback for log messages.

New in version 0.12.0.

remove_termination_listener(*callback*)

Remove a listening callback for connection termination.

Parameters

callback (*callable*) – The callable or coroutine function that was passed to `Connection.add_termination_listener()`.

New in version 0.21.0.

coroutine reset_type_codec(*typename*, *, *schema='public'*)

Reset *typename* codec to the default implementation.

Parameters

- **typename** – Name of the data type the codec is for.
- **schema** – Schema name of the data type the codec is for (defaults to `'public'`)

New in version 0.12.0.

coroutine set_builtin_type_codec(*typename*, *, *schema='public'*, *codec_name*, *format=None*)

Set a builtin codec for the specified scalar data type.

This method has two uses. The first is to register a builtin codec for an extension type without a stable OID, such as ‘hstore’. The second use is to declare that an extension type or a user-defined type is wire-compatible with a certain builtin data type and should be exchanged as such.

Parameters

- **typename** – Name of the data type the codec is for.
- **schema** – Schema name of the data type the codec is for (defaults to `'public'`).
- **codec_name** – The name of the builtin codec to use for the type. This should be either the name of a known core type (such as `"int"`), or the name of a supported extension type. Currently, the only supported extension type is `"pg_contrib.hstore"`.
- **format** – If *format* is `None` (the default), all formats supported by the target codec are declared to be supported for *typename*. If *format* is `'text'` or `'binary'`, then only the specified format is declared to be supported for *typename*.

Changed in version 0.18.0: The *codec_name* argument can be the name of any known core data type. Added the *format* keyword argument.

coroutine set_type_codec(*typename*, *, *schema='public'*, *encoder*, *decoder*, *format='text'*)

Set an encoder/decoder pair for the specified data type.

Parameters

- **typename** – Name of the data type the codec is for.
- **schema** – Schema name of the data type the codec is for (defaults to `'public'`)
- **format** – The type of the argument received by the *decoder* callback, and the type of the *encoder* callback return value.

If *format* is `'text'` (the default), the exchange datum is a `str` instance containing valid text representation of the data type.

If *format* is `'binary'`, the exchange datum is a `bytes` instance containing valid _binary_ representation of the data type.

If *format* is `'tuple'`, the exchange datum is a type-specific `tuple` of values. The table below lists supported data types and their format for this mode.

Type	Tuple layout
`interval`	(`months`, `days`, `microseconds`)
`date`	(`date ordinal relative to Jan 1 2000`,) `-2^31` for negative infinity timestamp `2^31-1` for positive infinity timestamp.
`timestamp`	(`microseconds relative to Jan 1 2000`,) `-2^63` for negative infinity timestamp `2^63-1` for positive infinity timestamp.
`timestamp with time zone`	(`microseconds relative to Jan 1 2000 UTC`,) `-2^63` for negative infinity timestamp `2^63-1` for positive infinity timestamp.
`time`	(`microseconds`,)
`time with time zone`	(`microseconds`, `time zone offset in seconds`)

- **`encoder`** – Callable accepting a Python object as a single argument and returning a value encoded according to *format*.
- **`decoder`** – Callable accepting a single argument encoded according to *format* and returning a decoded Python object.

Example:

```
>>> import asyncpg
>>> import asyncio
>>> import datetime
>>> from dateutil.relativedelta import relativedelta
>>> async def run():
...     con = await asyncpg.connect(user='postgres')
...     def encoder(delta):
...         ndelta = delta.normalized()
...         return (ndelta.years * 12 + ndelta.months,
...                 ndelta.days,
...                 ((ndelta.hours * 3600 +
...                    ndelta.minutes * 60 +
...                    ndelta.seconds) * 1000000 +
...                  ndelta.microseconds))
...     def decoder(tup):
...         return relativedelta(months=tup[0], days=tup[1],
...                              microseconds=tup[2])
...     await con.set_type_codec(
...         'interval', schema='pg_catalog', encoder=encoder,
...         decoder=decoder, format='tuple')
...     result = await con.fetchval(
...         "SELECT '2 years 3 mons 1 day'::interval")
...     print(result)
...     print(datetime.datetime(2002, 1, 1) + result)
...
```

(continues on next page)

(continued from previous page)

```
>>> asyncio.get_event_loop().run_until_complete(run())
relativedelta(years=+2, months=+3, days=+1)
2004-04-02 00:00:00
```

New in version 0.12.0: Added the `format` keyword argument and support for 'tuple' format.

Changed in version 0.12.0: The `binary` keyword argument is deprecated in favor of `format`.

Changed in version 0.13.0: The `binary` keyword argument was removed in favor of `format`.

Note: It is recommended to use the `'binary'` or `'tuple'` *format* whenever possible and if the underlying type supports it. Asyncpg currently does not support text I/O for composite and range types, and some other functionality, such as `Connection.copy_to_table()`, does not support types with text codecs.

terminate()

Terminate the connection without waiting for pending data.

transaction(*, *isolation=None*, *readonly=False*, *deferrable=False*)

Create a `Transaction` object.

Refer to PostgreSQL documentation on the meaning of transaction parameters.

Parameters

- **isolation** – Transaction isolation mode, can be one of: *'serializable'*, *'repeatable_read'*, *'read_committed'*. If not specified, the behavior is up to the server and session, which is usually `read_committed`.
- **readonly** – Specifies whether or not this transaction is read-only.
- **deferrable** – Specifies whether or not this transaction is deferrable.

1.3.2 Prepared Statements

Prepared statements are a PostgreSQL feature that can be used to optimize the performance of queries that are executed more than once. When a query is *prepared* by a call to `Connection.prepare()`, the server parses, analyzes and compiles the query allowing to reuse that work once there is a need to run the same query again.

```
>>> import asyncpg, asyncio
>>> loop = asyncio.get_event_loop()
>>> async def run():
...     conn = await asyncpg.connect()
...     stmt = await conn.prepare('''SELECT 2 ^ $1''')
...     print(await stmt.fetchval(10))
...     print(await stmt.fetchval(20))
...
>>> loop.run_until_complete(run())
1024.0
1048576.0
```

Note: asyncpg automatically maintains a small LRU cache for queries executed during calls to the `fetch()`, `fetchrow()`, or `fetchval()` methods.

> **Warning:** If you are using pgbouncer with `pool_mode` set to `transaction` or `statement`, prepared statements will not work correctly. See *Why am I getting prepared statement errors?* for more information.

class **PreparedStatement**

A representation of a prepared statement.

cursor(**args*, *prefetch=None*, *timeout=None*) → *CursorFactory*

Return a *cursor factory* for the prepared statement.

Parameters

- **args** – Query arguments.
- **prefetch** (*int*) – The number of rows the *cursor iterator* will prefetch (defaults to `50`.)
- **timeout** (*float*) – Optional timeout in seconds.

Returns

A `CursorFactory` object.

executemany(*args*, *, *timeout: float = None*)

Execute the statement for each sequence of arguments in *args*.

Parameters

- **args** – An iterable containing sequences of arguments.
- **timeout** (*float*) – Optional timeout value in seconds.

Return None

This method discards the results of the operations.

New in version 0.22.0.

explain(**args*, *analyze=False*)

Return the execution plan of the statement.

Parameters

- **args** – Query arguments.
- **analyze** – If `True`, the statement will be executed and the run time statitics added to the return value.

Returns

An object representing the execution plan. This value is actually a deserialized JSON output of the SQL `EXPLAIN` command.

fetch(**args*, *timeout=None*)

Execute the statement and return a list of `Record` objects.

Parameters

- **query** (*str*) – Query text
- **args** – Query arguments
- **timeout** (*float*) – Optional timeout value in seconds.

Returns

A list of `Record` instances.

`fetchrow`(**args*, *timeout=None*)

Execute the statement and return the first row.

Parameters

- **`query`** (*`str`*) – Query text
- **`args`** – Query arguments
- **`timeout`** (*`float`*) – Optional timeout value in seconds.

Returns

The first row as a `Record` instance.

`fetchval`(**args*, *column=0*, *timeout=None*)

Execute the statement and return a value in the first row.

Parameters

- **`args`** – Query arguments.
- **`column`** (*`int`*) – Numeric index within the record of the value to return (defaults to 0).
- **`timeout`** (*`float`*) – Optional timeout value in seconds. If not specified, defaults to the value of `command_timeout` argument to the `Connection` instance constructor.

Returns

The value of the specified column of the first record.

`get_attributes`()

Return a description of relation attributes (columns).

Returns

A tuple of *`asyncpg.types.Attribute`*.

Example:

```
st = await self.con.prepare('''
    SELECT typname, typnamespace FROM pg_type
''')
print(st.get_attributes())

# Will print:
#   (Attribute(
#       name='typname',
#       type=Type(oid=19, name='name', kind='scalar',
#                 schema='pg_catalog')),
#    Attribute(
#       name='typnamespace',
#       type=Type(oid=26, name='oid', kind='scalar',
#                 schema='pg_catalog')))
```

`get_name`() → str

Return the name of this prepared statement.

New in version 0.25.0.

`get_parameters`()

Return a description of statement parameters types.

Returns

A tuple of *`asyncpg.types.Type`*.

Example:

```
stmt = await connection.prepare('SELECT ($1::int, $2::text)')
print(stmt.get_parameters())

# Will print:
#   (Type(oid=23, name='int4', kind='scalar', schema='pg_catalog'),
#    Type(oid=25, name='text', kind='scalar', schema='pg_catalog'))
```

get_query() → str

Return the text of the query for this prepared statement.

Example:

```
stmt = await connection.prepare('SELECT $1::int')
assert stmt.get_query() == "SELECT $1::int"
```

get_statusmsg() → str

Return the status of the executed command.

Example:

```
stmt = await connection.prepare('CREATE TABLE mytab (a int)')
await stmt.fetch()
assert stmt.get_statusmsg() == "CREATE TABLE"
```

1.3.3 Transactions

The most common way to use transactions is through an `async with` statement:

```
async with connection.transaction():
    await connection.execute("INSERT INTO mytable VALUES(1, 2, 3)")
```

asyncpg supports nested transactions (a nested transaction context will create a savepoint.):

```
async with connection.transaction():
    await connection.execute('CREATE TABLE mytab (a int)')

    try:
        # Create a nested transaction:
        async with connection.transaction():
            await connection.execute('INSERT INTO mytab (a) VALUES (1), (2)')
            # This nested transaction will be automatically rolled back:
            raise Exception
    except:
        # Ignore exception
        pass

    # Because the nested transaction was rolled back, there
    # will be nothing in `mytab`.
    assert await connection.fetch('SELECT a FROM mytab') == []
```

Alternatively, transactions can be used without an `async with` block:

```
tr = connection.transaction()
await tr.start()
try:
    ...
except:
    await tr.rollback()
    raise
else:
    await tr.commit()
```

See also the *Connection.transaction()* function.

class Transaction

Represents a transaction or savepoint block.

Transactions are created by calling the `Connection.transaction()` function.

async with c:

start and commit/rollback the transaction or savepoint block automatically when entering and exiting the code inside the context manager block.

commit()

Exit the transaction or savepoint block and commit changes.

rollback()

Exit the transaction or savepoint block and rollback changes.

start()

Enter the transaction or savepoint block.

1.3.4 Cursors

Cursors are useful when there is a need to iterate over the results of a large query without fetching all rows at once. The cursor interface provided by asyncpg supports *asynchronous iteration* via the `async for` statement, and also a way to read row chunks and skip forward over the result set.

To iterate over a cursor using a connection object use *Connection.cursor()*. To make the iteration efficient, the cursor will prefetch records to reduce the number of queries sent to the server:

```
async def iterate(con: Connection):
    async with con.transaction():
        # Postgres requires non-scrollable cursors to be created
        # and used in a transaction.
        async for record in con.cursor('SELECT generate_series(0, 100)'):
            print(record)
```

Or, alternatively, you can iterate over the cursor manually (cursor won't be prefetching any rows):

```
async def iterate(con: Connection):
    async with con.transaction():
        # Postgres requires non-scrollable cursors to be created
        # and used in a transaction.

        # Create a Cursor object
```

(continues on next page)

(continued from previous page)

```
        cur = await con.cursor('SELECT generate_series(0, 100)')

        # Move the cursor 10 rows forward
        await cur.forward(10)

        # Fetch one row and print it
        print(await cur.fetchrow())

        # Fetch a list of 5 rows and print it
        print(await cur.fetch(5))
```

It's also possible to create cursors from prepared statements:

```
async def iterate(con: Connection):
    # Create a prepared statement that will accept one argument
    stmt = await con.prepare('SELECT generate_series(0, $1)')

    async with con.transaction():
        # Postgres requires non-scrollable cursors to be created
        # and used in a transaction.

        # Execute the prepared statement passing `10` as the
        # argument -- that will generate a series or records
        # from 0..10.  Iterate over all of them and print every
        # record.
        async for record in stmt.cursor(10):
            print(record)
```

Note: Cursors created by a call to *`Connection.cursor()`* or *`PreparedStatement.cursor()`* are *non-scrollable*: they can only be read forwards. To create a scrollable cursor, use the `DECLARE ... SCROLL CURSOR` SQL statement directly.

Warning: Cursors created by a call to *`Connection.cursor()`* or *`PreparedStatement.cursor()`* cannot be used outside of a transaction. Any such attempt will result in `InterfaceError`.

To create a cursor usable outside of a transaction, use the `DECLARE ... CURSOR WITH HOLD` SQL statement directly.

class CursorFactory

A cursor interface for the results of a query.

A cursor interface can be used to initiate efficient traversal of the results of a large query.

async for row in c

Execute the statement and iterate over the results asynchronously.

await c

Execute the statement and return an instance of *`Cursor`* which can be used to navigate over and fetch subsets of the query results.

class Cursor

An open *portal* into the results of a query.

fetch(*n*, *, *timeout=None*)

Return the next *n* rows as a list of `Record` objects.

Parameters
`timeout` (*float*) – Optional timeout value in seconds.

Returns
A list of `Record` instances.

fetchrow(*, *timeout=None*)

Return the next row.

Parameters
`timeout` (*float*) – Optional timeout value in seconds.

Returns
A `Record` instance.

forward(*n*, *, *timeout=None*) → int

Skip over the next *n* rows.

Parameters
`timeout` (*float*) – Optional timeout value in seconds.

Returns
A number of rows actually skipped over (<= *n*).

1.3.5 Connection Pools

create_pool(*dsn=None*, *, *min_size=10*, *max_size=10*, *max_queries=50000*, *max_inactive_connection_lifetime=300.0*, *setup=None*, *init=None*, *loop=None*, *connection_class=<class 'asyncpg.connection.Connection'>*, *record_class=<class 'asyncpg.Record'>*, ***connect_kwargs*)

Create a connection pool.

Can be used either with an `async with` block:

```
async with asyncpg.create_pool(user='postgres',
                               command_timeout=60) as pool:
    await pool.fetch('SELECT 1')
```

Or to perform multiple operations on a single connection:

```
async with asyncpg.create_pool(user='postgres',
                               command_timeout=60) as pool:
    async with pool.acquire() as con:
        await con.execute('''
           CREATE TABLE names (
              id serial PRIMARY KEY,
              name VARCHAR (255) NOT NULL)
        ''')
        await con.fetch('SELECT 1')
```

Or directly with `await` (not recommended):

```
pool = await asyncpg.create_pool(user='postgres', command_timeout=60)
con = await pool.acquire()
try:
    await con.fetch('SELECT 1')
finally:
    await pool.release(con)
```

Warning: Prepared statements and cursors returned by *Connection.prepare()* and *Connection.cursor()* become invalid once the connection is released. Likewise, all notification and log listeners are removed, and `asyncpg` will issue a warning if there are any listener callbacks registered on a connection that is being released to the pool.

Parameters

- **dsn** (*str*) – Connection arguments specified using as a single string in the following format: `postgres://user:pass@host:port/database?option=value`.
- ****connect_kwargs** – Keyword arguments for the *connect()* function.
- **connection_class** (`Connection`) – The class to use for connections. Must be a subclass of *Connection*.
- **record_class** (*type*) – If specified, the class to use for records returned by queries on the connections in this pool. Must be a subclass of *Record*.
- **min_size** (*int*) – Number of connection the pool will be initialized with.
- **max_size** (*int*) – Max number of connections in the pool.
- **max_queries** (*int*) – Number of queries after a connection is closed and replaced with a new connection.
- **max_inactive_connection_lifetime** (*float*) – Number of seconds after which inactive connections in the pool will be closed. Pass `0` to disable this mechanism.
- **setup** (*coroutine*) – A coroutine to prepare a connection right before it is returned from `Pool.acquire()`. An example use case would be to automatically set up notifications listeners for all connections of a pool.
- **init** (*coroutine*) – A coroutine to initialize a connection when it is created. An example use case would be to setup type codecs with *Connection.set_builtin_type_codec()* or *Connection.set_type_codec()*.
- **loop** – An asyncio event loop instance. If `None`, the default event loop will be used.

Returns

An instance of *Pool*.

Changed in version 0.10.0: An `InterfaceError` will be raised on any attempted operation on a released connection.

Changed in version 0.13.0: An `InterfaceError` will be raised on any attempted operation on a prepared statement or a cursor created on a connection that has been released to the pool.

Changed in version 0.13.0: An `InterfaceWarning` will be produced if there are any active listeners (added via *Connection.add_listener()* or *Connection.add_log_listener()*) present on the connection at the moment of its release to the pool.

Changed in version 0.22.0: Added the *record_class* parameter.

class Pool

A connection pool.

Connection pool can be used to manage a set of connections to the database. Connections are first acquired from the pool, then used, and then released back to the pool. Once a connection is released, it's reset to close all open cursors and other resources *except* prepared statements.

Pools are created by calling `create_pool()`.

acquire(*, *timeout=None*)

Acquire a database connection from the pool.

Parameters
timeout (*float*) – A timeout for acquiring a Connection.

Returns
An instance of `Connection`.

Can be used in an `await` expression or with an `async with` block.

```
async with pool.acquire() as con:
    await con.execute(...)
```

Or:

```
con = await pool.acquire()
try:
    await con.execute(...)
finally:
    await pool.release(con)
```

coroutine close()

Attempt to gracefully close all connections in the pool.

Wait until all pool connections are released, close them and shut down the pool. If any error (including cancellation) occurs in `close()` the pool will terminate by calling `Pool.terminate()`.

It is advisable to use `python:asyncio.wait_for()` to set a timeout.

Changed in version 0.16.0: `close()` now waits until all pool connections are released before closing them and the pool. Errors raised in `close()` will cause immediate pool termination.

coroutine copy_from_query(*query*, **args*, *output*, *timeout=None*, *format=None*, *oids=None*, *delimiter=None*, *null=None*, *header=None*, *quote=None*, *escape=None*, *force_quote=None*, *encoding=None*)

Copy the results of a query to a file or file-like object.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.copy_from_query()`.

New in version 0.24.0.

coroutine copy_from_table(*table_name*, *, *output*, *columns=None*, *schema_name=None*, *timeout=None*, *format=None*, *oids=None*, *delimiter=None*, *null=None*, *header=None*, *quote=None*, *escape=None*, *force_quote=None*, *encoding=None*)

Copy table contents to a file or file-like object.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.copy_from_table()`.

New in version 0.24.0.

coroutine copy_records_to_table(*table_name*, ***, *records*, *columns=None*, *schema_name=None*, *timeout=None*)

Copy a list of records to the specified table using binary COPY.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.copy_records_to_table()`.

New in version 0.24.0.

coroutine copy_to_table(*table_name*, ***, *source*, *columns=None*, *schema_name=None*, *timeout=None*, *format=None*, *oids=None*, *freeze=None*, *delimiter=None*, *null=None*, *header=None*, *quote=None*, *escape=None*, *force_quote=None*, *force_not_null=None*, *force_null=None*, *encoding=None*)

Copy data to the specified table.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.copy_to_table()`.

New in version 0.24.0.

coroutine execute(*query: str*, **args*, *timeout: float = None*) → str

Execute an SQL command (or commands).

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.execute()`.

New in version 0.10.0.

coroutine executemany(*command: str*, *args*, ***, *timeout: float = None*)

Execute an SQL *command* for each sequence of arguments in *args*.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.executemany()`.

New in version 0.10.0.

coroutine expire_connections()

Expire all currently open connections.

Cause all currently open connections to get replaced on the next `acquire()` call.

New in version 0.16.0.

coroutine fetch(*query*, **args*, *timeout=None*, *record_class=None*) → list

Run a query and return the results as a list of `Record`.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.fetch()`.

New in version 0.10.0.

coroutine fetchrow(*query*, **args*, *timeout=None*, *record_class=None*)

Run a query and return the first row.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.fetchrow()`.

New in version 0.10.0.

coroutine fetchval(*query*, **args*, *column=0*, *timeout=None*)

Run a query and return a value in the first row.

Pool performs this operation using one of its connections. Other than that, it behaves identically to `Connection.fetchval()`.

New in version 0.10.0.

get_idle_size()

Return the current number of idle connections in this pool.

New in version 0.25.0.

get_max_size()

Return the maximum allowed number of connections in this pool.

New in version 0.25.0.

get_min_size()

Return the minimum number of connections in this pool.

New in version 0.25.0.

get_size()

Return the current number of connections in this pool.

New in version 0.25.0.

coroutine release(*connection*, ***, *timeout=None*)

Release a database connection back to the pool.

Parameters

- **connection** (Connection) – A `Connection` object to release.
- **timeout** (`float`) – A timeout for releasing the connection. If not specified, defaults to the timeout provided in the corresponding call to the `Pool.acquire()` method.

Changed in version 0.14.0: Added the *timeout* parameter.

set_connect_args(*dsn=None*, ***connect_kwargs*)

Set the new connection arguments for this pool.

The new connection arguments will be used for all subsequent new connection attempts. Existing connections will remain until they expire. Use `Pool.expire_connections()` to expedite the connection expiry.

Parameters

- **dsn** (`str`) – Connection arguments specified using as a single string in the following format: `postgres://user:pass@host:port/database?option=value`.
- ****connect_kwargs** – Keyword arguments for the `connect()` function.

New in version 0.16.0.

terminate()

Terminate all connections in the pool.

1.3.6 Record Objects

Each row (or composite type value) returned by calls to `fetch*` methods is represented by an instance of the *Record* object. `Record` objects are a tuple-/dict-like hybrid, and allow addressing of items either by a numeric index or by a field name:

```
>>> import asyncpg
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> conn = loop.run_until_complete(asyncpg.connect())
>>> r = loop.run_until_complete(conn.fetchrow('''
...     SELECT oid, rolname, rolsuper FROM pg_roles WHERE rolname = user'''))
>>> r
<Record oid=16388 rolname='elvis' rolsuper=True>
>>> r['oid']
16388
>>> r[0]
16388
>>> dict(r)
{'oid': 16388, 'rolname': 'elvis', 'rolsuper': True}
>>> tuple(r)
(16388, 'elvis', True)
```

Note: `Record` objects currently cannot be created from Python code.

class Record

A read-only representation of PostgreSQL row.

len(r)

Return the number of fields in record *r*.

r[field]

Return the field of *r* with field name or index *field*.

name in r

Return `True` if record *r* has a field named *name*.

iter(r)

Return an iterator over the *values* of the record *r*.

get(name[, default])

Return the value for *name* if the record has a field named *name*, else return *default*. If *default* is not given, return `None`.

New in version 0.18.

values()

Return an iterator over the record values.

keys()

Return an iterator over the record field names.

items()

Return an iterator over `(field, value)` pairs.

class ConnectionSettings

A read-only collection of Connection settings.

settings.setting_name

Return the value of the "setting_name" setting. Raises an `AttributeError` if the setting is not defined.

Example:

```
>>> connection.get_settings().client_encoding
'UTF8'
```

1.3.7 Data Types

class Attribute(*name*, *type*)

Database relation attribute.

name

Attribute name.

type

Attribute data type *asyncpg.types.Type*.

class BitString(*bitstring: Optional[bytes] = None*)

Immutable representation of PostgreSQL *bit* and *varbit* types.

classmethod from_int(*x: int*, *length: int*, *bitorder: Literal['big', 'little'] = 'big'*, ***, *signed: bool = False*) → _BitString

Represent the Python int x as a BitString. Acts similarly to int.to_bytes.

Parameters

- **x** (*int*) – An integer to represent. Negative integers are represented in two's complement form, unless the argument signed is False, in which case negative integers raise an OverflowError.
- **length** (*int*) – The length of the resulting BitString. An OverflowError is raised if the integer is not representable in this many bits.
- **bitorder** – Determines the bit order used in the BitString representation. By default, this function uses Postgres conventions for casting ints to bits. If bitorder is 'big', the most significant bit is at the start of the string (this is the same as the default). If bitorder is 'little', the most significant bit is at the end of the string.
- **signed** (*bool*) – Determines whether two's complement is used in the BitString representation. If signed is False and a negative integer is given, an OverflowError is raised.

Return BitString

A BitString representing the input integer, in the form specified by the other input args.

New in version 0.18.0.

to_int(*bitorder: Literal['big', 'little'] = 'big'*, ***, *signed: bool = False*) → int

Interpret the BitString as a Python int. Acts similarly to int.from_bytes.

Parameters

- `bitorder` – Determines the bit order used to interpret the BitString. By default, this function uses Postgres conventions for casting bits to ints. If bitorder is 'big', the most significant bit is at the start of the string (this is the same as the default). If bitorder is 'little', the most significant bit is at the end of the string.
- `signed` (*bool*) – Determines whether two's complement is used to interpret the BitString. If signed is False, the returned value is always non-negative.

Return int
An integer representing the BitString. Information about the BitString's exact length is lost.

New in version 0.18.0.

`class Box`(*high: Sequence[float]*, *low: Sequence[float]*)
Immutable representation of PostgreSQL *box* type.

`class Circle`(*center:* Point, *radius: float*)
Immutable representation of PostgreSQL *circle* type.

`class Line`(*A: float*, *B: float*, *C: float*)
Immutable representation of PostgreSQL *line* type.

`class LineSegment`(*p1: Sequence[float]*, *p2: Sequence[float]*)
Immutable representation of PostgreSQL *lseg* type.

`class Path`(**points: Sequence[float]*, *is_closed: bool = False*)
Immutable representation of PostgreSQL *path* type.

`class Point`(*x: Union[SupportsFloat, SupportsIndex, str, bytes, bytearray]*, *y: Union[SupportsFloat, SupportsIndex, str, bytes, bytearray]*)
Immutable representation of PostgreSQL *point* type.

`class Polygon`(**points: Sequence[float]*)
Immutable representation of PostgreSQL *polygon* type.

`class Range`(*lower=None*, *upper=None*, ***, *lower_inc=True*, *upper_inc=False*, *empty=False*)
Immutable representation of PostgreSQL *range* type.

`class ServerVersion`(*major*, *minor*, *micro*, *releaselevel*, *serial*)
PostgreSQL server version tuple.

`major`
Alias for field number 0

`micro`
Alias for field number 2

`minor`
Alias for field number 1

`releaselevel`
Alias for field number 3

`serial`
Alias for field number 4

`class Type`(*oid*, *name*, *kind*, *schema*)
Database data type.

kind
Type kind. Can be “scalar”, “array”, “composite” or “range”.

name
Type name. For example “int2”.

oid
OID of the type.

schema
Name of the database schema that defines the type.

1.4 Frequently Asked Questions

1.4.1 Does asyncpg support DB-API?

No. DB-API is a synchronous API, while asyncpg is based around an asynchronous I/O model. Thus, full drop-in compatibility with DB-API is not possible and we decided to design asyncpg API in a way that is better aligned with PostgreSQL architecture and terminology. We will release a synchronous DB-API-compatible version of asyncpg at some point in the future.

1.4.2 Can I use asyncpg with SQLAlchemy ORM?

Yes. SQLAlchemy version 1.4 and later supports the asyncpg dialect natively. Please refer to its documentation for details. Older SQLAlchemy versions may be used in tandem with a third-party adapter such as asyncpgsa or databases.

1.4.3 Can I use dot-notation with `asyncpg.Record`? It looks cleaner.

We decided against making *asyncpg.Record* a named tuple because we want to keep the `Record` method namespace separate from the column namespace. That said, you can provide a custom `Record` class that implements dot-notation via the `record_class` argument to *connect()* or any of the Record-returning methods.

```
class MyRecord(asyncpg.Record):
    def __getattr__(self, name):
        return self[name]
```

1.4.4 Why can’t I use a cursor outside of a transaction?

Cursors created by a call to *Connection.cursor()* or *PreparedStatement.cursor()* cannot be used outside of a transaction. Any such attempt will result in `InterfaceError`. To create a cursor usable outside of a transaction, use the `DECLARE ... CURSOR WITH HOLD` SQL statement directly.

1.4.5 Why am I getting prepared statement errors?

If you are getting intermittent `prepared statement "__asyncpg_stmt_xx__" does not exist` or `prepared statement “__asyncpg_stmt_xx__” already exists` errors, you are most likely not connecting to the PostgreSQL server directly, but via pgbouncer. pgbouncer, when in the `"transaction"` or `"statement"` pooling mode, does not support prepared statements. You have several options:

- if you are using pgbouncer only to reduce the cost of new connections (as opposed to using pgbouncer for connection pooling from a large number of clients in the interest of better scalability), switch to the *connection pool* functionality provided by asyncpg, it is a much better option for this purpose;
- disable automatic use of prepared statements by passing `statement_cache_size=0` to *`asyncpg.connect()`* and *`asyncpg.create_pool()`* (and, obviously, avoid the use of *`Connection.prepare()`*);
- switch pgbouncer's `pool_mode` to `session`.

1.4.6 Why do I get `PostgresSyntaxError` when using `expression IN $1`?

`expression IN $1` is not a valid PostgreSQL syntax. To check a value against a sequence use `expression = any($1::mytype[])`, where `mytype` is the array element type.

PYTHON MODULE INDEX

INDEX