PyVertica Documentation

Release 1.5.2

Spil Games

December 05, 2013

Contents

pyvertica is a package which contains the shared logic for connecting and writing to a Vertica database.

CHAPTER 1

Installation

pyvertica can be installing by executing `pip install pyvertica`.

Note: When using the `BaseImporter`, do not forget to create the batch history table. An SQL example can be found in the class documentation.

CHAPTER 2

Links

- documentation
- source

CHAPTER 3

Command-line usage

3.1 vertica_batch_import

Tool to import a CSV-like file directly into Vertica.

```
usage: vertica_batch_import [-h] [--commit]
                            [--partial-commit-after PARTIAL_COMMIT_AFTER]
                            [--log {debug,info,warning,error,critical}]
                            [--truncate-table] [--delimiter DELIMITER]
                            [--enclosed-by ENCLOSED_BY] [--skip SKIP]
                            [--null NULL]
                            [--record-terminator RECORD_TEMINATOR]
                            dsn table_name file_path

Vertica batch importer

positional arguments:
  dsn                   ODBC data source name
  table_name            name of table (including schema, eg: staging.my_table)
  file_path             absolute path to the file to import

optional arguments:
  -h, --help            show this help message and exit
  --commit              commit after import (without it will perform a dry-
                        run)
  --partial-commit-after PARTIAL_COMMIT_AFTER
                        partial commit after num of lines (default: 1000000)
  --log {debug,info,warning,error,critical}
                        loglevel of loghandler (default: info)
  --truncate-table      truncate table before import
  --delimiter DELIMITER
                        delimiter to split columns (default: ;)
  --enclosed-by ENCLOSED_BY
                        the quote character (default: ")
  --skip SKIP           number of lines to skip (default: 0)
  --null NULL           represents a null value (default: empty string)
  --record-terminator RECORD_TEMINATOR
                        specifies the end of a record (default: newline)
```

3.2 vertica_migrate

Tool to migrate data from one to another Vertica cluster.

```
usage: vertica_migrate [-h] [--commit]
                       [--log-level {debug,info,warning,error,critical}]
                       [--skip-ddls] [--clever-ddls] [--skip-data]
                       [--even-not-empty] [--limit LIMIT] [--truncate]
                       [--source-not-reconnect] [--target-not-reconnect]
                       [--config-path CONFIG_PATH]
                      source target [objects [objects ...]]

Vertica Migrator

positional arguments:
  source                ODBC data source name
  target                ODBC data source name
  objects               List of objects (schemas or table) to migrate

optional arguments:
  -h, --help            show this help message and exit
  --commit              commit DDLS and copy data (without it will perform a
                        dry-run)
  --log-level {debug,info,warning,error,critical}
                        loglevel of loghandler (default: info)
  --skip-ddls           Do not copy the DDLs over.
  --clever-ddls         If when copying a DDL an object with the same name
                        already exists, skip the copy.
  --skip-data           Do not copy the data over.
  --even-not-empty      Do not stop if the target DB is not empty.
  --limit LIMIT         Limit the number of rows to copy over, per table.
  --truncate            Truncate destination tables before copying data over.
  --source-not-reconnect
                        Do not try to avoid load balancer by reconnecting.
  --target-not-reconnect
                        Do not try to avoid load balancer by reconnecting.
  --config-path CONFIG_PATH
                        Absolute path to a config file (useful for storing
                        credentials).
```

To not expose passwords on the command-line, it is mandatory to pass them as a config file (`--config-path`). Example:

```
[vertica_migrate]
target_pwd=targetpassword
source_pwd=sourcepassword
log_level=warning
```

Any command-line argument accepting a string (eg: `--log-level warning`) is accepted (eg: `log_level=warning`). The following extra options are available in the config-file:

target_user Username of the target Vertica database.

target_pwd Password of the target Vertica database.

target_host Hostname of the target Vertica database.

source_user Username of the source Vertica database.

source_pwd Password of the source Vertica database.

source_host Hostname of the target Vertica database.

CHAPTER 4

Usage within Python code

4.1 Creating a PYODBC connection to Vertica

pyvertica.connection.**connection_details**(*con*)
Given one connection objects returns information about it.

Parameters **con** – An instance of `pyodbc.Connection`.

return: A `dict` with the following keys / values:

host Connected node IP address

user Connected username

db Connected database name

pyvertica.connection.**get_connection**(*reconnect=True*, ***kwargs*)
Get `pyodbc` connection for the given `dsn`.

Usage example:

```
from pyvertica.connection import get_connection


connection = get_connection('TestDSN')
cursor = connection.cursor()
```

The connection will be made in two steps (with the assumption that you are connection via a load-balancer). The first step is connecting to the load-balancer and selecting a random node address. Then it will connect to that specific node and return this connection instance. This is done to avoid that all the data has to pass the load-balancer.

Note: Depending on the given keyword arguments, you need to have a `odbc.ini` file on your system.

Parameters

- **reconnect** – A `boolean` asking to reconnect to skip load balancer.
- **kwargs** – Keyword arguments accepted by the `pyodbc` module. See: http://code.google.com/p/pyodbc/wiki/Module#connect

Returns Return an instance of `pyodbc.Connection`.

4.2 Writing multiple lines in a batch

class `pyvertica.batch.`**`VerticaBatch`**(*table_name*, *odbc_kwargs={}*, *truncate_table=False*, *reconnect=True*, *analyze_constraints=True*, *column_list=[]*, *copy_options={}*, *connection=None*)

Object for writing multiple records to Vertica in a batch.

Usage example:

```
from pyvertica.batch import VerticaBatch

batch = VerticaBatch(
    odbc_kwargs={'dsn': 'VerticaDWH'},
    table_name='schema.my_table',
    truncate=True,
    column_list=['column_1', 'column_2'],
    copy_options={
        'DELIMITER': ',',
    }
)

row_list = [
    ['row_1_val_1', 'row_1_val_2'],
    ['row_2_val_1', 'row_2_val_2'],
    ...
]

for column_data_list in row_list:
    batch.insert_list(column_data_list)

error_bool, error_file_obj = batch.get_errors()

if error_bool:
    print error_file_obj.read()

batch.commit()
```

Note: It is also possible to call `commit()` multiple times (for example after every 50000 records). Please note that after the first insert and after calling `commit()`, the output of `get_errors()` will reflect the new serie of inserts and thus not contain the "old" inserts.

Note: Creating a new batch object will not create a lock on the target table. This will happen only after first insert.

Note: Although the batch object is automagically reusable, after a `commit()` the locks are realeased up to next insert.

Parameters

- **table_name** – A `str` representing the table name (including the schema) to write to. Example: `'staging.my_table'`.
- **odbc_kwargs** – A `dict` containing the ODBC connection keyword arguments. E.g.:

```
{
    'dsn': 'TestDSN',
}
```

See Also:

https://code.google.com/p/pyodbc/wiki/Module

- **truncate_table** – A `bool` indicating if the table needs truncating before first insert. Default: `False`. *Optional*.
- **reconnect** – A `bool` passed to the connection object to decide if pyvertica should directly reconnect to a random node to bypass a load balancer.
- **analyze_constraints** – A `bool` indicating if a `ANALYZE_CONSTRAINTS` startement should be executed when getting errors. Default: `True`. *Optional*.
- **column_list** – A `list` containing the columns that will be written. *Optional*.
- **copy_options** – A `dict` containing the keys to override. For a list of existing keys and their defaults, see `copy_options_dict`. *Optional*.
- **connection** – A `pyodbc.Connection` to use instead of opening a new connection. If this parameter is supplied, `odbc_kwargs` may not be supplied. Default: `None`. *Optional*.

commit()
Commit the current transaction.

copy_options_dict = {'REJECTEDFILE': True, 'REJECTMAX': 0, 'DELIMITER': ';', 'NO COMMIT': True, 'EN
Default copy options for SQL query.

Note: By default `REJECTEDFILE` is set to `__debug__`, which is `True`, unless you've set the `PYTHONOPTIMIZE` environment variable.

---

get_batch_count()
Return number (`int`) of inserted items since last commit.

Warning: When using `insert_raw()` this value represents the number of raw `str` objects inserted, not the number of lines!

Returns An `int`.

get_cursor()
Return a cursor to the database.

This is useful when you want to add extra data within the same transaction of the batch import.

Returns Instance of `pyodbc.Cursor`.

get_errors()
Get errors that were raised since the last commit.

This will check constraint errors and rejected data by the database. Please note that this will remove the rejected data file after calling this method. Therfore it is not possible to call this method more than once per batch!

Note: Since this is checking the contraints as well, it is assumed that all contrains were met before starting the batch. Otherwise, these errors will show up within this method.

Returns

A `tuple` with as first item a `int` representing the number of errors. The second item is a file-like object containing the error-data in plain text. Since this is an instance of `tempfile.TemporaryFile`, it will be removed automatically.

Note: The file-like object can be empty, when `REJECTEDFILE` is set to `False`.

---

get_total_count()
Return total number (`int`) of inserted items.

Warning: When using `insert_raw()` this value represents the number of raw `str` objects inserted, not the number of lines!

Returns An `int`.

insert_line(**args*, ***kwargs*)
Insert a `str` containing all the values.

This is useful when inserting lines directly from a CSV file for example.

Note: When you have a loghandler with `DEBUG` level, every query will be logged. For performance reason, this log statement is only executed when `__debug__` equals `True` (which is the default case). For a better performance, you should invoke the Python interpreter with the `-O` argument or set the environment variable `PYTHONOPTIMIZE` to something.

Example:

```
batch.insert_line('"value_1";"value_2"')
```

Parameters **line_str** – A `str` representing the line to insert. Make sure the `str` is formatted according `copy_options_dict`. Example: `'"value1";"value2";"value3"'`.

insert_list(*value_list*)
Insert a `list` of values (instead of a `str` representing a line).

Example:

```
batch.insert_list(['value_1', 'value_2'])
```

Parameters **value_list** – A `list`. Each item should represent a column value.

insert_raw(**args*, ***kwargs*)
Insert a raw `str`.

A raw `str` does not have to be a complete row, but can be a part of a row or even multiple rows. This is useful when you have a file that is already in a format readable by Vertica.

rollback()
Rollback the current transaction.

4.3 Base importer class

class pyvertica.importer.**BaseImporter**(*reader_obj*, *schema_name*, *batch_source_path*, *odbc_kwargs={}*, ***kwargs*)

Base class for importing data into Vertica.

Note, before using this base importer, make sure you have created the history table for batch imports:

```
CREATE TABLE meta.batch_history (
    batch_source_name VARCHAR(255),
    batch_source_type_name VARCHAR(255),
    batch_source_path VARCHAR(255),
    batch_import_timestamp TIMESTAMP
)
```

Usage example:

```
class AdGroupPerformanceReportImporter(BaseImporter):
    table_name = 'adwords_ad_group_performance'
    batch_source_name = 'adwords_api'
    batch_source_type_name = 'ad_group_performance_report'

    mapping_list = (
        {
            'field_name': 'AccountCurrencyCode',
            'db_field_name': 'account_currency_code',
            'db_data_type': 'VARCHAR(10)',
        },
        {
            'field_name': 'AccountDescriptiveName',
            'db_field_name': 'account_descriptive_name',
            'db_data_type': 'VARCHAR(512)',
        },
        {
            'field_name': 'AccountTimeZoneId',
            'db_field_name': 'account_time_zone_id',
            'db_data_type': 'VARCHAR(100)',
        },
        ...
    )

iterable_object = [
    {
        'AccountCurrencyCode': 'EUR',
        'AccountDescriptiveName': 'Test account description',
        'AccountTimeZoneId': '(GMT+01:00) Amsterdam,'
    },
    ...
]

report_importer = AdGroupPerformanceReportImporter(
    iterable_object,
    dsn='VerticaTST',
    schema_name='test',
    batch_source_path='ADGROUP_PERFORMANCE_REPORT.1234.20120521',
)
report_importer.start_import()
```

In the example above, we are importing a `list` of dicts. More likely, this `iterable_object` would be a

reader class for your data-source, which is iterable and would return a `dict` with the expected fields‘.

Parameters

- **reader_obj** – An object that is iterable and returns for every line the data as a `dict`.

  Note: Passing a `list` of `dict` objects will work as well.

- **odbc_kwargs** – A `dict` containing the ODBC connection keyword arguments. E.g.:

  ```
  {
      'dsn': 'TestDSN',
  }
  ```

  See Also:

  https://code.google.com/p/pyodbc/wiki/Module

- **schema_name** – Name of the DB schema to use.
- **batch_source_path** – A `str` describing the path to the source. This can be a file path when importing from a file, or an identifier when the source is an API. This should be unique for every import!
- **kwargs** – Optional extra keyword arguments, will be stored as `self._kwargs`.

batch_history_table = ‘meta.batch_history’

Name of the database table containing the batch history (including the schema name) (`str`). The structure of this table is:

```
batch_source_name VARCHAR(255)
batch_source_type_name VARCHAR(255)
batch_source_path VARCHAR(255)
batch_import_timestamp TIMESTAMP
```

batch_source_name = ‘’

The name of the source which the data is retrieved from. E.g.: for AdWords, this this could be something like `'adwords_api'`.

batch_source_type_name = ‘’

The type of data that is imported from the source. E.g.: for the AdWords API, this could be something like `ADGROUP_PERFORMANCE_REPORT`.

extra_fields = ({‘db_data_type’: ‘VARCHAR(255)’, ‘field_name’: ‘batch_source_name’}, {‘db_data_type’: ‘VARCH

A `tuple` of `dict` objects to prepend fields to the data.

This list enables the possibility to add extra data to the database, for example data related to the import (source name, source identifier, import timestamp, ...).

Each `dict` must contain the following keys:

field_name A `str` representing the DB field name.

db_data_type A `str` representing the field type in the database, eg: `'varchar(10)'`.

Then, for every field, you should define a method within your class which is is named following this template: `get_extra_{field_name}_data`. This method will be called for every imported record with a `dict` containing the row data.

Warning: Make sure there is no collision between these fields and the fields defined in `mapping_list`.

classmethod get_batch_source_path_exists(*batch_source_path*, *odbc_kwargs={}*)
Check if the batch source-path exists in the database.

Parameters

- **batch_source_path** – The batch source-path (`str`).
- **odbc_kwargs** – A `dict` containing the ODBC connection keyword arguments. E.g.:

```
{
    'dsn': 'TestDSN',
}
```

See Also:

https://code.google.com/p/pyodbc/wiki/Module

Returns `True` if it already exists, else `False`.

get_extra_batch_import_timestamp_data(*row_data_dict*)
Return batch import timestamp.

Parameters **row_data_dict** – A `dict` containing the row-data.

Returns A `str` in ISO 8601 format, with a space a separator.

get_extra_batch_source_name_data(*row_data_dict*)
Return batch source name.

Parameters **row_data_dict** – A `dict` containing the row-data.

Returns The value set in `batch_source_name`.

get_extra_batch_source_path_data(*row_data_dict*)
Return batch source path.

Parameters **row_data_dict** – A `dict` containing the row-data.

Returns A `str` containing the batch-source path (this is given as the `batch_source_path` argument on constructing this class).

classmethod get_last_imported_batch_source_path(*odbc_kwargs*)
Return the last imported batch source-path.

Parameters **odbc_kwargs** – A `dict` containing the ODBC connection keyword arguments. E.g.:

```
{
    'dsn': 'TestDSN',
}
```

See Also:

https://code.google.com/p/pyodbc/wiki/Module

Returns A `str` representing the last imported batch source-path.

get_sql_create_table_statement()
Return SQL statement for creating the DB table.

This will first use the fields specified in `extra_fields`, followed by the fields in `mapping_list`.

Returns A `str` representing the SQL statement.

mapping_list = ()
A `tuple` of `dict` objects to map record columns to db columns.

The fields specified in this list might be a sub-set of the available fields in the record. Only specify the fields you want to store in the DB.

Each `dict` must contain the following keys:

field_name A `str` representing the field name as it is in the `dict` containing the data (as returned by the `reader_obj`).

db_data_type A `str` representing the field type in the database, eg: `'varchar(10)'`.

Optionally, each `dict` can contain the following keys:

db_field_name A `str` representing the field name within the database. This only needs to be set when it does not match with the source field name.

start_import()
Start the import.

This will import all the data from the `reader_obj` argument (given when constructing `BaseImporter`). In case there are no errors, it will commit the import at the end.

Raises `BatchImportError` when there are errors during the import. Errors are logged to the logger object.

Raises `AlreadyImportedError` when there already an import exists with the same batch-source path. Before starting your import, you can test this by calling `get_batch_source_path_exists()`.

table_name = ''
Name of the database table (excluding the schema) (`str`).

4.4 Exceptions

exception pyvertica.importer.**BatchImportError**
Exception is raised when the batch import is returning errors.

exception pyvertica.importer.**AlreadyImportedError**
Exception is raised when trying to import the same source twice.

exception pyvertica.migrate.**VerticaMigratorError**
Error specific to the `pyvertica.migrate` module.

Python Module Index

p