pgcom

Release 0.2.9

viktorsapozhok

Apr 04, 2022

CONTENTS

Communication manager for PostgreSQL database, provides a collection of convenience wrappers over Psycopg adapter to simplify the usage of basic Psycopg methods in conjunction with Pandas DataFrames.

CHAPTER

ONE

KEY FEATURES

- Reading from database table to Pandas DataFrame.
- Writing from DataFrame to a table.
- Adaptation between DataFrames and COPY FROM.
- Methods to resolve conflicts in DataFrame before applying COPY FROM.
- Tools for setting asynchronous communication with database using LISTEN and NOTIFY commands.

1.1 Installation

Pgcom supports Python 3.6 or newer.

To install the package, you can simply use pip:

```
$ pip install pgcom
```

or clone the repository:

```
$ git clone git@github.com:viktorsapozhok/pgcom.git
$ cd pgcom
$ pip install .
```

To install the package in development mode with the possibility to run tests in local environment, use following:

```
$ git clone git@github.com:viktorsapozhok/pgcom.git
$ cd pgcom
$ pip install --no-cache-dir --editable .[test]
```

1.2 Tutorial

1.2.1 Basic Usage

Here is an interactive session showing the basic commands usage.

```
>>> import pandas as pd
>>> from pgcom import Commuter
```

(continues on next page)

(continued from previous page)

```
# create commuter
>>> commuter = Commuter(dbname="test", user="postgres", password="secret", host=
↪"localhost")

# execute a command: this creates a new table
>>> commuter.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data␣
↪varchar);")

# write from DataFrame to a table
>>> df = pd.DataFrame([[100, "abc"], [200, "abc'def"]], columns=["num", "data"])
>>> commuter.insert(table_name="test", data=df)

# read from table to a DataFrame
>>> commuter.select("SELECT * FROM test")
   id  num     data
0   1  100      abc
1   2  200  abc'def

# pass data to fill a query placeholders
>>> commuter.select("SELECT * FROM test WHERE data = (%s)", ("abc'def",))
   id  num     data
0   2  200  abc'def
```

1.2.2 Writing to a table with copy from

PostgreSQL COPY FROM command copies data from a file-system file to a table (appending the data to whatever is in the table already).

Commuter's *copy_from* method provides an adaptation between Pandas DataFrame and COPY FROM, the DataFrame format however must be compatible with database table (data types, columns order, etc).

```
>>> commuter.copy_from(table_name="test", data=df)
```

A slight adaptation of DataFrame to the table structure can be attained by setting `format_data` parameter to `True`. This enables to use DataFrames with the incomplete set of columns given in any order.

```
>>> df = pd.DataFrame([["abc", 100], ["abc'def", 200]], columns=["data", "num_2"])

# DataFrame has column "num_2" not presented in table
>>> commuter.copy_from("test", df)
pgcom.exc.CopyError: column "num_2" of relation "test" does not exist

>>> commuter.copy_from("test", df, format_data=True)
>>> commuter.select("SELECT * FROM test")
   id   num     data
0   1  None      abc
1   2  None  abc'def
```

1.2.3 Upsert with copy from

If DataFrame contains rows conflicting with table constraints and you need to implement UPSERT, you can specify `where` parameter of *copy_from* method. Then it removes rows from the table before applying COPY FROM.

On the other hand, if you want to sanitize DataFrame and remove conflicts from it rather than from the table, you can use *resolve_primary_conflicts* and *resolve_foreign_conflicts*.

```
>>> commuter.execute("CREATE TABLE test (id integer PRIMARY KEY, num integer, data␣
↪varchar);")
>>> df_1 = pd.DataFrame([[1, 100, "a"], [2, 200, "b"]], columns=["id", "num", "data"])
>>> commuter.copy_from("test", df_1)

# df_2 has primary key conflict
>>> df_2 = pd.DataFrame([[2, 201, "bb"], [3, 300, "c"]], columns=["id", "num", "data"])
>>> commuter.copy_from("test", df_2)
pgcom.exc.CopyError: duplicate key value violates unique constraint "test_pkey"

# remove all rows from test table where id >= 2
>>> commuter.copy_from("test", df_2, where="id >= 2")
>>> commuter.select("SELECT * FROM test")
   id  num data
0   1  100    a
1   2  201   bb
2   3  300    c

>>> df_3 = pd.DataFrame([[3, 301, "cc"], [4, 400, "d"]], columns=["id", "num", "data"])

# remove conflicts from the DataFrame
>>> commuter.resolve_primary_conflicts("test", df_3)
   id  num data
0   4  400    d
```

Note: Be careful when resolving conflicts on DataFrame. Since both methods query data from the table, the whole table will be queried if you don't specify `where` parameter.

1.2.4 Encode categorical columns

If DataFrame contains a column with string categories which you want to place in a separate table with a serial primary key. And you want to replace categories with the corresponding key value, to minimize the original table size, you can use `encode_category` method.

It implements writing of all the unique values in categorical column given by `category_name` to the table given by parameter `category_table`.

In the example below, we have a DataFrame with a categorical column `city`. We store it in a separate table called `cities`. And replace column with the corresponding `city_id`.

```
>>> df
     city  year
0  Berlin  2010
1  Berlin  2011
```

(continues on next page)

(continued from previous page)

```
2  London  2015
3   Paris  2012
4  Berlin  2018

>>> commuter.execute("CREATE TABLE cities (city_id SERIAL PRIMARY KEY, city TEXT)")
>>> df = commuter.encode_category(
...     data=df, category="city", key="city_id", category_table="cities")
>>> df
     city  year  city_id
0  Berlin  2010        1
1  Berlin  2011        1
2  London  2015        2
3   Paris  2012        3
4  Berlin  2018        1

>>> commuter.select("SELECT * FROM cities")
   city_id    city
0        1  Berlin
1        2  London
2        3   Paris
```

When categories are presented by multiple columns, its suggested to use `encode_composite_category` method. It implements writing of all the unique combinations given by multiple columns in DataFrame to the table given by `category_table`.

```
>>> df
     city  year country
0  Berlin  2010 Germany
1  Berlin  2011 None
2  London  2015 UK
3   Paris  2012 France
4  Berlin  2018 UK

>>> cmd = "CREATE TABLE cities (city_id SERIAL PRIMARY KEY, city TEXT, country TEXT)"
>>> commuter.execute(cmd)

>>> cats = {"city: city", "country": "country"}
>>> df = commuter.encode_composite_category(
...     data=df, categories=cats, key="city_id",
...     category_table="cities", na_value="NONE")
>>> df
     city  year  country  city_id
0  Berlin  2010  Germany        1
1  Berlin  2018  Germany        1
2  Berlin  2011     NONE        2
3  London  2015       UK        3
4   Paris  2012   France        4

>>> commuter.select("SELECT * FROM cities")
   city_id    city  country
0        1  Berlin  Germany
1        2  Berlin     NONE
```

(continues on next page)

(continued from previous page)

```
2        3  London       UK
3        4   Paris   France
```

1.2.5 Connection options

A connection pooling technique is used to maintain a "pool" of active database connections in memory which are reused across the requests.

Testing the connection for liveness is available by using `pre_ping` argument. This feature will normally emit "SELECT 1" statement on each request to the database. If an error is raised, the pool will be immediately restarted.

```
>>> commuter = Commuter(pre_ping=True, **conn_args)
```

The exponential backoff strategy is used for reconnection. By default, it implements 3 reconnection attempts. This can be changed by setting `max_reconnects` argument.

```
>>> commuter = Commuter(pre_ping=True, max_reconnects=5, **conn_args)
```

Note: When creating a new instance of *Commuter*, the connection pool is created by calling `make_pool` and the connection is established. The typical usage of *Commuter* is therefore once per particular database, held globally for the lifetime of a single application process.

Warning: So far a simple connection pool is used and it can't be shared across different threads.

1.2.6 Extras

Here is the use cases of other *Commuter* methods.

Select data from the table and return a scalar value.

```
>>> commuter.select_one("SELECT MAX(num) FROM test")
300
```

Insert one row to the table passing values using key-value arguments.

```
>>> commuter.insert_row("test", id=5, num=500, data="abc'def")
```

When using a serial column to provide unique identifiers, it can be very handy to insert one row and return the serial ID assigned to it.

```
>>> row_id = commuter.insert_row("test", return_id="id", num=500, data="abc'def")
```

Insert rows using custom placeholders, e.g. to insert PostGIS data.

```
>>> commuter.insert("test", data,
...     columns=["name", "geom"],
...     placeholders=["%s", "ST_GeomFromText(%s, 4326)"])
```

Check if the table exists.

```
>>> commuter.is_table_exist("test")
True
```

Check if the specific entry exists in the table. It implements a simple query building a WHERE clause from kwargs.

```
# SELECT 1 FROM TABLE test WHERE id=5 AND num=500
>>> commuter.is_entry_exist("test", id=5, num=500)
True
```

Delete entry from the table, specifying a WHERE clause using kwargs.

```
# DELETE FROM TABLE test WHERE id=5 AND num=500
>>> commuter.delete_entry("test", id=5, num=500)
True
```

Return the number of active connections to the database.

```
>>> commuter.get_connections_count()
9
```

1.2.7 Listener

PostgreSQL provides tools for setting asynchronous interaction with database session using LISTEN and NOTIFY commands.

A client application registers as a listener on the notification channel with the LISTEN command (and can stop listening with the UNLISTEN command). When the command NOTIFY is executed, the application listening on the channel is notified. A payload can be passed to provide some extra data to the listener. This is commonly used when sending notifications that table rows have been modified.

Notifications are received after trigger is fired, the poll method can be used to check for the new notifications without wasting resources.

Methods *create_notify_function* and *create_trigger* present basic query constructors, which can be used to define a new trigger and a new function associated with this trigger. Some custom definitions can be done using execute method.

Here is the example of simple application receiving notification when rows are inserted to the table. See API reference for more details

```
from pgcom import Listener

>>> listener = Listener(dbname="test", user="postgres", password="secret", host=
↪"localhost")

# create a function called by trigger, it generates a notification
# which is sending to test_channel
>>> listener.create_notify_function(func_name='notify_trigger', channel='test_channel')

# create a trigger executed AFTER INSERT STATEMENT
>>> listener.create_trigger(table_name='test', func_name='notify_trigger')

# register function callback activated on the notification
>>> def on_notify(payload):
```

(continues on next page)

(continued from previous page)

```
...     print("received notification")

# listening loop, callback is activated on every INSERT to "test" table
>>> listener.poll(channel='test_channel', on_notify=on_notify)
received notification
received notification
```

Note: Note that the payload is only available from PostgreSQL 9.0: notifications received from a previous version server will have the payload attribute set to the empty string.

df = commuter.encode_composite_category(data=df, categories={“city: city”, “country”: “country”}, key=”city_id”, category_table=”cities”)

1.3 API Reference

1.3.1 Connection handlers

BaseConnector(**kwargs)	Base class for all connectors.
Connector([pool_size, pre_ping, max_reconnects])	Setting a connection with database.

pgcom.base.BaseConnector

class BaseConnector(***kwargs: str*)

Base class for all connectors.

Methods

close_all()	Close all active connections.
open_connection()	Generates a new connection.

pgcom.base.BaseConnector.close_all

abstract BaseConnector.**close_all**() → None

Close all active connections.

pgcom.base.BaseConnector.open_connection

abstract BaseConnector.**open_connection**() → Iterator[psycopg2.connect]

Generates a new connection.

pgcom.connector.Connector

class Connector(*pool_size: int = 20*, *pre_ping: bool = False*, *max_reconnects: int = 3*, ***kwargs: str*)

Setting a connection with database.

Besides the basic connection parameters any other connection parameter supported by psycopg2.connect can be passed as a keyword.

Parameters

- **pool_size** – The maximum amount of connections the pool will support.
- **pre_ping** – If True, the pool will emit a "ping" on the connection to test if the connection is alive. If not, the connection will be reconnected.
- **max_reconnects** – The maximum amount of reconnects, defaults to 3.

Methods

close_all()	Close all the connections handled by the pool.
make_pool()	Create a connection pool.
open_connection()	Generate a free connection from the pool.
ping(conn)	Ping the connection for liveness.
restart_pool()	Close all the connections and create a new pool.

pgcom.connector.Connector.close_all

Connector.**close_all**() → None

Close all the connections handled by the pool.

pgcom.connector.Connector.make_pool

Connector.**make_pool**() → psycopg2.pool.SimpleConnectionPool

Create a connection pool.

A connection pool that can't be shared across different threads.

pgcom.connector.Connector.open_connection

Connector.**open_connection**() → Iterator[psycopg2.connect]

Generate a free connection from the pool.

If pre_ping is True, then the connection is tested whether its alive or not. If not, then reconnect.

pgcom.connector.Connector.ping

static Connector.**ping**(*conn: psycopg2.connect*) → bool

Ping the connection for liveness.

Implements a ping ("SELECT 1") on the connection. Return True if the connection is alive, otherwise False.

Parameters **conn** – The connection object to ping.

pgcom.connector.Connector.restart_pool

Connector.**restart_pool**() → psycopg2.pool.SimpleConnectionPool

Close all the connections and create a new pool.

1.3.2 Communication agents

BaseCommuter(connector)	Base class for all commuters.
Commuter([pool_size, pre_ping, max_reconnects])	Communication agent.

pgcom.base.BaseCommuter

class BaseCommuter(*connector:* pgcom.base.BaseConnector)

Base class for all commuters.

Parameters **connector** – Instance of connection handler, any subclass inherited from *BaseConnector*.

Methods

execute(cmd[, values])	Execute a database operation (query or command).

pgcom.base.BaseCommuter.execute

BaseCommuter.**execute**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*) → None

Execute a database operation (query or command).

Parameters

- **cmd** – SQL query to be executed.
- **values** – Query parameters.

Returns List of rows of a query result and list of column names. Two empty lists are returned if there is no records to fetch.

Raises **QueryExecutionError** – if execution fails.

pgcom.commuter.Commuter

class Commuter(*pool_size: int = 20*, *pre_ping: bool = False*, *max_reconnects: int = 3*, ***kwargs: str*)

Communication agent.

When creating a new instance of Commuter, the connection pool is created and the connection is established. The typical usage of Commuter is therefore once per particular database, held globally for the lifetime of a single application process.

Parameters

- **pool_size** – The maximum amount of connections the pool will support.
- **pre_ping** – If True, the pool will emit a "ping" on the connection to test if the connection is alive. If not, the connection will be reconnected.
- **max_reconnects** – The maximum amount of reconnects, defaults to 3.

Methods

copy_from(table_name, data[, format_data, ...])	Places DataFrame to a buffer and apply COPY FROM command.
delete_entry(table_name, **kwargs)	Delete entry from the table.
encode_category(data, category, key, ...[, ...])	Encode categorical column.
encode_composite_category(data, categories, ...)	Encode categories represented by multiple columns.
execute(cmd[, values])	Execute a database operation (query or command).
execute_script(path2script)	Execute query from file.
get_connections_count()	Returns the amount of active connections.
insert(table_name, data[, columns, placeholders])	Write rows from a DataFrame to a database table.
insert_return(cmd[, values, return_id])	Insert a new row to the table and return the serial key of the newly inserted row.
insert_row(table_name[, return_id])	Implements insert command.
is_entry_exist(table_name, **kwargs)	Return True if entry already exists, otherwise return False.
is_table_exist(table_name)	Return True if table exists, otherwise False.
make_where(keys)	Build WHERE clause from list of keys.
resolve_foreign_conflicts(table_name, ...[, ...])	Resolve foreign key conflicts in DataFrame.
resolve_primary_conflicts(table_name, data)	Resolve primary key conflicts in DataFrame.
select(cmd[, values])	Read SQL query into a DataFrame.
select_one(cmd[, values, default])	Select the first element of returned DataFrame.

pgcom.commuter.Commuter.copy_from

Commuter.**copy_from**(*table_name: str*, *data: pandas.core.frame.DataFrame*, *format_data: bool = False*, *sep: str = ','*, *na_value: str = ''*, *where: Optional[Union[str, psycopg2.sql.Composed]] = None*) → None

Places DataFrame to a buffer and apply COPY FROM command.

Parameters

- **table_name** – Name of the table where to insert.
- **data** – DataFrame from where to insert.
- **format_data** – Reorder columns and adjust dtypes wrt to table metadata from information_schema.
- **sep** – String of length 1. Field delimiter for the output file. Defaults to ",".
- **na_value** – Missing data representation, defaults to "".
- **where** – WHERE clause used to specify a condition while deleting data from the table before applying copy_from, DELETE command is not executed if not specified.

Raises **CopyError** – if execution fails.

pgcom.commuter.Commuter.delete_entry

Commuter.**delete_entry**(*table_name: str*, ***kwargs: Any*) → None

Delete entry from the table.

Implements a simple query to delete a specific entry from the table. WHERE clause is created from `**kwargs`.

Parameters

- **table_name** – Name of the database table.
- ****kwargs** – Parameters to create WHERE clause.

Examples

Delete rows with version=100 from the table.

```
>>> self.delete_entry("dict_versions", version=100)
```

pgcom.commuter.Commuter.encode_category

Commuter.**encode_category**(*data: pandas.core.frame.DataFrame*, *category: str*, *key: str*, *category_table: str*, *category_name: Optional[str] = None*, *key_name: Optional[str] = None*, *na_value: Optional[str] = None*) → pandas.core.frame.DataFrame

Encode categorical column.

Implements writing of all the unique values in categorical column given by `category_name` to the table given by `category_table`.

Replaces all the values in `category` column in the original DataFrame with the corresponding integer values assigned to categories via serial primary key constraint.

Parameters

- **data** – Pandas.DataFrame with categorical column.
- **category** – Name of the categorical column in DataFrame the method is applied for.
- **key** – Name of the DataFrame column with encoded values.
- **category_table** – Name of the table with stored categories.
- **category_name** – Name of the categorical column in `category_table`. Defaults to `category`.
- **key_name** – Name of the column in `category_table` contained the encoded values. Defaults to `key`.
- **na_value** – Missing data representation.

Returns Pandas.DataFrame with encoded category.

pgcom.commuter.Commuter.encode_composite_category

Commuter.**encode_composite_category**(*data: pandas.core.frame.DataFrame*, *categories: Dict[str, str]*, *key: str*, *category_table: str*, *key_name: Optional[str] = None*, *na_value: Optional[str] = None*) → pandas.core.frame.DataFrame

Encode categories represented by multiple columns.

Implements writing of all the unique combinations given by multiple columns in DataFrame to the table given by `category_table`.

Dictionary `categories` provides a mapping between DataFrame and `category_table` column names.

Parameters

- **data** – Pandas.DataFrame with categorical columns.
- **categories** – Dictionary provided the mapping between column names. Dict keys provide names of columns in `data` represented category, values represent column names in `category_table`.
- **key** – Name of the DataFrame column with encoded values.
- **category_table** – Name of the table with stored categories.
- **key_name** – Name of the column in `category_table` contained the encoded values. Defaults to `key`.
- **na_value** – Missing data representation.

Returns Pandas.DataFrame with encoded category.

pgcom.commuter.Commuter.execute

Commuter.**execute**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*) → None

Execute a database operation (query or command).

Parameters

- **cmd** – SQL query to be executed.
- **values** – Query parameters.

Returns List of rows of a query result and list of column names. Two empty lists are returned if there is no records to fetch.

Raises **QueryExecutionError** – if execution fails.

pgcom.commuter.Commuter.execute_script

Commuter.**execute_script**(*path2script: str*) → None

Execute query from file.

Parameters **path2script** – Path to the file with the query.

pgcom.commuter.Commuter.get_connections_count

Commuter.**get_connections_count**() → int

Returns the amount of active connections.

pgcom.commuter.Commuter.insert

Commuter.**insert**(*table_name: str*, *data: pandas.core.frame.DataFrame*, *columns: Optional[List[str]] = None*, *placeholders: Optional[List[str]] = None*) → None

Write rows from a DataFrame to a database table.

Parameters

- **table_name** – Name of the destination table.
- **data** – Pandas.DataFrame with the data to be inserted.
- **columns** – List of column names used for insert. If not specified then all the columns are used. Defaults to None.
- **placeholders** – List of placeholders. If not specified then the default placeholders are used. Defaults to None.

Examples

```
>>> self.insert("people", data)
```

Insert two columns, name and age.

```
>>> self.insert("people", data, columns=["name", "age"])
```

You can customize placeholders to implement advanced insert, e.g. to insert geometry data in a database with PostGIS extension.

```
>>> self.insert(
...     table_name="polygons",
...     data=data,
...     columns=["name", "geom"],
...     placeholders=["%s", "ST_GeomFromText(%s, 4326)"])
```

pgcom.commuter.Commuter.insert_return

Commuter.**insert_return**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*, *return_id: Optional[str] = None*) → int

Insert a new row to the table and return the serial key of the newly inserted row.

Parameters

- **cmd** – INSERT INTO command.
- **values** – Collection of values to be inserted.
- **return_id** – Name of the returned serial key.

pgcom.commuter.Commuter.insert_row

Commuter.**insert_row**(*table_name: str*, *return_id: Optional[str] = None*, ***kwargs: Any*) → Optional[int]

Implements insert command.

Inserted values are passed through the keyword arguments.

Parameters

- **table_name** – Name of the destination table.
- **return_id** – Name of the returned serial key.

pgcom.commuter.Commuter.is_entry_exist

Commuter.**is_entry_exist**(*table_name: str*, ***kwargs: Any*) → bool

Return True if entry already exists, otherwise return False.

Implements a simple query to verify if a specific entry exists in the table. WHERE clause is created from `**kwargs`.

Parameters

- **table_name** – Name of the database table.
- ****kwargs** – Parameters to create WHERE clause.

Examples

Implement query `SELECT 1 FROM people WHERE id=5 AND num=100`.

```
>>> self.is_entry_exist("my_table", id=5, num=100)
True
```

pgcom.commuter.Commuter.is_table_exist

Commuter.**is_table_exist**(*table_name: str*) → bool

Return True if table exists, otherwise False.

Parameters **table_name** – Name of the table where to insert.

pgcom.commuter.Commuter.make_where

static Commuter.**make_where**(*keys: List[str]*) → psycopg2.sql.Composed

Build WHERE clause from list of keys.

Examples

```
>>> self.make_where(["version", "task"])
"version=%s AND task=%s"
```

pgcom.commuter.Commuter.resolve_foreign_conflicts

Commuter.**resolve_foreign_conflicts**(*table_name: str*, *parent_name: str*, *data: pandas.core.frame.DataFrame*, *where: Optional[Union[str, psycopg2.sql.Composed]] = None*) → pandas.core.frame.DataFrame

Resolve foreign key conflicts in DataFrame.

Remove all the rows from the DataFrame conflicted with foreign key constraint.

Parameter `where` is used to reduce the amount of querying data.

Parameters

- **table_name** – Name of the child table, where the data needs to be inserted.
- **parent_name** – Name of the parent table.
- **data** – DataFrame with foreign key conflicts.
- **where** – WHERE clause used when querying from the `table_name`.

Returns DataFrame without foreign key conflicts.

pgcom.commuter.Commuter.resolve_primary_conflicts

Commuter.**resolve_primary_conflicts**(*table_name: str*, *data: pandas.core.frame.DataFrame*, *where: Optional[Union[str, psycopg2.sql.Composed]] = None*) → pandas.core.frame.DataFrame

Resolve primary key conflicts in DataFrame.

Remove all the rows from the DataFrame conflicted with primary key constraint.

Parameter `where` is used to reduce the amount of querying data.

Parameters

- **table_name** – Name of the table.
- **data** – DataFrame where the primary key conflicts need to be resolved.
- **where** – WHERE clause used when querying data from the `table_name`.

Returns DataFrame without primary key conflicts.

pgcom.commuter.Commuter.select

Commuter.**select**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*) → pandas.core.frame.DataFrame

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result of the query.

Parameters

- **cmd** – string SQL query to be executed.
- **values** – Parameters to pass to execute method.

Returns Pandas.DataFrame.

pgcom.commuter.Commuter.select_one

Commuter.**select_one**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*, *default: Optional[Any] = None*) → Any

Select the first element of returned DataFrame.

Parameters

- **cmd** – string SQL query to be executed.
- **values** – Parameters to pass to execute method.
- **default** – If query result is empty, then return the default value.

Properties

connector	

pgcom.commuter.Commuter.connector

Commuter.**connector:** *pgcom.connector.Connector*

1.3.3 Asynchronous interaction

Listener([pool_size, pre_ping, max_reconnects])	Listener on the notification channel.

pgcom.listener.Listener

class Listener(*pool_size: int = 20*, *pre_ping: bool = False*, *max_reconnects: int = 3*, ***kwargs: str*)

Listener on the notification channel.

This class implements an asynchronous interaction with database offered by PostgreSQL commands LISTEN and NOTIFY.

Notifications are received after trigger is fired, the *poll* method can be used to check for the new notifications without wasting resources.

Methods *create_notify_function* and *create_trigger* present basic query constructors, which can be used to define a new trigger and a new function associated with this trigger. Some custom definitions can be done using *execute* method.

Parameters

- **pool_size** – The maximum amount of connections the pool will support.
- **pre_ping** – If True, the pool will emit a "ping" on the connection to test if the connection is alive. If not, the connection will be reconnected.
- **max_reconnects** – The maximum amount of reconnects, defaults to 3.

Methods

create_notify_function(func_name, channel)	Create a function called by trigger.
create_trigger(table_name, func_name[, ...])	Create trigger.
execute(cmd[, values])	Execute a database operation (query or command).
poll(channel[, on_notify, on_timeout, ...])	Listen to the channel and activate callbacks on the notification.

pgcom.listener.Listener.create_notify_function

Listener.create_notify_function(*func_name: str*, *channel: str*) → None

Create a function called by trigger.

This function generates a notification, which is sending to the specified channel when trigger is fired.

Parameters

- **func_name** – Name of the function.
- **channel** – Name of the the channel the notification is sending to.

pgcom.listener.Listener.create_trigger

Listener.create_trigger(*table_name: str*, *func_name: str*, *trigger_name: Optional[str] = None*, *when: str = 'AFTER'*, *event: str = 'INSERT'*, *for_each: str = 'STATEMENT'*) → None

Create trigger.

Creates a new trigger associated with the table and executed the specified function when certain events occur.

Parameters

- **table_name** – The name of the table the trigger is for.
- **func_name** – A user-supplied function, which is executed when the trigger fires.
- **trigger_name** – The name to give to the new trigger. If not specified, then the automatically created name will be assigned.
- **when** – One of "BEFORE", "AFTER", "INSTEAD OF". Determines when function is called.
- **event** – One of "INSERT", "UPDATE", "DELETE", "TRUNCATE". Use "OR" for event combinations, e.g. "INSERT OR UPDATE".
- **for_each** – One of "ROW", "STATEMENT". This specifies whether the trigger should be fired once for every row affected by the event, or just once per SQL statement.

pgcom.listener.Listener.execute

Listener.**execute**(*cmd: Union[str, psycopg2.sql.Composed]*, *values: Optional[Union[Sequence[Any], Mapping[str, Any]]] = None*) → None

Execute a database operation (query or command).

Parameters

- **cmd** – SQL query to be executed.
- **values** – Query parameters.

Returns List of rows of a query result and list of column names. Two empty lists are returned if there is no records to fetch.

Raises **QueryExecutionError** – if execution fails.

pgcom.listener.Listener.poll

Listener.**poll**(*channel: str*, *on_notify: Optional[Callable[[str], None]] = None*, *on_timeout: Optional[Callable] = None*, *on_close: Optional[Callable] = None*, *on_error: Optional[Callable[[Exception], None]] = None*, *timeout: int = 5*) → None

Listen to the channel and activate callbacks on the notification.

This function sleeps until awakened when there is some data to read on the connection.

Parameters

- **channel** – Name of the notification channel.
- **on_notify** – Callback to be executed when the notification has arrived.
- **on_timeout** – Callback to be executed by timeout.
- **on_close** – Callback to be executed when connection is closed.
- **on_error** – Callback to be executed if error occurs.
- **timeout** – Timeout in seconds.

1.4 ChangeLog

1.4.1 0.2.9 (2022-04-04)

- Fixed *encode_composite_category* for integer categories.

1.4.2 0.2.8 (2021-08-16)

- Added *encode_composite_category* method.

1.4.3 0.2.7 (2021-06-28)

- Fixed *copy_from* to make it compatible with *psycopg2 2.9* release.

1.4.4 0.2.5 (2021-03-11)

- Fixed #17
- Fixed #18
- Fixed #19

1.4.5 0.2.4 (2021-02-28)

- Fixed issue with custom placeholders in *insert* method

1.4.6 0.2.3 (2021-02-27)

- Deprecated `schema` parameter
- Added *is_entry_exist* method
- Added *delete_entry* method
- Added *encode_category* method
- Added *make_where* method
- Updated *insert* method to support customized placeholders

1.4.7 0.2.2 (2020-07-27)

- Added adaptation for numpy `int64` and `float64`.

1.4.8 0.2.1 (2020-07-13)

- Added *BaseConnector*
- Added *BaseCommuter*

1.4.9 0.2.0 (2020-07-04)

- Added support of `psycopg2.sql.Composed` arguments to *Commuter* methods
- Fixed query parameters issues in *Commuter* methods (#11)
- Made connect args compatible with `psycopg2.connect`
- Added `pre_ping` and `max_connects` options to `Connector` (#12, #14)
- Added connection pooling to `Connector`
- Deprecated SQLAlchemy dependencies

1.4.10 0.1.7 (2020-05-31)

- Fixed #9

1.4.11 0.1.6 (2020-05-28)

- Updated `_format_data` to fix text fields with comma

1.4.12 0.1.5 (2020-03-16)

- Fixed data formatting on integer columns with missed values (#5)

1.4.13 0.1.4 (2020-01-21)

- Changed `where` argument type in *resolve_primary_conflicts* from positional to optional
- Changed `where` argument type in *resolve_foreign_conflicts* from positional to optional
- Fixed bug in copying from DataFrame with incomplete set of columns (#3)
- Added new test

1.4.14 0.1.3 (2020-01-19)

- Added support for the missing SQLAlchemy dependency (#1)
- Added `_execute` (#2)
- Added pending transaction handler to *copy_from*
- Raised `ExecutionError` when execute command fails
- Replaced `pandas.to_sql` in *insert* by `psycopg.execute_batch`
- Changed sqlalchemy engine url builder
- Added new tests

1.4.15 0.1.2 (2020-01-16)

- Changed *select* method
- Changed *insert* method
- Fixed exception in *copy_from*

1.4.16 0.1.1 (2020-01-10)

- Added *Listener* class
- Added `fix_schema` decorator
- Added *select_one* method
- Added `where` argument to *resolve_foreign_conflicts* method
- Added `where` argument to *copy_from* method
- Added `_table_columns` method
- Added `_primary_key` method
- Added `_foreign_key` method
- Moved sql queries to queries.py
- Deprecated `f_key`, `filter_col` arguments of *resolve_foreign_conflicts* method
- Deprecated `p_key`, `filter_col` argument of *resolve_primary_conflicts* method
- Deprecated `return_scalar` argument of *select* method
- Deprecated `get_columns` method

1.4.17 0.1.0 (2020-01-02)

Pre-release

MIT License (see LICENSE).

INDEX