Teradata Python Module

Reference by ericscheie on 26 Jul 2015

The Teradata Python Module is a freely available, open source, library for the Python programming language, whose aim is to make it easy to script powerful interactions with Teradata Database. It adopts the philosophy of udaSQL, providing a DevOps focused SQL Execution Engine that allows developers to focus on their SQL and procedural logic without worrying about Operational requirements such as external configuration, query banding, and logging.

The Teradata Python Module is released under an MIT license. The source is available on GitHub and the package is available for download and install from PyPI.

Table of Contents

1.0 Getting Started

The following sections run through installation, connectivity options, and a simple Hello World example.

1.1 Installing the Teradata Python Module

The Teradata Python Module has been certified to work with Python 3.4+ / 2.7+, Windows/Linux/Mac, 32/64 bit.

The easiest way to install the “teradata” python module is using pip.

```
pip install teradata
```

If you don’t have pip installed, you can download the package from PyPI, unzip the folder, then double click the

setup.py file or run

```
setup.py install
```

1.2 Connectivity Options

The Teradata Python Module can use either the REST API for Teradata Database or Teradata ODBC to connect to Teradata. If using the REST API, make sure Teradata REST Services (tdrestd) is deployed and the target Teradata system is registered with the Service. If using ODBC, make sure the Teradata ODBC driver is installed on the same machine as where the Teradata Python Module will be executed.

The Teradata Python Module includes two sub-modules that implement the Python Database API Specification v2.0, one using REST (teradata.tdrest) and one using ODBC (teradata.tdodbc). Though these modules can be accessed directly, its recommended to use the base UdaExec module instead as it provides all the extra DevOps enabled features.

1.3 Hello World Example

In this example, we will connect to a Teradata Database and run a simple query to fetch the Query Band information for the session that we create.

Example 1 - HelloWorld.py

```
1  import teradata
2
3  udaExec = teradata.UdaExec (appName="HelloWorld", version="1.0",
4          logConsole=False)
5
6  session = udaExec.connect(method="odbc", system="tdprod",
7          username="xxx", password="xxx");
8
9  for row in session.execute("SELECT GetQueryBand()"):
10     print(row)
```

Let's break the example down line by line. The first line, "import teradata", imports the Teradata Python Module for use in the script.

The second line initializes the "UdaExec" framework that provides DevOps support features such as configuration and logging. We tell UdaExec the name and version of our application during initialization so that we can get feedback about our application in DBQL and Teradata Viewpoint as this information is included in the QueryBand of all Database sessions created by our script. We also tell UdaExec not to log to the console (e.g. logConsole=False) so that our print statement is easier to read.

The third line creates a connection to a Teradata system named "tdprod" using ODBC as the connection method. The last line executes the "SELECT GetQueryBand()" SQL statement and iterates over the results, printing each row returned. Since "SELECT GetQueryBand()" statement only returns one row, only one row is printed.

Let's go ahead and run the script by executing "python HelloWorld.py". Below is the result:

```
Row 1: ['=S> ApplicationName=HelloWorld;Version=1.0;JobID=1;ClientUser=example;Production=false;
udaAppLogFile=/home/example/PyTd/Example1/logs/HelloWorld.20150608153012-1.log;gitRevision=f4cc453;
gitDirty=False;UtilityName=PyTd;UtilityVersion=15.10.00.00;']
```

From the output, we see that one row was returned with a single string column. We also see quite a bit of information was added to the QueryBand of the session we created. We can see the application name and version we specified when initializing UdaExec as well as the name of a log file. If we look at this location on the file system we can see the log file that was generated:

```
1  2015-06-24 16:30:47,875 - teradata.udaexec - INFO - Initializing UdaExec...
2  2015-06-24 16:30:47,875 - teradata.udaexec - INFO - Reading config files: ['/etc/udaexec.ini: Not Found
3  2015-06-24 16:30:47,875 - teradata.udaexec - INFO - No previous run number found as /home/example/PyTd/
4  2015-06-24 16:30:47,875 - teradata.udaexec - INFO - Cleaning up log files older than 90 days.
5  2015-06-24 16:30:47,875 - teradata.udaexec - INFO - Removed 0 log files.
6  2015-06-24 16:30:47,883 - teradata.udaexec - INFO - Checkpoint file not found: /home/example/PyTd/Examp
7  2015-06-24 16:30:47,883 - teradata.udaexec - INFO - No previous checkpoint found, executing from beginn
8  2015-06-24 16:30:47,884 - teradata.udaexec - INFO - Execution Details:
9  /*******************************************************************************
10  * Application Name: HelloWorld
11  *          Version: 1.0
12  *       Run Number: 20150624163047-1
13  *             Host: sdlc4157
14  *         Platform: Linux-2.6.32-431.el6.x86_64-x86_64-with-centos-6.5-Final
15  *          OS User: example
16  *   Python Version: 3.4.3
17  *  Python Compiler: GCC 4.4.7 20120313 (Red Hat 4.4.7-11)
18  *     Python Build: ('default', 'Apr  7 2015 16:47:35')
19  *  UdaExec Version: 15.10.00.00
20  *     Program Name: HelloWorld.py
21  *      Working Dir: /home/example/PyTd/Example1
22  *      Git Version: git version 1.7.1
23  *     Git Revision: f4cc453
24  *        Git Dirty: True [ M Example1/HelloWorld.py,?? Example2/]
25  *          Log Dir: /home/example/PyTd/Example1/logs
26  *         Log File: /home/example/PyTd/Example1/logs/HelloWorld.20150624163047-1.log
27  *     Config Files: ['/etc/udaexec.ini: Not Found', '/home/example/udaexec.ini: Not Found', '/home/exa
28  *      Query Bands: ApplicationName=HelloWorld;Version=1.0;JobID=20150624163047-1;ClientUser=example;P
29  *******************************************************************************/
30 2015-06-24 16:30:47,884 - teradata.udaexec - INFO - Creating connection: {'password': 'XXXXXX', 'method
31 2015-06-24 16:30:47,884 - teradata.tdodbc - INFO - Loading ODBC Library: libodbc.so
32 2015-06-24 16:30:48,131 - teradata.udaexec - INFO - Connection successful. Duration: 0.246 seconds. Det
33 2015-06-24 16:30:48,132 - teradata.udaexec - INFO - Query Successful. Duration: 0.001 seconds, Rows: 1,
34 2015-06-24 16:30:48,133 - teradata.tdodbc - WARNING - 1 open connections found on exit, attempting to c
35 2015-06-24 16:30:48,185 - teradata.udaexec - INFO - UdaExec exiting.
```

In the logs, you can see connection information and all the SQL statements submitted along with their durations. If any errors had occurred, those would have been logged too.

The second to last log entry is a WARNING message that an open connection was not explicitly closed. Explicitly closing resources when done is always a good idea. In the next sections, we show how this can be done automatically using the "with" statement.

2.0 DevOps Features

The following sections discuss the DevOps oriented features provided by the Teradata Python Module. These features help simplify development and provide the feedback developers need once their applications are put into QA and production.

2.1 External Configuration

In the first "Hello World" example, we depended on no external configuration information for our script to run. What if we now wanted to run our HelloWorld.py script against a different database system? We would need to modify the source of our script, which is somewhat inconvenient and error prone. Luckily the UdaExec framework makes it easy to maintain configuration information outside of our source code.

Example 2 – PrintTableRows.py

```
1  import teradata
2
3  udaExec = teradata.UdaExec ()
4
5  with udaExec.connect("${dataSourceName}") as session:
6      for row in session.execute("SELECT * FROM ${table}"):
7          print(row)
```

In this example, we remove all the hard coded configuration data and instead load our configuration parameters from external configuration files. We also call connect using the "with" statement so that the connection is closed after use even when exceptions are raised.

You may be wondering what *${dataSourceName}* means above. Well, a dollar sign followed by optional curly braces means replace *${whatever}* with the value of the external configuration variable named "whatever". In this example, we make a connection to a data source whose name and configuration is defined outside of our script. We then perform a SELECT on a table whose name is also configured outside of our script.

UdaExec allows any SQL statement to make reference to an external configuration parameter using the dollar sign/curly brace syntax. When actually wanting to include a "$" literal in a SQL statement that isn't a parameter substitution, you must escape the dollar sign with another dollar sign (e.g. "$$").

Here is our external configuration file that we name "udaexec.ini" and place in the same directory as our python script.

Example 2 - udaexec.ini

```
1  # Application Configuration
2  [CONFIG]
3  appName=PrintTableRows
4  version=2
5  logConsole=False
6  dataSourceName=TDPROD
7  table=DBC.DBCInfo
8
9  # Default Data Source Configuration
10 [DEFAULT]
11 method=odbc
12 charset=UTF8
13
14 # Data Source Definition
15 [TDPROD]
16 system=tdprod
17 username=xxx
18 password=xxx
```

An external configuration file should contain one section named "CONFIG" that contains application configuration name/value pairs, a section named "DEFAULT" that contains default data source name/value pairs, and one or more user defined sections that contain data source name/value pairs.

In this example, we are connecting to *${dataSourceName}*, which resolves to "TDPROD" as dataSourceName is a property in the CONFIG section. The TDPROD data source is defined in our configuration file and provides the name of the system we are connecting to as well as the username and password. It also inherits the properties in the DEFAULT section, which in this case, defines that we will use ODBC as the connection method and "UTF8" as the session character set.

You'll notice in this example we didn't specify the "appName" and "version" when initializing UdaExec. If you look at the method signature for UdaExec, you'll see that the default values for appName and version are "${appName}" and "${version}". When not specified as method arguments, these values are looked up in the external configuration. This is true for almost all configuration parameters that can be passed to the UdaExec constructor so that any setting can be set or changed without changing your code.

If we run the example script above using "python PrintTableRows.py", we get the following output:

```
Row 1: ['LANGUAGE SUPPORT MODE', 'Standard']
Row 2: ['RELEASE', '15.00.01.02']
Row 3: ['VERSION', '15.00.01.02']
```

Looking at the generated log file, we see the following log entry:

```
2015-06-08 16:54:55,728 - teradata.udaexec - INFO - Reading config files: ['/etc/udaexec.ini: Not Found',
'/home/example/udaexec.ini: Not Found', '/home/example/PyTd/Example2/udaexec.ini: Found']
```

As you can see, UdaExec is attempting to load external configuration from multiple files. By default, UdaExec looks for a system specific configuration file, a user specific configuration file, and an application specific configuration file. The location of these files can be specified as arguments to the UdaExec constructor. Below are the argument names along with their default values.

Table 1 – Config File Locations

Name	Description	Default Value
systemConfigFile	The system wide configuration file(s). Can be a single value or a list.	*"/etc/udaexec.ini"*

userConfigFile	The user specific configuration file(s). Can be a single value or a list.	*"~/udaexec.ini" or "%HOMEPATH%/udaexec.ini"*
appConfigFile	The application specific configuration file (s). Can be a single value or a list.	*"udaexec.ini"*

Configuration data is loaded in the order shown above, from least specific to most specific, with later configuration files overriding the values specified by earlier configuration files when conflicts occur.

If we had wanted to name our configuration file in this example "PrintTableRows.ini" instead of "udaexec.ini", then we could've specified that when creating the UdaExec object. E.g.

```
udaExec = teradata.UdaExec (appConfigFile="PrintTableRows.ini")
```

If we wanted to have multiple application configuration files, then we could've specified a list of file names instead. E.g.

```
udaExec = teradata.UdaExec (appConfigFile=["PrintTableRows.ini", "PrintTableRows2.ini"])
```

If you find that even that isn't flexible enough, you can always override the external configuration file list used by UdaExec by passing it in the "configFiles" argument. When the "configFiles" list is specified, systemConfigFile, userConfigFile, and appConfigFile values are ignored.

In addition to using external configuration files, application configuration options can also be specified via the command line. If we wanted to change the table name we select from in the example above, we can specify the table value on the command line e.g. "python PrintTableRows.py --table=ExampleTable" which would instead print the rows of a table named "ExampleTable". Configuration options specified on the command line override those in external configuration files. UdaExec has a parameter named "parseCmdLineArgs" that is True by default. You can set this value to False to prevent command line arguments from being included as part of the UdaExec configuration.

Sometimes it may be necessary to get or set UdaExec application configuration parameters in the code directly. You can do this by using the "config" dictionary-like object on the UdaExec instance. E.g.

```
udaExec = teradata.UdaExec ()
print(udaExec.config["table"])
udaExec.config["table"] = "ExampleTable"
```

As you can see, using external configuration makes it easy to write scripts that are reasonably generic and that can execute in a variety of environments. The same script can be executed against a Dev, Test, and Prod environment with no changes, making it easier to adopt and automate a DevOps workflow.

2.2 Logging

The UdaExec object automatically enables logging when it is initialized. Logging is implemented using Python's standard logging module. If you create a logger in your script, your custom log messages will also be logged along with the UdaExec log messages.

By default, each execution of a script that creates the UdaExec object gets its own unique log file. This has the potential to generate quite a few files. For this reason, UdaExec also automatically removes log files that are older than a configurable number of days.

Below is a list of the different logging options and their default values. Logging options can be specified in the UdaExec constructor, in the application section of external configuration files, or on the command line.

Table 2 – Logging Options

Name	Description	Default Value
configureLogging	Flags if UdaExec will configure logging.	True
logDir	The directory that contains log files.	*"logs"*
logFile	The log file name.	*"${appName}.${runNumber}.log"*
logLevel	The level that determines what log messages are logged (i.e. CRITICAL, ERROR, WARNING, INFO, DEBUG, TRACE)	*"INFO"*
logConsole	Flags if logs should be written to stdout in addition to the log file.	True
logRetention	The number of days to retain log files. Files in the log directory older than the specified number of days are deleted.	90

If the logging features of UdaExec don't meet the requirements of your application, then you can configure UdaExec not to configure logging and instead configure it yourself.

Log messages generated at INFO level contain all the status of all submitted SQL statements and their durations. If there are problems during script execution, the log files provide the insight needed to diagnose any issues. If more information is needed, the log level can be increased to *"DEBUG"* or *"TRACE"*.

2.3 Checkpoints

When an error occurs during script execution, exceptions get raised that typically cause the script to exit. Let's suppose you have a script that performs 4 tasks but it is only able to complete 2 of them before an unrecoverable exception is raised. In some cases, it would be nice to be able to re-run the script when the error condition is resolved and have it automatically resume execution of the 2 remaining tasks. This is exactly the reason UdaExec includes support for checkpoints.

A checkpoint is simply a string that denotes some point during script execution. When a checkpoint is reached, UdaExec saves the checkpoint string off to a file. UdaExec checks for this file during initialization. If it finds a previous checkpoint, it will ignore all execute statements until the checkpoint specified in the file is reached.

Example 3 - CheckpointExample.py

```
import teradata

udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}") as session:
    session.execute("-- Task 1")
    udaExec.checkpoint("Task 1 Complete")

    session.execute("-- Task 2")
    udaExec.checkpoint("Task 2 Complete")

    session.execute("-- Task 3")
    udaExec.checkpoint("Task 3 Complete")

    session.execute("-- Task 4")
    udaExec.checkpoint("Task 4 Complete")


# Script completed successfully, clear checkpoint
# so it executes from the beginning next time
udaExec.checkpoint()
```

In the example above, we are calling execute 4 different times and setting a checkpoint after each call. If we were to re-run the script after the 3rd execute failed, the first two calls to execute would be ignored. Below are the related log entries when re-running our CheckpointExample.py script after the 3rd execute failed.

```
2015-06-25 14:15:29,017 - teradata.udaexec - INFO - Initializing UdaExec...
2015-06-25 14:15:29,026 - teradata.udaexec - INFO - Found checkpoint file: "/home/example/PyTd/Example3
2015-06-25 14:15:29,027 - teradata.udaexec - INFO - Resuming from checkpoint "Task 2 Complete".
2015-06-25 14:15:29,028 - teradata.udaexec - INFO - Creating connection: {'method': 'odbc', 'system': '
2015-06-25 14:15:29,250 - teradata.udaexec - INFO - Connection successful. Duration: 0.222 seconds. Det
2015-06-25 14:15:29,250 - teradata.udaexec - INFO - Skipping query, haven't reached resume checkpoint y
2015-06-25 14:15:29,250 - teradata.udaexec - INFO - Skipping query, haven't reached resume checkpoint y
2015-06-25 14:15:29,250 - teradata.udaexec - INFO - Reached resume checkpoint: "Task 2 Complete". Resu
2015-06-25 14:15:29,252 - teradata.udaexec - INFO - Query Successful. Duration: 0.001 seconds, Rows: 0,
2015-06-25 14:15:29,252 - teradata.udaexec - INFO - Reached checkpoint: "Task 3 Complete"
2015-06-25 14:15:29,252 - teradata.udaexec - INFO - Saving checkpoint "Task 3 Complete" to /home/exampl
2015-06-25 14:15:29,253 - teradata.udaexec - INFO - Query Successful. Duration: 0.001 seconds, Rows: 0,
2015-06-25 14:15:29,254 - teradata.udaexec - INFO - Reached checkpoint: "Task 4 Complete"
2015-06-25 14:15:29,254 - teradata.udaexec - INFO - Saving checkpoint "Task 4 Complete" to /home/exampl
2015-06-25 14:15:29,328 - teradata.udaexec - INFO - Clearing checkpoint....
2015-06-25 14:15:29,329 - teradata.udaexec - INFO - Removing checkpoint file /home/example/PyTd/Example
2015-06-25 14:15:29,329 - teradata.udaexec - INFO - UdaExec exiting.
```

As you can see from the logs, all calls to execute are skipped until the "Task 2 Complete" checkpoint is reached. At the end of our script we call "udaExec.checkpoint()" without a checkpoint string. This call clears the checkpoint file so that the next time we run our script, it will execute from the beginning.

While skipping calls to execute help to resume after an error, there are situations where this alone will not always work. If the results of a query are necessary for program execution, then the script may hit additional errors when being resumed. For example, let's assume our script now loads a configuration parameter from a table.

```
udaExec.config["mysetting"] = session.execute("SELECT mysetting FROM
    MyConfigTable").fetchone()[0]
```

A call to execute returns a Cursor into a result set, so we call fetchone()[0] to get the first column of the first row in the result set. If the execute call is skipped, then fetchone() will return None and the lookup of the first column will fail. There are several ways we can workaround this problem. The first way is to force execute to run regardless of checkpoints by specifying the parameter runAlways=True. E.g.

```
udaExec.config["mysetting"] = session.execute("SELECT mysetting FROM
    MyConfigTable", runAlways=True).fetchone()[0]
```

This is a good approach if we want to set "mysetting" even on resume. If "mysetting" is not necessary for resume though, then another way to prevent errors is to check the UdaExec "skip" attribute. E.g.

```
if not udaExec.skip:
    udaExec.config["mysetting"] = session.execute("SELECT mysetting FROM
    MyConfigTable").fetchone()[0]
```

With this approach, we only access the "mysetting" column if execute will not be skipped.

UdaExec saves checkpoints to a file named *"${appName}.checkpoint"* located in the same directory the script is executed by default. The checkpoint file can be changed by specifying the "checkpointFile" parameter in the UdaExec constructor, in an external configuration file, or on the command line. To disable file-based checkpoints, "checkpointFile" can be set to None in the UdaExec constructor or it can be set to an empty string in an external configuration file.

If it is desirable to load checkpoints from and save checkpoints to a place other than a local file (e.g. a database table), then a custom checkpoint manager implementation can be used to handle loading, saving, and clearing checkpoint details. Below is an example of a custom checkpoint manager that loads and saves checkpoints to a database table.

```
class MyCheckpointManager (teradata.UdaExecCheckpointManager):
    def __init__(self, session):
    self.session = session
    def loadCheckpoint(self):
        for row in self.session.execute("""SELECT * FROM ${checkPointTable}
                                           WHERE appName = '${appName}'"""):
            return row.checkpointName
    def saveCheckpoint(self, checkpointName):
        self.session.execute("""UPDATE ${checkPointTable} SET checkpointName = ?
                                WHERE appName = '${appName}' ELSE
                                INSERT INTO ${checkPointTable} VALUES ('${appName}', ?)""",
                             (checkpointName, checkpointName))
    def clearCheckpoint(self):
        self.session.execute("""DELETE FROM ${checkPointTable}
```

```
                                WHERE appName = '${appName}'""",
                            ignoreErrors=[3802])
```

To use this custom checkpoint manager, you can disable the checkpointFile and call the setCheckpointManager method on UdaExec. E.g.

```
udaexec = teradata.UdaExec(checkpointFile=None)
with udaexec.connect("${dsn}") as session:
    udaexec.setCheckpointManager(MyCheckpointManager(session))
    # The rest of my program logic.
```

2.4 Query Banding

UdaExec automatically sets session Query Bands for any connections you create so that the runtime characteristics of your application can be monitored in DBQL and Teradata Viewpoint. Reviewing application log files along with the associated log entries in DBQL are great ways to get feedback on the overall execution of your application. The table below lists the name and descriptions of the Query Bands that are set.

Table 3 - Query Bands

Name	Description
ApplicationName	The name of your application
Version	The version of your application
JobID	The run number of this particular execution
ClientUser	The OS user name.
Production	True if a production App, else False
udaAppLogFile	Path of the generated log file
gitRevision	The GIT revision of the application.
gitDirty	True if files have been modified since last commit to GIT
UtilityName	The nickname of the Teradata Python Module - PyTd
UtilityVersion	The version of the Teradata Python Module

Additional custom Query Bands can be set by passing a map (dict) as the queryBand argument to UdaExec.connect().

3.0 Database Interactions

UdaExec implements the Python Database API Specification v2.0 while adding additional convenience on top. The only deviation from this specification is that UdaExec enables auto commit by default. It is recommended to review the Python Database API Specification v2.0 first and then review the following sections for more details.

3.1 Cursors

Since only a single Cursor is needed most of the time, UdaExec creates an internal cursor for each call to connect() and allows execute, executemany, and callproc to be called directly on the connection object. Calls to these methods on the Connection object simply invoke those same methods on the internal cursor. The internal cursor is closed when the connection is closed.

Calls to execute, executemany, and callproc return the Cursor for convenience. Cursors act as iterators, so the results of an execute call can easily be iterated over in a “for” loop. Rows act like tuples or dictionaries, and even allow columns to be accessed by name similar to attributes on an object. Below is an example. All 3 print statements print the same thing for each row.

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}") as session:
for row in session.execute("""SELECT InfoKey AS name, InfoData as val
            FROM DBC.DBCInfo"""):
        print(row[0] + ": " + row[1])
        print(row["name"] + ": " + row["val"])
        print(row.name + ": " + row.val)
```

There are situations where it may be necessary to use a separate cursor in addition to the one created by default. A good example of this is when wanting to perform queries while iterating over the results of another query. To accomplish this, two cursors must be used, one to iterate and one to invoke the additional queries. Below is an example.

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}") as session:
    with session.cursor() as cursor:
    for row in cursor.execute("SELECT * from ${tableName}"):
            session.execute("DELETE FROM ${tableName} WHERE id = ?", (row.id, )):
```

Like connections, cursors should be closed when you're finished using them. This is best accomplished using the “with” statement.

3.2 Parameterized SQL

You can pass parameters to SQL statements using the question mark notation. The following example inserts a row into an employee table.

```
session.execute("""INSERT INTO employee (id, firstName, lastName, dob)
                   VALUES (?, ?, ?, ?)""", (1,"James", "Kirk", "2233-03-22"))
```

To insert multiple rows, executemany can be used. To insert them using batch mode, pass in the parameter batch=True. E.g.

```
session.executemany("""INSERT INTO employee (id, firstName, lastName, dob)
                       VALUES (?, ?, ?, ?)""",
                    ((1,"James", "Kirk", "2233-03-22"),
                     (2,"Jean-Luc", "Picard", "2305-07-13")),
                    batch=True)
```

Batch mode sends all the parameter sequences to the database in a single “batch” and is much faster than sending the parameter sequences individually.

3.3 Stored Procedures

Stored procedures can be invoked using the “callproc” method. OUT parameters should be specified as teradata.OutParam instances. INOUT parameters should be specified as teradata.InOutParam instances. An optional name can be specified with output parameters that can be used to access the returned parameter by name. Additionally, a data type name can be specified so that the output parameter is converted to the proper Python object. E.g.

```
results = session.callproc("MyProcedure", (teradata.InOutParam("inputValue", "inoutVar1"), teradata.OutP
print(results.inoutVar1)
print(results.outVar1)
```

3.4 Transactions

UdaExec enables auto commit by default. To disable auto commit and instead commit transactions manually, set autoCommit=False on the call to connect or in the data source's external configuration.

Transactions can be manually committed or rolled back using the commit() and rollback() methods on the Connection object. E.g.

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}", autoCommit=False) as session:
    session.execute("CREATE TABLE ${tableName} (${columns})")
    session.commit()
```

3.5 Data Types

To keep a consistent interface and implementation for both REST and ODBC, UdaExec gets all data values, with the exception of binary data (e.g. BYTE, VARBYTE, BLOB), in their string representation before converting them to their Python representation.

The interface that UdaExec uses to perform the conversion is called teradata.datatypes.DataTypeConverter with the default implementation being teradata.datatypes.DefaultDataTypeConverter. If you would like to customize how data gets converted from strings to Python objects, you can specify a custom DataTypeConverter during connect. E.g.

```
udaExec.connect("${dataSourceName}", dataTypeConverter=MyDataTypeConverter())
```

It is recommended to derive your custom DataTypeConverter from DefaultDataTypeConverter so that you can perform conversion for the data types you're interested in while delegating to the default implementation for any of the remaining ones.

The table below specifies the data types that get converted by the DefaultDataTypeConverter. Any data types not in the table below are returned as a Python Unicode string (e.g. VARCHAR, CLOB, UDT, ARRAY, etc.)

Data Type	**Python Object**
BYTE	bytearray
VARBYTE	bytearray
BYTEINT	decimal.Decimal
SMALLINT	decimal.Decimal
INTEGER	decimal.Decimal
BIGINT	decimal.Decimal
REAL, FLOAT, DOUBLE PRECISION	decimal.Decimal
DECIMAL, NUMERIC	decimal.Decimal

NUMBER	decimal.Decimal
DATE	datetime.date
TIME	datetime.time
TIME WITH TIME ZONE	datetime.time
TIMESTAMP	datetime.datetime
TIMESTAMP WITH TIME ZONE	datetime.datetime
INTERVAL	teradata.datatypes.Interval
BLOB	bytearray
JSON	dict or list, result of json.loads()
PERIOD	teradata.datatypes.Period

3.6 Unicode

The Teradata Python Module supports Unicode by default but you must make sure your session character set is set to UTF8 or UTF16 to successfully submit or retrieve Unicode data. If this is not the default, you can explicitly set your session character set by passing in "charset=UTF8" into the connect method or by specifying it in your data sources external configuration.

3.7 Ignoring Errors

Sometimes it is necessary to execute a SQL statement even though there is a chance it may fail. For example, if your script depends on a table that may or may not already exist, the simple thing to do is to try to create the table and ignore the "table already exists" error. UdaExec makes it easy to do this by allowing clients to specify error codes that can safely be ignored. For example, the following execute statement will not raise an error even if the checkpoints table already exists.

```
session.execute("""CREATE TABLE ${dbname}.checkpoints (
    appName VARCHAR(1024) CHARACTER SET UNICODE,
    checkpointName VARCHAR(1024) CHARACTER SET UNICODE)
    UNIQUE PRIMARY INDEX(appName)""",
    ignoreErrors=[3803])
```

If you want to ignore all errors regardless of the error code, you can include the "continueOnError=True" parameter to execute. This will cause any errors to be caught and logged and not raised up to your application.

3.8 Password Protection

Teradata ODBC along with Teradata Wallet can be used to avoid storing passwords in clear text in external configuration files. As UdaExec uses dollar signs to reference external configuration values, dollar signs used to reference Teradata Wallet keys must be escaped with an extra dollar sign. E.g.

```
udaExec.connect("${dataSourceName}", password="$$tdwallet(password_$$(tdpid)")
```

3.9 Query Timeouts

The execute, executemany, and callproc methods all accept a queryTimeout parameter for specifying the number of seconds to wait for the query to return. If the query does not complete within the specified timeout, it is aborted and an exception will be raised. E.g.

```
session.execute("SELECT * FROM ${table}", queryTimeout=60)
```

3.10 External SQL Scripts

UdaExec can be used to execute SQL statements that are stored in files external to your Python script. To execute the SQL statements in an external file, simply pass the execute method the location of the file to execute. E.g.

```
session.execute(file="myqueries.sql")
```

A semi-colon is used as the default delimiter when specifying multiple SQL statements. Any occurrence of a semi-colon outside of a SQL string literal or comments is treated as a delimiter. When SQL scripts contain SQL stored procedures that contain semi-colons internal to the procedure, the delimiter should be change to something other than the default. To use a different character sequence as the delimiter, the delimiter parameter can be used. E.g.

```
session.execute(file="myqueries.sql", delimiter=";;")
```

UdaExec also has limited support for executing BTEQ scripts. Any BTEQ commands starting with a "." are simply ignored, while everything else is treated as a SQL statement and executed. To execute a BTEQ script, pass in a fileType=*"bteq"* parameter. E.g.

```
session.execute(file="myqueries.bteq", fileType="bteq")
```

SQL statements in external files can reference external configuration values using the *${keyname}* syntax.

Therefore, any use of "$" in an external SQL file must be escaped if it is not intended to reference an external configuration value.

Any parameters passed to execute will be passed as parameters to the SQL statements in the external file. Execute will still return a cursor when executing a SQL script, the cursor will point to the results of the last SQL statement in the file.

Comments can be included in SQL files. Multi-line comments start with "/*" and end with "*/". Single line comments start with "--". Comments are submitted to the database along with the individual SQL statements.

4.0 Reference

This section defines the full set of method parameters supported by the API.

4.1 UdaExec Parameters

UdaExec accepts the following list of parameters during initialization. The column labeled "E" flags if a parameter can be specified in an external configuration file.

Name	Description	E	Default Value
appName	The name of our application	Y	None - Required field
version	The version of our application	Y	None - Required field
checkpointFile	The location of the checkpoint file. Can be None to disable file-based checkpoints.	Y	*${appName}.checkpoint*
runNumberFile	The path of the file containing the previous runNumber.	Y	*.runNumber*
runNumber	A string that represents this particular execution of the python script. Used in the log file name as well as included in the Session QueryBand.	Y	*YYYYmmddHHMMSS-X*
configureLogging	Flags if UdaExec will configure logging.	Y	True
logDir	The directory that contains log files.	Y	*"logs"*
logFile	The log file name.	Y	*"${appName}.${runNumber}.log"*
logLevel	The level that determines what log messages are logged (i.e. CRITICAL, ERROR, WARNING, INFO, DEBUG, TRACE)	Y	*"INFO"*
logConsole	Flags if logs should be written to stdout in addition to the log file.	Y	True
logRetention	The number of days to retain log files. Files in the log directory older than the specified	Y	90

	number of days are deleted.		
systemConfigFile	The system wide configuration file(s). Can be a single value or a list.	N	*"/etc/udaexec.ini"*
userConfigFile	The user specific configuration file(s). Can be a single value or a list.	N	*"~/udaexec.ini" or "%HOMEPATH%/udaexec.ini"*
appConfigFile	The application specific configuration file (s). Can be a single value or a list.	N	*"udaexec.ini"*
configFiles	The full list of external configuration files. Overrides any values in systemConfigFile, userConfigFile, appConfigFile.	N	None
configSection	The name of the application config section in external configuration files.	N	*CONFIG*
parseCmdLineArgs	Flags whether or not to include command line arguments as part of the external configuration variables.	N	True
gitPath	The path to the GIT executable to use to include GIT information in the session QueryBand.	Y	Defaults to system path
production	Flags if this app is a production application, applies this value to session QueryBand.	Y	False
odbcLibPath	The path to the ODBC library to load.	Y	Defaults to OS specific library path
dataTypeConverter	The DataTypeConverter implementation to use to convert data types from their string representation to python objects.	N	datatypes.DefaultDataTypeConverter()

4.2 Connect Parameters

The following table lists the parameters that the UdaExec.connect() method accepts. With the exception of the "externalDSN" parameter, all the parameters below can be specified in the DEFAULT or named data source sections of external configuration files. While the externalDSN parameter cannot be specified directly in an external configuration file, it can reference the name of an external configuration variable using *${keyname}* syntax. The "Type" column indicates if a parameter is specific to a connectivity option, if it is blank it applies to all types.

When using ODBC as the connection method, any parameters passed to the connect method or specified in an external configuration that are not listed below will be automatically be appened to the connect string passed to the ODBC driver. For example, to reference a named data source defined in an odbc.ini file, you can simply call udaExec.connect(method="odbc", DSN="mydsn").

Name	Description	Type	Default Value
externalDSN	The name of the data source defined in external configuration files.		None - Optional
method	The type of connection to make. Possible values are "rest" or "odbc"		None - Required field
dbType	The type of system being connected to. The only supported option at the present release is "Teradata"		*Teradata*
system	The name of the system to connect. For ODBC it's the tdpid, for REST its the system alias configured in the REST service		None
username	The Database username to use to connect.		None
password	The Database password to use to connect.		None
host	The host name of the server hosting the REST service.	REST	None
port	The port number of REST Service	REST	Defaults to 1080 for http and 1443 for https
protocol	The protocol to use for REST connections (i.e. http or https). When using https,	REST	*http*
webContext	The web context of the REST service	REST	*/tdrest*
charset	The session character set (e.g. UTF8, UTF16, etc.)		None
database	The default database name to apply to the session		None
autoCommit	Enables or disables auto commit mode. When auto commit mode is disabled, transactions must be committed manually.		True

transactionMode	The transaction mode to use i.e. "Teradata" or "ANSI"		*Teradata*
queryBands	A map (dict) of query band key/value pairs to include the session's QueryBand.		None
dataTypeConverter	The DataTypeConverter implementation to use to convert data types from their string representation to python objects.		datatypes.DefaultDataTypeConverter()
sslContext	The ssl.SSLContext to use to establish SSL connections.	REST	None
verifyCerts	Flags if REST SSL certificate should be verified, ignored if sslContext is not None.	REST	True
****kwargs**	A variable number of name/value pairs to append to the ConnectString passed to SQLDriverConnect. For the full list of options supported by the Teradata ODBC driver, see the ODBC Driver for Teradata User Guide.	ODBC	None

4.3 Execute Parameters

The following table lists the parameters that the execute method accepts.

Name	Description	Default Value
query	The query to execute.	None, required if file is None
params	The list or tuple containing the parameters to pass in to replace question mark placeholders.	None
file	The path of an external script to execute.	None
fileType	The type of file to execute if different than a standard delimited SQL script (i.e. bteq)	None
delimiter	The delimiter character to use for SQL scripts.	*;*
runAlways	When True, the query or script will be executed regardless if the previous checkpoint has	False

	been reached.	
continueOnError	When True, all errors will be caught and logged but not raised up to the application.	False
ignoreErrors	The list or sequence of error codes to ignore.	None
queryTimeout	The number of seconds to wait for a response before aborting the query and returning.	0 - indicates wait indefinitely
logParamCharLimit	The maximum number of characters to log per query parameter. When a parameter exceeds the limit it is truncated in the logs and an ellipsis ("...") is appended.	80 characters per parameter
logParamFrequency	The amount of parameter sets to log when executemany is invoked. Setting this value to X means that every Xth parameter set will be logged in addition to the first and last parameter set. When this value is set to zero, no parameters are logged.	1 - all parameters sets are logged.

DISCUSSION

ulrich
51 comments
Joined 09/09

29 Jul 2015

Quite interesting module!

I was able to install it and get it running via the RestAPI from a mac.

One questions on the sql files option.

How can I place comments in these files?

-- seems to work

/* */ seems not to work

seems not to work

So question is if -- is the only valid comment

Also can you explain how git can be used.

Thanks Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ericscheie
68 comments
Joined 11/08

29 Jul 2015

"--" and "/* ... */" should both work. Comments are submitted to the database along with SQL statements. The python module does not parse them out. Therefore, the comment syntax is what is supported by the database.

Regarding git, if your application files are part of a git repoistory, then git revision information will be included in the query band of any sessions created by your application. Getting started with git is a bit out of scope for this document but if you have git installed you can create a new repository by running "git init", then "git add filename" to add files, then "git commit -m 'Message here.'" to commit the files to the repository.

UPDATE: A bug was found with using comments that may have caused the comment related issue you were having. The bug is fixed and a new 15.10.00.02 release of the python module is available.

cw171001
7 comments
Joined 05/09

30 Jul 2015

Hi Eric

This is great . It looks like a great dba utility language.

One question . I extract show table statements in bteq and export to files. It would nice to be able to do this in python but I cannot get show table to work

Non-working
```
import teradata
udaExec = teradata.UdaExec (appName="TEST");
print("Connecting to DEV_LIDB840")
# This does not work
with udaExec.connect("DEV_LIDB840") as session:
   for row in session.execute("SHOW SEL * FROM ${table}"):
      print(row)
```

ericscheie
68 comments
Joined 11/08

30 Jul 2015

Can you share the error that you are getting? It looks like your SQL syntax is incorrect. This works:

```
1 for row in conn.execute("SHOW TABLE ${table}"):
2     print(row)
```

cw171001
7 comments
Joined 05/09

31 Jul 2015

Hi Eric
Found the issue , it does work perfectly. It is the way the print command behaves when printing to screen. When I redirect to file its perfect.
I am going to re-write a gcfr export enviroment scipt to using python instead of bteq.

Script looks like this
```
import teradata
udaExec = teradata.UdaExec (appName="TEST");
with udaExec.connect("DEV_LIDB840") as session:
   for row in session.execute("SHOW TABLE
DWT04T_GCFR.GCFR_PROCESS"):
      print(row)
```

Output TDShow.py>J.txt
```
Row 1: [CREATE SET TABLE DWT04T_GCFR.GCFR_Process
,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
    (
     Process_Name VARCHAR(30) CHARACTER SET LATIN
NOT CASESPECIFIC NOT NULL,
     Description VARCHAR(240) CHARACTER SET UNICODE
NOT CASESPECIFIC NOT NULL,
     Process_Type BYTEINT NOT NULL,
     Ctl_Id SMALLINT NOT NULL,
     Stream_Key SMALLINT NOT NULL,
     In_DB_Name VARCHAR(128) CHARACTER SET UNICODE
NOT CASESPECIFIC,
     In_Object_Name VARCHAR(128) CHARACTER SET
UNICODE NOT CASESPECIFIC,
     Out_DB_Name VARCHAR(128) CHARACTER SET UNICODE
NOT CASESPECIFIC,
     Out_Object_Name VARCHAR(128) CHARACTER SET
UNICODE NOT CASESPECIFIC,
     Target_TableDatabaseName VARCHAR(128) CHARACTER
SET UNICODE NOT CASESPECIFIC,
     Target_TableName VARCHAR(128) CHARACTER SET
UNICODE NOT CASESPECIFIC,
     Temp_DatabaseName VARCHAR(128) CHARACTER SET
UNICODE NOT CASESPECIFIC,
     Key_Set_Id SMALLINT,
     Domain_Id SMALLINT,
     Code_Set_Id SMALLINT,
     Collect_Stats BYTEINT CHECK ( Collect_Stats  IN (NULL ,0
,1 ) ),
     Truncate_Target BYTEINT CHECK ( Truncate_Target  IN
(NULL ,0 ,1 ) ),
```

```
    Verification_Flag BYTEINT CHECK ( Verification_Flag  IN
(NULL ,0 ,1 ) ),
    File_Qualifier_Reset_Flag BYTEINT CHECK (
File_Qualifier_Reset_Flag  IN (NULL ,0 ,1 ) ),
    Update_Date DATE FORMAT 'YYYY-MM-DD' NOT NULL
DEFAULT DATE ,
    Update_User VARCHAR(30) CHARACTER SET UNICODE
NOT CASESPECIFIC NOT NULL DEFAULT USER ,
    Update_Ts TIMESTAMP(6) NOT NULL DEFAULT
CURRENT_TIMESTAMP(6))
UNIQUE PRIMARY INDEX ( Process_Name );]
```

Output to screen

```
W:\TeradataStudio\Python_Tests>TD_SHOW.py
Target_TableDatabaseName VARCHAR(128) CHARACTER SET
UNICODE NOT CASESPECIF
File_Qualifier_Reset_Flag BYTEINT CHECK (
File_Qualifier_Reset_Flag IN (N
Update_User VARCHAR(30) CHARACTER SET UNICODE NOT
CASESPECIFIC NOT NULL DE
UNIQUE PRIMARY INDEX ( Process_Name );]EFAULT
CURRENT_TIMESTAMP(6))
```

ericscheie
68 comments
Joined 11/08

31 Jul 2015

The reason the results of SHOW TABLE were not displayed properly on the terminal is because the newline characters in the result were mostly likely carriage returns instead of line feeds. To ensure it displays correctly, you can split the output based on the presence of any newline sequence and print the lines individually. E.g.

```
import re
for line in re.split("\r\n|\n\r|\r|\n", row[0]):
    print(line)
```

chillerm
9 comments
Joined 04/11

31 Jul 2015

Absolutely amazing share, thank you. Looking forward to using this in the future.

cr255014
5 comments
Joined 01/14

31 Jul 2015

Very nice article. I was able to connect to my 14.10 box using Python Teradata module.
I have observed a Wierd thing with the module. after I execute my query, the session remains in responding state even after I receivie all the rows from database, Is this is something known issue with the module?

ericscheie
68 comments
Joined 11/08

31 Jul 2015

I was able to reproduce the issue of the session staying in the responding state. I opened a bug for the issue and released a new version with the fix (15.10.00.03). Thanks for reporting it.

ulrich
51 comments
Joined 09/09

03 Aug 2015

Hi Eric,

yes, the issue with the comments is fixed.

Also git is working nicely - my question was not related to git usage itself but on how go get the git info into the logs. I was not sure if additional configuration is needed but obvoiusly it is not - very nice!

Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ratzesberger
3 comments
Joined 02/11

03 Aug 2015

Eric,
Awesome job!
All,
Time to drop shell scripts and bteq from your set of tools and switch to this python devops based approach. Everybody should be encourage to fork the lib on github and extend it!

twitter: @ratzesberger https://twitter.com/ratzesberger

overcaster
2 comments
Joined 02/15

04 Aug 2015

Hi!
When i try run **Example 1 - HelloWorld.py** , i receive an error :(

```
C:\Python34\python.exe C:/Users/avurbano/Documents/Work/pyt
Import successfully compelted
Traceback (most recent call last):
  File "C:/Users/avurbano/Documents/Work/python/teradata.py
    import teradata
  File "C:\Users\avurbano\Documents\Work\python\teradata.py
    udaExec = teradata.UdaExec(appName="HelloWorld", versio
AttributeError: 'module' object has no attribute 'UdaExec'
```

Module installation completed without errors, code completion works fine and show method udaExec.

ericscheie
68 comments
Joined 11/08

04 Aug 2015

@overcaster - I believe the problem is that you are naming your script "teradata.py". This conflicts with the namespace of the Teradata Python Module. Therefore, your script is looking for UdaExec class in your script and not in the Teradata Python Module. Renaming your script to something else should fix the problem.

overcaster
2 comments
Joined 02/15

05 Aug 2015

Yes, you absolutely right :) Looks all works fine, problems only under Linux. I think problem with ODBC driver. But under Windows all work fine.

ulrich
51 comments
Joined 09/09

05 Aug 2015

Eric,

I want to log the output of an query into the application log.
" If you create a logger in your script, your custom log messages will also be logged along with the UdaExec log messages." - I tried it but didn't get it working. Might be a very basic mistake I make. Can you share an example how to add own log messages to the application log?

And one additional requirement - not sure how to best solve it.

In case I write a generic application - e.g. to copy a table from one DB to a new DB (just as an example) - the application name would be static. But each call for different tables would need a different checkpoint file to avoid conflicts between different runs of the same applications for different tables. I solved this right now by adding a hash of the configuration file name to the application name but I thing an additional parameter like instance would be better. And additing this instance to the checkpoint file name. Just some thoughts...

Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ulrich
51 comments
Joined 09/09

05 Aug 2015

Solved the logging issue :-)

```
import logging

logger = logging.getLogger(__name__)

rc = 0
with session:
    for row in session.execute("""
MY SELECT which should return 0 rows for being correct for
"""):
        rc += 1
        logger.error(row)

if rc == 0:
    udaExec.checkpoint("Step 5 Complete")
else:
    logger.error("ERROR - new and old table structure are
    exit(8)
```

Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ericscheie
68 comments
Joined 11/08

05 Aug 2015

@ulrich - Glad you were able to get logging working. You can

change the name of the checkpoint file easily enough without changing the name of the application. The UdaExec constructor takes a checkpointFile name argument either via the constructor or from the external configuration file. Here's an example:

```
[CONFIG]
appName=MyApp
instanceName=MyInstance
checkpointFile=${appName}-${instanceName}.checkpoint
```

ulrich
51 comments
Joined 09/09

06 Aug 2015

Hi Eric,

works like a charm!

Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ulrich
51 comments
Joined 09/09

06 Aug 2015

Hi Eric,

minor question:
Can the `Working Dir be configiured?`

`No parameter is specified for this.`

As this holds the checkpoint files some admins might prefer not use the source code file dir for this.

First generic job is ready and tested. I learned a lot about the module and how to use it. Really nice package!

Ulrich

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ericscheie
68 comments
Joined 11/08

06 Aug 2015

@ulrich - Note that the working directory is actually the directory you are located in when you execute the script and not necessarily the directory that contains the script. You can change the location of the checkpoint file and log files using the following configuration.

```
[CONFIG]
appName=myapp
workingDir=/var/opt/myapp
checkpointFile=${workingDir}/${appName}.checkpoint
logDir=${workingDir}/log
```

Thanks for the feedback.

jegan.velappan
2 comments
Joined 08/15

06 Aug 2015

I'm trying to use the teradata package and when I try to execute udaExec.connect command, its giving me the below error. I'm using Python 2.7.5 and Mac OS X 10.9.5
My assumption is, system --> Teradata Server Name. I even tried with the default value in the page "tdprod" and getting the same error. So, not sure if Python is connecting with the Teradata Server. I have my Teradata ODBC in my MAC as I connect to the Server using Tableau and Studio Express without issues. Any help?

jegan.velappan
2 comments
Joined 08/15

06 Aug 2015

I'm trying to use the teradata package and when I try to execute udaExec.connect command, its giving me the below error. I'm using Python 2.7.5 and Mac OS X 10.9.5
My assumption is, system --> Teradata Server Name. I even tried with the default value in the page "tdprod" and getting the same error. So, not sure if Python is connecting with the Teradata Server.
I have my Teradata ODBC 15x in my MAC as I connect to the Server using Tableau and Studio Express without issues. Any help?
Given below is the error message:

>>> session = udaExec.connect(method="odbc", system="tdprod",username="user", password="pswd")
Traceback (most recent call last):
File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/logging/__init__.py", line 874, in emit
stream.write(fs % msg.encode("UTF-8"))
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position 207: ordinal not in range(128)
Logged from file udaexec.py, line 45

File "<stdin>", line 1
SyntaxError: EOL while scanning string literal

ericscheie
68 comments
Joined 11/08

06 Aug 2015

@jegan.velappan - I'm not able to reproduce your error using Mac OS X 10.10.2 and Python 2.7.10. Do you have any non-ascii characters in any of the strings you are passing to UdaExec.connect? If so, prepending them with "u" e.g. (u"MyString") should fix the problem. Otherwise, using Python3 should also resolve the problem.

ulrich
51 comments
Joined 09/09

07 Aug 2015

@eric thanks for the clarification!

The workaround / the solution worked fine.

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ulrich
51 comments
Joined 09/09

07 Aug 2015

@eric question - is native JDBC support also on the roadmap? Would have some advantages as it only requires JDBC driver distribution and not ODBC installation for clients who don't use the RestAPI (which is my favorite)...

feel free to donate bitcoin:12kgAUHFUqvG2sQgaRBXFhCwyf9HXdkGud

ericscheie
68 comments
Joined 11/08

07 Aug 2015

@urlich There are no plans at this time to also support JDBC as a connection option. If people don't want to install ODBC, then we recommend they use the REST API instead.

privet3711
6 comments
Joined 07/11

10 Aug 2015

Any idea how to pass parameters to session.executemany from pandas dataframe ?
=I need to load all data from pandas dataframe to teradata table

```
session.executemany("""INSERT INTO employee (id,
firstName, lastName, dob)

                    VALUES (?, ?, ?, ?)""",

                  ((1,"James", "Kirk",
"2233-03-22"),

                   (2,"Jean-Luc", "Picard",
"2305-07-13")),

                  batch=True)
```

cr255014
5 comments
Joined 01/14

12 Aug 2015

Can we utilize Utility sessions like FastExport, FastLoad and MultiLoad like we use in JDBC?

ericscheie
68 comments
Joined 11/08

12 Aug 2015

@cr255014 - Unforuntately, no. The connectivity methods used by the Teradata Python Module (ODBC/REST) do not support FastExport, FastLoad, or MultiLoad at this time.

privet3711
6 comments
Joined 07/11

13 Aug 2015

how do you pass Null as parameter

ericscheie
68 comments
Joined 11/08

13 Aug 2015

@privet3711 - The Python Built-in constant "None" is used to represent a SQL NULL.

privet3711
6 comments
Joined 07/11

13 Aug 2015

Issue with reading timestamp (3) I extract same data via teradata and pyodbc , 3rd column has type timestamp(3)
_____ here is my code _____

import teradata

import pyodbc

udaExec = teradata.UdaExec (appName="DfToTera", version="1.0",logConsole=False,logRetention=0,configureLogging=False)
session = udaExec.connect(method="odbc", system="oneview",username="xxx", password="xxx",database="ud183")

for row in session.execute("""SELECT * from test4 order by 1"""): print(row[2])

conn = pyodbc.connect('DRIVER={Teradata}; DBCNAME=oneview;UID=xxx;PWD=xxx; database=ud183',autocommit=True)

for row in conn.execute("""SELECT * from test4 order by 1"""): print(row[2])

_____
results below when selecting using teradata connection shows incorrect decimal point , any ideas ?

In[292]: for row in session.execute("""SELECT * from test4 order by 1"""):

print(row[2])

2015-07-01 12:18:32.000673

2013-09-10 10:34:08.000260

2014-09-18 15:27:31.000490

2014-06-25 11:59:59.000873

2015-01-16 17:30:43.000900

In[293]: for row in conn.execute("""SELECT * from test4 order by 1"""):

print(row[2])

2015-07-01 12:18:32.673000

2013-09-10 10:34:08.260000

2014-09-18 15:27:31.490000

2014-06-25 11:59:59.873000

2015-01-16 17:30:43.900000

ericscheie
68 comments
Joined 11/08

13 Aug 2015

@privet3711 - I opened a bug for the timestamp(3) milliseconds being reported incorrectly and released a new version with the fix (15.10.00.04). Thanks for reporting.

privet3711
6 comments
Joined 07/11

14 Aug 2015

when trying to export large panda dataframe I'm getting error like this
[Teradata][ODBC Teradata Driver] SQL request exceeds maximum allowed length of 1 MB
Do you know anyway around it except manauly splitting parameters array into smaller arrrays

df2 = pd.read_sql("select top 50000 story_id,story_id_moment,chng_ts,est, cast (chng_ts as date) dt1 from ver1_story_dtl_tbl",conn)
data = [tuple(x) for x in df2.to_records(index=False)]
con.executemany("insert into ud183.test4 values(?,?,?,?,?)",data,batch=True)

_____
DatabaseError: (0, u'[22001] [Teradata][ODBC Teradata Driver] SQL request exceeds maximum allowed length of 1 MB')

ericscheie
68 comments
Joined 11/08

14 Aug 2015

@privet3711 - No, there is no other way. Teradata ODBC limits batch inserts to only certain size and number of rows. Therefore, to continue using Batch mode, you will need to call executemany for chunks of data whose size/count are below this limit.

privet3711
6 comments
Joined 07/11

15 Aug 2015

Eric , just a comment , it would be very nice to see fastload / multiload capabilities added to the tool. Teradata is designed for

handling pretty large tables and right now we can't efficiently load data back to Teradata which limits out use of Python with Teradata and pushes us to use SAS instead. would really like to see something that loads pandas dataframe to teradata using fastload

chillerm
9 comments
Joined 04/11

18 Aug 2015

Anyone have advice or examples for compiling stored procedures? I've tried a few ways but am unable to get past it. Closest I've been able to come was trying to simply execute the definition from a file. The SP compiles just fine from SQL Assistant or Studio. Curious, on the last example, if I can just change the max object / string size?

Both of the below are executed with

x = session.execute(file='sql/sp_grnt_rvk_role.sql')

Trying the file directly:

File "C:\Python34\lib\site-packages\teradata-15.10.0.3-py3.4.egg\teradata\tdodbc.py", line 147, in checkStatus

teradata.api.DatabaseError: (3706, "[42000] [Teradata][ODBC Teradata Driver][Teradata Database] Syntax error: expected something between the beginning of the request and the 'DECLARE' keyword. ")

Triple quoting the replace statement within the file:

File "C:\Python34\lib\site-packages\teradata-15.10.0.3-py3.4.egg\teradata\tdodbc.py", line 147, in checkStatus
teradata.api.DatabaseError: (6724, '[HY090] [Teradata][ODBC Teradata Driver][Teradata Database] The object name is too long in NFD/NFC. ')

ericscheie
68 comments
Joined 11/08

18 Aug 2015

@chillerm - If your file contains a single replace statement that includes semi-colons, then I'm guessing the multi-statement parsing performed by the python module is what is causing the error. Since you actually don't want to use the semi-colon as the statement delimiter, try passing in a different character sequence as the delimiter, one that does not occur in your file. E.g.

x = session.execute(file='sql/sp_grnt_rvk_role.sql', delimiter=';;')

chillerm
9 comments
Joined 04/11

18 Aug 2015

Ok, so interestingly enough if I read my file in as a string

with open ('sql/sp_grnt_rvk_role.sql', 'r') as myfile:
data=myfile.read()
proc_def=str(data)
and then execute with
session.execute(proc_def):
it seems to work just fine.

chillerm
9 comments
Joined 04/11

18 Aug 2015

Sorry I didn't see your response. You're exactly right, I I was successful by replacing the delimiter.

alpanchino
6 comments
Joined 08/11

19 Aug 2015

Anyone can advice what I am doing wrong. Neither odbc or rest method is not working for my teradata python module (have tried for both of python versions 2.7.10 and 3.4.3):
TDExpress15.0.0.2_Sles10:cat TestRest.py

```
import teradata

udaExec = teradata.UdaExec(appName="Test", version="1.0", logConsole=False)
        session = udaExec.connect(method="rest", host="192.168.100.15", system="TD15", username="dbc", password="dbc");                    for row in session.execute("SELECT current_timestamp"):

   print(row)
```

```
TDExpress15.0.0.2_Sles10:~/tdch/python/python TestRest.py
Traceback (most recent call last):
  File "TestRest.py", line 7, in <module>
    for row in session.execute("SELECT current_timestamp"):
  File "/usr/local/lib/python2.7/site-packages/teradata/udaexec.py", line 514, in execute
    self.internalCursor.execute(query, params, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/teradata/udaexec.py", line 560, in execute
    self._execute(self.cursor.execute, query, params, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/teradata/udaexec.py", line 611, in _execute
    raise e
teradata.api.InterfaceError: (400, 'HTTP Status: 400, URL: /tdrest/systems/TD15/queries, Details:  {u\'message\': u\'Can not construct instance of com.teradata.rest.api.def.ResultFormat from String value \\\'array\\\': value not one of declared Enum instance names: [OBJECT, ARRAY, CSV]\\n at [Source: org.apache.catalina.connector.CoyoteInputStream@38465c9a; line: 1, column: 93] (through reference chain: com.teradata.rest.api.model.QueryRequest["format"])\'}')


TDExpress15.0.0.2_Sles10:cat TestOdbc.py


import teradata

udaExec = teradata.UdaExec(appName="Test", version="1.0", logConsole=False)
        session = udaExec.connect(method="odbc", system="default", username="dbc", password="dbc");
                                for row in session.execute("SELECT GetQueryBand()"):

    print(row)


TDExpress15.0.0.2_Sles10:~/tdch/python/python TestOdbc.py
Traceback (most recent call last):
  File "TestOdbc.py", line 5, in <module>
    session = udaExec.connect(method="odbc", system="default", username="dbc", password="dbc");
  File "/usr/local/lib/python2.7/site-packages/teradata/udaexec.py", line 137, in connect
    raise e
teradata.api.DatabaseError: (113, u"[08001] [Teradata][Unix system error]  113 Socket error - The Teradata server can't currently be reached over this network")
```

P.S. It seems ODBC works well:

```
TDExpress15.0.0.2_Sles10:/opt/teradata/client/15.00/odbc_64/bin/tdxodbc

Enter Data Source Name: tdlocal
Enter UserID: dbc
Enter Password:

Connecting with SQLConnect(DSN=tdlocal,UID=dbc,PWD=*)...

.....ODBC connection successful.

ODBC version        = -03.52.0000-
DBMS name           = -Teradata-
DBMS version        = -15.00.0002  15.00.00.02-
Driver name         = -tdata.so-
```

```
Driver version      = -15.00.00.00-
Driver ODBC version = -03.51-

(type quit to terminate adhoc)
Enter SQL string : select current_timestamp

Executing SQLExecDirect("select current_timestamp")...

SQL Statement [1]: 1 rows affected.

Current TimeStamp(6)
2015-08-19 21:35:24.600000+00:00

--------------------------------------------------------------------------------

Teradata REST service works as well:

TDExpress15.0.0.2_Sles10:~/tdch/python/test.py

#!/usr/bin/python

import requests

from requests.auth import HTTPBasicAuth

import json

import os

tdUser = os.environ.get('REST_USER', 'dbc')

tdPassword = os.environ.get('REST_PASSWORD', 'dbc')

restServer = "{0}:{1}".format(os.environ.get('REST_HOST',
'localhost'), os.environ.get('REST_PORT', '1080'))

response = requests.get('http://{}/tdrest/systems
/TD15/databases'.format(restServer),

auth=HTTPBasicAuth(tdUser, tdPassword),

headers={'content-type': 'application/json','Accept':
'application/vnd.com.teradata.rest-v1.0+json, */*; q=0.01'})

print(response.text)

print("status: {0}".format(response.status_code))

TDExpress15.0.0.2_Sles10:~/tdch/python/python test.py
[{"name":"All","dbKind":"U"},{"name":"console","dbKind":"U"},
{"name":"Crashdumps","dbKind":"U"},{"name":"DBC","dbKind":"U"},
{"name":"dbcmngr","dbKind":"D"},{"name":"Default","dbKind":"U"},
{"name":"EXTUSER","dbKind":"U"},
{"name":"financial","dbKind":"D"},
{"name":"labs_querygrid_lab2","dbKind":"U"},
{"name":"LockLogShredder","dbKind":"U"},
{"name":"manufacturing","dbKind":"D"},
{"name":"PUBLIC","dbKind":"U"},{"name":"qg2batt","dbKind":"D"},
{"name":"qg2demo","dbKind":"D"},
{"name":"querygrid_demo","dbKind":"D"},
{"name":"retail","dbKind":"D"},{"name":"Samples","dbKind":"D"},
{"name":"SQLJ","dbKind":"D"},{"name":"SysAdmin","dbKind":"U"},
{"name":"SYSBAR","dbKind":"D"},
{"name":"SYSJDBC","dbKind":"D"},
{"name":"SYSLIB","dbKind":"D"},
{"name":"SYSSPATIAL","dbKind":"D"},
{"name":"SystemFe","dbKind":"U"},
{"name":"SYSUDTLIB","dbKind":"D"},
{"name":"SYSUIF","dbKind":"D"},
```

{"name":"Sys_Calendar","dbKind":"U"},
{"name":"td01","dbKind":"U"},{"name":"TDCH","dbKind":"D"},
{"name":"TDPUSER","dbKind":"U"},
{"name":"TDQCD","dbKind":"D"},{"name":"TDStats","dbKind":"D"},
{"name":"tduser","dbKind":"U"},{"name":"tdwm","dbKind":"U"},
{"name":"TD_SERVER_DB","dbKind":"D"},
{"name":"TD_SYSFNLIB","dbKind":"D"},
{"name":"TD_SYSXML","dbKind":"D"},
{"name":"tpch","dbKind":"D"},
{"name":"transportation","dbKind":"D"},
{"name":"twm_md","dbKind":"D"},
{"name":"twm_results","dbKind":"D"},
{"name":"twm_source","dbKind":"D"},
{"name":"viewpoint","dbKind":"U"}]

status: 200

sdc
1 comment
Joined 06/15

19 Aug 2015

Is there a way to use the module to return a key which is generated as identity? The most expensive thing in my application right now is inserting a row which has an automatically generated key and then finding the just-generated key by making sure it is not in a list of keys that are referenced in other tables. This link suggests that the ODBC driver supports this kind of thing:
http://www.info.teradata.com/htmlpubs/DB_TTU_13_10
/index.html#page/Connectivity/B035_2509_071A
/2509ch06.08.25.html

ericscheie
68 comments
Joined 11/08

19 Aug 2015

@alpanchino - What is the version of your REST service? Is it a beta version? That particular error is a case issue that was corrected before the GCA of rest.

For the ODBC error, I would check the name of the system you are passing python and make sure that is resolving to a pingable IP address.

ericscheie
68 comments
Joined 11/08

19 Aug 2015

@sds - Yes, this definitely possible. The trick is to pass in the ReturnGeneratedKeys="C" option to the ODBC driver. This can be done directly on the call to udaExec.connect(). This option will cause the ODBC driver to return the generated keys in the result set which you access just like you would any result set. E.g.

```
1 conn = udaExec.connect(..., ReturnGeneratedKeys="C")
2 generatedId = conn.execute("INSERT INTO ...").fetchone()[0]
```

alpanchino
6 comments
Joined 08/11

20 Aug 2015

@ericschie - The REST API version -> Version 1.00.00
Regarding ODBC - yes the name is pingable by name/ip
/localhost. Moreover the tdxodbc can connect to the host...
Does the module use ODBCINI environment variable or different method to find odbc.ini file with DSNnames?

```
---------------------------------------------------------------------------
/opt/teradata/client/15.00/odbc_64/bin/tdxodbc
Enter Data Source Name: tdlocal
Enter UserID: dbc
Enter Password:
Connecting with SQLConnect(DSN=tdlocal,UID=dbc,PWD=*)...
.....ODBC connection successful.
ODBC version        = -03.52.0000-
DBMS name           = -Teradata-
DBMS version        = -15.00.0002  15.00.00.02-
Driver name         = -tdata.so-
Driver version      = -15.00.00.00-
Driver ODBC version = -03.51-
(type quit to terminate adhoc)
Enter SQL string : select current_timestamp
Executing SQLExecDirect("select current_timestamp")...
SQL Statement [1]: 1 rows affected.
Current TimeStamp(6)
2015-08-19 21:35:24.600000+00:00
============================================================================
```

alpanchino
6 comments
Joined 08/11

20 Aug 2015

What is the last version of Teradata REST (tdrestd)? I have the issue with tdrestd-15.10.00.00.Dev-1 package installed.

ericscheie
68 comments
Joined 11/08

20 Aug 2015

@alpanchino - Your REST version is indeed a pre-release version, please upgrade to the GCA version (there should not be "Dev" in the package name).

If you want to connect to a data source named "tdlocal" in an odbc.ini file, then replace the system="xxxx" parameter in your udaExec call with DSN="tdlocal". The system parameter is for the tdpid, not the data source name.

alpanchino
6 comments
Joined 08/11

20 Aug 2015

Thanks, Eric. Now it is working with if system parameter is tdpid. Just for checking I have tried DSN=tdlocal - it seems the module ignores the parameter :
....
session = udaExec.connect(method="odbc", DSN="tdlocal", username="dbc", password="dbc");
.....

Traceback (most recent call last):

File "TDCH_Odbc_Archive2.py", line 6, in <module>

session = udaExec.connect(method="odbc", DSN="tdlocal", username="dbc", password="dbc");

File "/usr/local/lib/python2.7/site-packages/teradata/udaexec.py", line 137, in connect

raise e

teradata.api.DatabaseError: (0, u'[08001] [Teradata][ODBC Teradata Driver] No DBCName entries were found in DSN/connection-string. ')

cr255014
5 comments
Joined 01/14

21 Aug 2015

@alpanchino: I believe issue is with your odbc.ini file. Can you check if DBCName exists for DSN "tdlocal" in your ODBCINI file?

ericscheie
68 comments
Joined 11/08

22 Aug 2015

@alpanchino - There was a bug that was preventing the ODBC driver from using the specified DSN. A fixed has been committed and an update released (15.10.00.06). Thanks for reporting.