

Teradata® Package for Python User Guide - 17.20

Release 17.20




November 2022

Copyright and Trademarks

Copyright © 2018 - 2023 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

| Safety type | Description |
|--|--|
|  NOTICE | Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury. |
|  CAUTION | Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury. |
|  WARNING | Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury. |

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable laws, all designs, specifications, statements, information, recommendations and content (collectively, "content") available from the Teradata Documentation website or contained in Teradata information products is presented "as is" and without any express or implied warranties, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or noninfringement, which are hereby disclaimed. In no event shall Teradata corporation, its suppliers or partners be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of content, even if advised of the possibility of such damage.

The Content available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The Content available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in the Content at any time without notice.

The Content is subject to change without notice. Users are solely responsible for their application of the Content. The Content does not constitute the technical or other professional advice of Teradata, its suppliers or partners. Users should consult their own technical advisors before implementing any Content. Results may vary depending on factors not tested by Teradata.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Confidential Information

Confidential Information means any and all confidential knowledge, data or information of Teradata, including, but not limited to, copyrights, patent rights, trade secret rights, trademark rights and all other intellectual property rights of any sort.

The Content available from the Teradata Documentation website or contained in Teradata information products may include Confidential Information and as such, the use of such Content is subject to the non-use and confidentiality obligations and protections of a non-disclosure agreement or other such agreements to protect Confidential Information that you have executed with Teradata.

Contents

| | |
|---|------------|
| Chapter 1: Introduction to Teradata Package for Python | 7 |
| Welcome to Teradata Package for Python | 7 |
| Teradata Package for Python Product Overview | 8 |
| The teradataml Package | 8 |
| Key Feature Additions and Changes | 9 |
| Chapter 2: Installing, Uninstalling, and Upgrading Teradata Package for Python | 12 |
| Installing Teradata Package for Python | 12 |
| Uninstalling Teradata Package for Python | 18 |
| Upgrading Teradata Package for Python | 18 |
| Chapter 3: teradataml Components | 20 |
| Helper Functions | 20 |
| Garbage Collection in teradataml | 22 |
| Execute SQL Queries using teradataml | 23 |
| Chapter 4: DataFrames for Tables and Views | 28 |
| DataFrames from Teradata Vantage Data Sources | 28 |
| Access Tables in a Non-Default Database in Vantage | 32 |
| Input Classes for UAF Functions | 33 |
| DataFrame Manipulation | 41 |
| DataFrame Metadata | 143 |
| Data Rotation | 149 |
| Saving teradataml DataFrames to Vantage | 163 |
| Exporting DataFrame to External Entities | 169 |
| Chapter 5: teradataml DataFrame Column | 178 |
| Access teradataml DataFrame Column | 178 |
| teradataml DataFrame Column Manipulation | 180 |
| Chapter 6: teradataml Window Aggregates | 201 |
| Window on DataFrame | 201 |
| Window on DataFrame Column | 204 |
| Window Aggregate Functions | 206 |
| Chapter 7: Context to Teradata Vantage | 210 |
| create_context | 210 |
| get_context | 215 |
| get_connection | 216 |

| | |
|---|------------|
| set_context | 216 |
| remove_context | 217 |
| Chapter 8: teradataml General Functions | 219 |
| Data Transfer Utility | 219 |
| Database Utility | 251 |
| File Management Functions | 257 |
| Miscellaneous Functions | 260 |
| Apply SET Operations on teradataml DataFrames | 263 |
| Table Operators | 278 |
| Chapter 9: teradataml Options | 312 |
| Configuration Options | 312 |
| Display Options | 319 |
| set_config_params | 324 |
| Chapter 10: Series | 326 |
| Series from teradataml DataFrame | 326 |
| Series Manipulation | 326 |
| Series Metadata | 328 |
| Chapter 11: BYOM | 330 |
| Supported External Model Types | 330 |
| save_byom() | 336 |
| list_byom() | 342 |
| retrieve_byom() | 345 |
| delete_byom() | 353 |
| set_byom_catalog() | 355 |
| set_license() | 356 |
| get_license() | 359 |
| BYOM Workflows | 360 |
| Chapter 12: Working with Geospatial Data | 361 |
| teradataml Geometry Types | 361 |
| teradataml GeoDataFrame | 375 |
| teradataml GeoDataFrameColumn | 391 |
| Chapter 13: Using Teradata Vantage Analytic Functions with Teradata Package for Python | 407 |
| Usage Notes | 408 |
| load_example_data() Function | 411 |
| Use Cases | 413 |
| Chapter 14: Teradata Package for Python Function Reference | 452 |
| Appendix A: Teradata Package for Python Limitations and Considerations | 453 |

| | |
|--|------------|
| Appendix B: Using teradataml with Native Object Store | 476 |
| Appendix C: teradataml Extension with SQLAlchemy | 492 |
| Appendix D: Data Type Mapping | 511 |
| Appendix E: Model Cataloging | 513 |
| Appendix F: ML Engine Specific Settings and Examples | 525 |
| Appendix G: Additional Information | 556 |

Introduction to Teradata Package for Python

Welcome to Teradata Package for Python

Using the Teradata Package for Python (teradataml)

The Teradata Package for Python (teradataml) is an open-source Python library package that combines the benefits of open-source Python language environment with the massive parallel processing capabilities of Teradata Vantage.

The purpose of this document is to help data scientists:

- Describe available features and functionalities of teradataml.
- Install, uninstall, and upgrade the teradataml package.
- Connect to Vantage.
- Use teradataml for data management, exploration, and execution of analytic functions.
- Understand limitations and considerations of teradataml.

Why Would I Use this Content?

The teradataml package is a Python library package like other open-source Python packages. The package interface makes available to Python users a collection of functions for analytics that reside on Vantage, so that Python users can perform in-database analytics with no SQL coding required. Also, the teradataml package provides functions for data manipulation and transformation, data filtering and sub-setting, and can be used in conjunction with open-source Python capabilities.

How Do I Use this Content?

Use this guide as a reference to find descriptions, usage notes, and examples of features and functions available in the teradataml package.

How Do I Get Started?

In your Python environment:

1. [Install the teradataml package.](#)
2. [Establish connection to Vantage.](#)

Then you can load data, manipulate and transform data for analysis, and execute analytic functions.

References to Other Relevant Content

On the Teradata documentation web site <https://docs.teradata.com/>:

- *Teradata Package for Python Function Reference*, B700-4008
- *Teradata Vantage™ Machine Learning Engine Analytic Function Reference*, B700-4003

- *Teradata Vantage™ - Analytics Database Analytic Functions*, B035-1206
- *Teradata Package for R User Guide*, B700-4005
- *Teradata Package for R Function Reference*, B700-4007
- *Teradata Vantage™ Modules for Jupyter Installation Guide*, B700-4010
- For VantageCloud Lake only: Build Scalable Python Analytics with Open Analytics Framework

On the Python Package Index (PyPI) site <https://pypi.org/user/teradata/>:

- Teradata SQL Driver for Python (teradatasql)
- Teradata SQL Driver Dialect for SQLAlchemy (teradatasqlalchemy)

Teradata Package for Python Product Overview

The Teradata Package for Python product combines the benefits of the open source Python language environment with the massive parallel processing capabilities of Teradata Vantage, which includes the Machine Learning Engine analytic functions and the Analytics Database in-database analytic functions. The Teradata Package for Python allows users to develop and run Python programs that take advantage of the Big Data and Machine Learning analytics capabilities of Vantage.

The Teradata Package for Python is `teradataml`, a Python library package like other open source Python packages. The package interface makes available to Python users a collection of functions for analytics that reside on Vantage, so that Python users can perform analytics with no SQL coding required. Specifically, the `teradataml` package provides functions for data manipulation and transformation, data filtering and sub-setting, and can be used in conjunction with open source Python libraries. The `teradataml` package uses SQLAlchemy and provides an interface similar to the Pandas Python library.

The Teradata Package for Python works over connections to:

- Vantage with Analytics Database and ML Engine
- VantageCloud Lake, VantageCloud Enterprise and VantageCore with Analytics Database only

Note:

For this type of connection, only Analytics Database analytic functions are accessible.

Teradata Vantage Modules for Jupyter

Teradata Package for Python is included in the Docker image of Teradata Vantage Modules for Jupyter, which also includes JupyterLab and other components to run as a Docker container on a client machine.

Teradata Vantage Modules for Jupyter allows users to access Vantage in Python, R or SQL from JupyterLab notebooks.

See *Teradata Vantage™ Modules for Jupyter Installation Guide*, B700-4010.

The teradataml Package

The `teradataml` package runs on the client system and is designed for data management, exploration, and execution of analytic functions.

The current version of the `teradataml` package includes over 100 functions, organized into these functional areas:

- Utility and database management functions
- Data exploration and preparation functions
- Analytic functions across Vantage

These functions support high-speed analytics processing required to operationalize analytics and automate data partitioning and parallel processing in Vantage.

teradataml, SQLAlchemy and Pandas

For Python users familiar with the Pandas Python package, the `teradataml` package builds on the concept and syntax of the pandas DataFrame object by creating the `teradataml DataFrame` object for data residing on Vantage.

The look and feel of a `teradataml DataFrame` is similar to a pandas DataFrame in Python. A `teradataml DataFrame` is a reference to a database object on the Python client, and it can represent a table, view, or query in Analytics Database.

The `teradataml` library provides an API to access or manipulate a `teradataml DataFrame`. These functions generate an SQL request to be executed in the Analytics Database or ML Engine through a Python DB-API connection. The `teradataml` package uses `teradatasqlalchemy`, an implementation of SQLAlchemy's Dialect interface, to provide enhanced support for rendering SQL. The `teradatasqlalchemy` dialect leverages the `teradatasql` DB-API implementation for connecting to Vantage.

The `teradataml DataFrame` supports lazy evaluation for a variety of operations on Vantage such as exploration, transformation, and machine learning. Only a subset of data is ever retrieved from Vantage, unless the user explicitly requests data to be transferred to the client.

These characteristics result in a pandas-like experience for users who want to perform analytics on Vantage from their preferred Python environment.

Key Feature Additions and Changes

The following table lists the key feature additions and changes in the Teradata Package for Python, `teradataml`.

| Date | Release | Description |
|----------|-------------|--|
| May 2023 | 17.20.00.03 | <ul style="list-style-type: none"> • Removed the constraint that analytic functions must be imported after creating context. Analytic functions now can be imported regardless of context creation. • ReadNOS and WriteNOS now accept dictionary value for <i>row_format</i> and <i>authorization</i> arguments. |

| Date | Release | Description |
|---------------|-------------|--|
| | | <ul style="list-style-type: none"> WriteNOS supports writing CSV files to external store. Changes to the DataFrame.join: <ul style="list-style-type: none"> Added new arguments <i>lprefix</i> and <i>rprefix</i>. Teradata recommends using these new arguments, instead of old arguments <i>lsuffix</i> and <i>rsuffix</i>. See join() Method for more details. New and old affix arguments can now be used independently. |
| March 2023 | 17.20.00.02 | <ul style="list-style-type: none"> New function <code>set_auth_token()</code> to set the JWT token automatically for using Open Analytics Framework APIs. New display option <code>display.suppress_vantage_runtime_warnings</code> to suppress the VantageRuntimeWarning raised by <code>teradataml</code>, when set to True. Updates to the following existing functions: <ul style="list-style-type: none"> <code>SimpleImputeFit</code> function arguments <code>stats_columns</code> and <code>stats</code> are made to be optional. <code>ReadNOS</code> function has new argument <code>table_format</code>, and argument <code>full_scan</code> is changed to <code>scan_pct</code>. Added support of hash by and local order by to <code>APPLY</code> and <code>DataFrame.apply()</code>. |
| January 2023 | 17.20.00.01 | <ul style="list-style-type: none"> New Unbounded Array Framework (UAF) Time Series functions; New input classes (<code>TDSeries</code>, <code>TDGenSeries</code>, <code>TDMatrix</code>, <code>TDAnalyticResult</code>) for UAF functions; New DataFrame functions (<code>pivot()</code>, <code>unpivot()</code>) to rotate data to create easy-to-read DataFrames; New DataFrame function <code>drop_duplicate()</code> to drop duplicate rows; New DataFrame property <code>is_art</code> to check whether DataFrame is created on an Analytic Result Table (ART) or not. Updates to the following existing functions: <ul style="list-style-type: none"> Native Object Store (NOS) functions <code>display_analytic_functions()</code> function <code>ColumnTransformer</code> function Analytics Database functions, UAF functions, NOS functions and BYOM functions are available only if underlying Vantage, which <code>teradataml</code> is connected to, supports the functions. And the functions should be imported only after the connection to Vantage is established. See Usage Notes when using Vantage Analytic Functions with teradataml, BYOM functions usage notes and NOS functions usage notes for more details. |
| November 2022 | 17.20.00.00 | <ul style="list-style-type: none"> Support for use with VantageCloud Lake edition. Support Analytics Database 17.20 Analytic Functions. New Open Analytics feature APIs (user environment management functions, <code>UserEnv</code> Class functions, <code>Apply</code> Class functions) for use exclusively with Open Analytics Framework on VantageCloud Lake. New <code>DataFrame.apply</code> method executes a user defined Python function on VantageCloud Lake. |

| Date | Release | Description |
|------|---------|---|
| | | <ul style="list-style-type: none">• New options (<code>auth_token</code>, <code>base_url</code>, <code>certificate_file</code>) for use with Open Analytics Framework on VantageCloud Lake.• New configuration option <code>set_config_params</code> set all config parameters in one go.• New BYOM function <code>ONNXPredict()</code> to score ONNX formatted models.• New database utility function <code>list_td_reserved_keywords()</code> to list Teradata reserved keyword.• New feature that special characters used in the password are encoded by default when using <code>create_context</code>. New optional argument <code>url_encode</code> can be set to 'False' to manually handle space and unreserved characters. |

Installing, Uninstalling, and Upgrading Teradata Package for Python

Installing Teradata Package for Python

Teradata Package for Python System Requirements

Required Software on Teradata Vantage

Based on the connection to Vantage, the `teradataml` package requires the following minimum software versions be installed on Vantage:

Teradata Vantage with Analytics Database Release 16.20.16.01 or later.

Required Software on Client

The `teradataml` package also requires a minimum version of the following software be installed on the client:

- `terdatasqlalchemy` 17.00.00.04
- `terdatasql` 17.10.00.11
- `sqlalchemy` 1.4.46

Note:

`teradataml` 17.20.00.xx versions only support SQLAlchemy 1.4, SQLAlchemy 2.0 is not yet supported.

- `pandas` 0.22.00
- `psutil`
- Operating Systems:
 - Windows OS: Windows 7
 - macOS: os x 10.9
 - Linux: Ubuntu 16.04, CentOS 7.0, RHEL 7.1, or SLES 12

Note:

The `teradataml` package only supports 64-bit version of these operating systems.

Installing Teradata Package for Python on a Windows OS Client

Installing Open Source Python on Windows

Python must be installed on the client system before you install the teradataml package.

The minimum version supported is Python 3.5.0. If you already have Python 3.5.0 or later version installed, skip this section.

1. Obtain the latest binary Python package through the **Download** link on the Python Software Foundation home page or directly from <https://www.python.org/ftp/python/>.
2. Double click the downloaded installer, for example, python-3.5.0-amd64.exe, and follow the instructions to install the package.
3. Update the **PATH** system variable in the **Environment Variables** of the **System Properties** on Windows OS.
 - a. Add the installed location of Python to **PATH**.
For example, expand the existing **PATH** with the following if you install Python 3.5.0 in the directory C:\Python35\:

```
C:\Python35\
```

- b. Add the location of pip program to **PATH**.
For example, expand the existing **PATH** with the following if you install Python 3.5.0 in the directory C:\Python35\:

```
C:\Python35\Scripts\
```

Installing Teradata Package for Python on Windows

1. Upgrade setuptools using the command:

```
py -3 -m pip install --upgrade setuptools
```

2. Install teradataml package along with its dependencies using the command:

```
py -3 -m pip install teradataml
```

Note:

teradataml has a dependency on terdatasqlalchemy and terdatasql.

Executing this command will also download and install sqlalchemy, terdatasqlalchemy, and terdatasql, if they are not yet installed.

terdatasqlalchemy, and terdatasql can also be separately installed from <https://pypi.org/user/teradata/>.

3. Verify the installation by [Testing Connection to Vantage](#).

Installing Teradata Package for Python on a macOS Client

Installing Open Source Python on macOS

Python must be installed on the client system before you install the teradataml package.

The minimum version supported is Python 3.5.0. If you already have Python 3.5.0 or later version installed, skip this section.

1. Obtain the latest binary Python package through the **Download** link on the Python Software Foundation home page or directly from <https://www.python.org/ftp/python/>.
2. Double click the downloaded installer, for example, python-3.5.0-macosx10.6.pkg, and follow the instructions to install the package.
3. Add the installed locations of Python and pip to the **PATH** environment variable. For example, if you install Python 3.5 in the default location, add the following:

```
/Library/Frameworks/Python.framework/Versions/3.5/bin
```

Installing Teradata Package for Python on macOS or Linux OS

1. Upgrade setuptools using the command:

```
pip install --upgrade setuptools
```

2. Install teradataml package along with its dependencies using the command:

```
pip install teradataml
```

Note:

teradataml has a dependency on teradatasqlalchemy and teradatasql.

Executing this command will also download and install sqlalchemy, teradatasqlalchemy, and teradatasql, if they are not yet installed.

teradatasqlalchemy, and teradatasql can also be separately installed from <https://pypi.org/user/teradata/>.

3. Verify the installation by [Testing Connection to Vantage](#).

Installing Teradata Package for Python on a Linux OS Client

Installing Open Source Python on Linux OS

Python must be installed on the client system before you can install teradataml package.

The minimum versions supported are:

- Ubuntu 16.04
- CentOS 7.0
- RHEL 7.1
- SLES 12

Refer to the following sections for steps to install dependencies on chosen platform, and then install Python.

Installing Open Source Python on Ubuntu

1. Install the build-essential package.

```
sudo apt install build-essential checkinstall
```

2. Install dependencies.

```
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev  
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev openssl
```

3. Get Python 3.5.0 source code, and configure and install Python.

- a. Make \$HOME/opt directory and change to this directory.

```
mkdir -p $HOME/opt  
cd $HOME/opt
```

- b. Get the Python 3.5.0 source code.

```
curl -O https://www.python.org/ftp/python/3.5.0/Python-3.5.0.tgz
```

- c. Unzip the downloaded Python package.

```
tar xzvf Python-3.5.0.tgz
```

- d. Change to the directory where the package is unzipped.

```
cd Python-3.5.0
```

- e. Configure the settings.

```
./configure --enable-shared --prefix=/usr/local LDFLAGS="-Wl,-  
rpath=/usr/local/lib"
```

- f. Install the Python.

```
sudo make altinstall
```

Installing Open Source Python on CentOS/RHEL

1. Install Development Tools.

```
sudo yum groupinstall 'Development Tools'
```

2. Get Python 3.5.0 source code, and configure and install Python.

- a. Make \$HOME/opt directory and change to this directory.

```
mkdir -p $HOME/opt  
cd $HOME/opt
```

- b. Get the Python 3.5.0 source code.

```
curl -O https://www.python.org/ftp/python/3.5.0/Python-3.5.0.tgz
```

- c. Unzip the downloaded Python package.

```
tar xzvf Python-3.5.0.tgz
```

- d. Change to the directory where the package is unzipped.

```
cd Python-3.5.0
```

- e. Configure the settings.

```
./configure --enable-shared --prefix=/usr/local --exec-  
prefix=/usr/local"
```


- f. Install the Python.

```
sudo make altinstall
```

Installing Open Source Python on SLES 12

1. Install GCC compiler and development environment.

```
sudo zypper in gcc-c++ make ncurses glibc-devel
```

2. Get Python 3.5.0 source code, and configure and install Python.

- a. Make \$HOME/opt directory and change to this directory.

```
mkdir -p $HOME/opt
cd $HOME/opt
```

- b. Get the Python 3.5.0 source code.

```
curl -O https://www.python.org/ftp/python/3.5.0/Python-3.5.0.tgz
```

- c. Unzip the downloaded Python package.

```
tar xzvf Python-3.5.0.tgz
```

- d. Change to the directory where the package is unzipped.

```
cd Python-3.5.0
```

- e. Configure the settings.

```
./configure --enable-shared --prefix=/usr/local --exec-prefix=/usr/local"
```

- f. Install the Python.

```
sudo make altinstall
```

Installing Teradata Package for Python on macOS or Linux OS

1. Upgrade setuptools using the command:

```
pip install --upgrade setuptools
```

2. Install teradataml package along with its dependencies using the command:

```
pip install teradataml
```

Note:

teradataml has a dependency on teradata sqlalchemy and teradata sql.

Executing this command will also download and install sqlalchemy, teradata sqlalchemy, and teradata sql, if they are not yet installed.

teradata sqlalchemy, and teradata sql can also be separately installed from <https://pypi.org/user/teradata/>.

3. Verify the installation by [Testing Connection to Vantage](#).

Testing Connection to Vantage

After installing the teradataml package on the client, you can import the teradataml module, and establish a connection to Vantage from the Python interpreter on the client.

1. Open Python interpreter.
2. Import the teradataml module:

```
from teradataml import *
```

3. Create the connection context using the create_context command:

```
<connection_name> = create_context(host = "<tdhost_fqdn>",  
username="<username>", password = "<password>")
```

4. Run the following command to check the connection context:

```
print (<connection_name>)
```

Uninstalling Teradata Package for Python

Uninstalling the teradataml package on Windows

Use the following command to uninstall the teradataml package:

```
py -3 -m pip uninstall -y teradataml
```

Uninstalling the teradataml package on macOS or Linux OS

Use the following command to uninstall the teradataml package:

```
pip uninstall -y teradataml
```

Upgrading Teradata Package for Python

Upgrading the teradataml package on Windows

To upgrade the teradataml package, you can use the following command:

```
py -3 -m pip install --no-cache-dir -U teradataml
```

Upgrading the teradataml package on macOS or Linux OS

Use the following command to upgrade the teradataml package:

```
pip install --no-cache-dir -U teradataml
```

teradataml Components

Helper Functions

load_example_data() Function

The `load_example_data()` is a helper function that loads the sample datasets.

Teradata Package for Python offers various APIs and each API provides some examples. To test these examples, users need the sample datasets loaded in Vantage.

The `load_example_data()` function can only be used in a restricted way. Function arguments only accept predetermined values as shown in the given examples in this User Guide and the [Teradata Package for Python Function Reference](#).

- *function_name*

This required argument contains the prefix name of the example JSON file to be used to load data.

Note:

You must specify the *function_name* values as specified in the example sections of corresponding teradataml APIs. If any other string is passed as prefix input, an error will be raised as 'prefix_str_example.json' file not found. This *_example.json file contains the schema information for the tables that can be loaded using this JSON file.

-
- *table_name*

This required argument specifies the name(s) of the table to be created in the database.

Note:

Table names provided here must have an equivalent datafile (CSV) present at teradataml/data. Schema information for the same must also be present in <function_name>_example.json as shown in 'function_name' argument description.

Note:

- The function creates a new table in Vantage with the name specified in the *table_name* argument. You must manually drop the table if required.
- If a table with the name provided for *table_name* argument already exists, this function skips creation and loading of the dataset.
- Database used for loading table depends on the following conditions:
 - If the configuration option **temp_table_database** is set, then the tables are loaded in the database specified in this option.
 - If the configuration option **temp_table_database** is not set and the *temp_database_name* argument is used while creating context, then the tables are loaded in the database specified in the *temp_database_name* argument.
 - If none of them are specified, then the tables are created in the connecting users' default database or the connecting database.

Example 1: When connection is created without *temp_database_name*, table is loaded in users' default database

```
>>> from teradataml import *

>>> con = create_context(host = 'tdhost', username='tduser', password
= 'tdpassword')

>>> load_example_data("pack", "ville_temperature")

# Create a teradataml DataFrame.
>>> df = DataFrame("ville_temperature")
```

Example 2: When connection is created with *temp_database_name*, table is loaded in that database

This examples shows when connection is created with the *temp_database_name* argument, then tables are loaded in the database specified in the *temp_database_name* argument.

This example also demonstrates loading multiple tables in a single function call.

```
>>> con = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', temp_database_name = "temp_db")

>>> load_example_data("GLM", ["admissions_train", "housing_train"])
```

```
# Create teradataml DataFrames.
>>> admission_train = DataFrame(in_schema("temp_db", "admissions_train"))
>>> housing_train = DataFrame(in_schema("temp_db", "housing_train"))
```

Example 3: When connection is created with *temp_database_name* and the configuration option *temp_table_database* is set

This example shows when connection is created with *temp_database_name* and the configuration option *temp_table_database* is also specified, the table is created in the database specified in the configuration option *temp_table_database*.

```
>>> from teradataml import *

>>> con = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', temp_database_name = "temp_db")

>>> configure.temp_table_database = "temp_db1"

>>> load_example_data("pack", "ville_temperature")

# Create a teradataml DataFrame.
>>> df = DataFrame(in_schema("temp_db1", "ville_temperature"))
```

Garbage Collection in teradataml

teradataml offers various DataFrame API's and analytic functions that enables users to process the data and run analytics on the data. While doing so, teradataml internally creates various database objects (Views and Tables) on Vantage, whenever and wherever necessary. All these internally generated views and tables are temporarily created and have a session wide scope. At the end of the session, these views and tables will be removed by teradataml. teradataml GarbageCollector takes care of cleaning up these internally created views and tables.

When a Garbage Collection is performed, cleanup is performed for:

- Current Session:
When a user ends a session, Garbage Collection of current active session is done. All temporarily created objects within that session are removed.
- Old stale or inactive Python sessions:
GarbageCollector also performs cleanup for other terminated Python sessions for which cleanup was not done properly due to abrupt interruption in the session.

Garbage Collection is triggered automatically upon execution of the following function calls.

1. [remove_context](#)

This API must be called when user wants to end the current session.

Calling this API triggers the Garbage Collection to cleanup objects from current session as well as old inactive sessions.

2. [create_context](#)

This API must be called when either user wants to create a new connection or wants to overwrite the existing context.

Calling this API also triggers Garbage Collection. When a new context is created, Garbage Collection is performed for the objects created by teradataml in an old, abruptly terminated Python sessions. If a context is being recreated, then Garbage Collection is performed for objects created in previous context and objects created by old abruptly terminated Python sessions.

3. [set_context](#)

When this API is used, old context is overwritten.

Calling this API triggers Garbage Collection for current context being overwritten and old inactive sessions.

4. At Session Exit

At the end of the session, if you issue `quit()`, then Garbage Collection is performed.

Note:

- Teradata recommends calling [remove_context](#) to end a teradataml session so that cleanup is done properly.
- GarbageCollector uses the current context to perform Garbage Collection of intermediate views and tables.

If the connecting user does not have permission to drop the views and tables from old inactive sessions, then those sessions are not cleaned up.

- GarbageCollector does not remove the objects created by teradataml from different client. It only attempts to remove the objects created by the same client.
 - While using Jupyter notebooks, if session is abruptly terminated, even though session is inactive, cleanup is not done until the kernel responsible for the Python session is shutdown. Same case is applicable for directly closing the notebook. You should close the notebook only after calling [remove_context](#) at the end.
-

Execute SQL Queries using teradataml

teradataml offers a way to execute SQL queries on Vantage from Python interface. User can use the `execute()` method of the connection object returned by `get_connection()` function to execute the queries.

Example

This example shows how to use `execute()` method to create a table, insert rows into table, select rows of the table and alter rows of the table.

1. Create a connection and get the connection parameters.

- Create the connection and get the engine.

```
>>> eng = create_context("<td_host>", "<td_user>", "<td_pwd>")
```

- Print the engine.

```
>>> eng
Engine(teradatasql://<td_user>:***@<td_host>)
```

- Get the connection object.

```
>>> conn = get_connection()
```

- Print the connection object.

```
>>> conn
<sqlalchemy.engine.base.Connection at 0x7fb598edb610>
```

2. Make sure the table to be created does not exist in the database.

Check if the table exists or not.

```
>>> eng.dialect.has_table(connection=conn, table_name="employee_info")
False
```

Note:

If the table is already present, drop the table as follows:

```
>>> db_drop_table("employee_info")
True
```

This is not needed when the table is not present.

3. Create table 'employee_info' using `execute()`.

- Define a "create table" statement.

```
>>> create_stmt = "create table employee_info (employee_no BIGINT,
first_name VARCHAR(50), last_name VARCHAR(50), date_of_joining DATE FORMAT
'YYYY-MM-DD', DOB DATE FORMAT 'YYYY-MM-DD');"
```

- Execute the "create table" statement.


```
>>> output = conn.execute(create_stmt)
```

- Print the output of execution.

```
>>> for row in output:
...     print(row)
```

Note:

Generally, the output of the execute() function returns the result set. For this example, as it is a create table statement, empty result set is generated.

- Check if the table exists or not.

```
>>> eng.dialect.has_table(connection=conn, table_name="employee_info")
True
```

- Create teradataml DataFrame.

```
>>> df = DataFrame("employee_info")
```

- Check the columns of the DataFrame.

```
>>> df.columns
['employee_no', 'first_name', 'last_name', 'date_of_joining', 'DOB']
```

- Check the shape of the DataFrame.

```
>>> df.shape
(0, 5)
```

4. Insert rows using execute().

- Define "insert" statements.

```
>>> ins_stmt = "insert into employee_info values (28, 'my_first_name',
'my_last_name', '2018-06-11', '1987-11-25');"

```

```
>>> ins_stmt1 = "insert into employee_info values (4, 'my_first_name_1',
'my_last_name_1', '2018-06-11', '1988-02-23');"

```

- Execute the "insert" statements.

```
>>> ins = conn.execute(ins_stmt)
```

```
>>> ins1 = conn.execute(ins_stmt1)
```

Note:

Both "ins" and "ins1" have empty result set.

- Check the shape of the DataFrame.

```
>>> df.shape
(2, 5)
```

- Print the teradataml DataFrame.

```
>>> df
   employee_no  first_name  last_name  date_of_joining  DOB
4             4  my_first_name_1  my_last_name_1      2018-06-11  1988-02-23
28            28    my_first_name    my_last_name      2018-06-11  1987-11-25
```

5. Execute SELECT query using execute().

- Execute SELECT query using execute().

```
>>> sel_result = conn.execute("select * from employee_info;")
```

- Print the entire result set.

```
>>> for row in sel_result:
...     print(row)
(4, 'my_first_name_1', 'my_last_name_1', datetime.date(2018, 6, 11),
datetime.date(1988, 2, 23))
(28, 'my_first_name', 'my_last_name', datetime.date(2018, 6, 11),
datetime.date(1987, 11, 25))
```

- Fetch and print only certain columns.

```
>>> sel_result = conn.execute("select * from employee_info;")

>>> for row in sel_result:
...     print(str(row['employee_no']) + " : " + row['first_name'])
4 : my_first_name_1
28 : my_first_name
```

6. Alter data using execute().

- Define an alter query statement.

```
>>> alter_stmt = "update employee_info set first_name = 'my_first_name_2'
where employee_no = 4;"
```

- Execute the alter statement.

```
>>> alter = conn.execute(alter_stmt)
```

This produces empty result set.

- Print the teradataml DataFrame.

```
>>> df
```

| | first_name | last_name | date_of_joining | DOB |
|-------------|-----------------|----------------|-----------------|------------|
| employee_no | | | | |
| 28 | my_first_name | my_last_name | 2018-06-11 | 1987-11-25 |
| 4 | my_first_name_2 | my_last_name_1 | 2018-06-11 | 1988-02-23 |

The first_name for the employee with employee_id 4 is changed.

DataFrames for Tables and Views

This section provides detailed usage examples of the functions for data management, exploration, preparation in `teradataml`.

`teradataml` DataFrame examples documented in this guide require certain datasets to be loaded. Use these commands to load them:

```
>>> from teradataml import load_example_data
>>> load_example_data("dataframe", ["scale_housing_test", "employee_info",
"sales", "admissions_train", "join_table1", "join_table2", "iris_test"])
```

See [load_example_data\(\) Function](#) for more details about 'load_example_data()' utility.

In this section, assume you have a context created with these commands:

```
>>> from teradataml import create_context, DataFrame

>>> create_context(host = "myhostname", username="myusername", password
= "mypassword")
```

DataFrames from Teradata Vantage Data Sources

You can construct a `teradataml` DataFrame from either an existing Vantage table or view or a SQL query result. The data source determines the DataFrame constructor function you use.

| Vantage Data Source | DataFrame Constructor Function |
|---------------------|--|
| Table or view | DataFrame Constructor DataFrame.from_table() Function |
| SQL query result | DataFrame.from_query() Function |

DataFrame Constructor

Use `DataFrame()` function to create a `teradataml` DataFrame from an existing table or view in Vantage.

Arguments:

The function takes a table name or view name as argument and creates a DataFrame based on the table or view in Vantage.

The function also takes an index label as an optional argument. The index label is used for sorting. The value of the index label can be a column name or a list of column names.

- If the index label is not specified for a table, the primary index of the base table is used as the index label.
- If the index label is not specified for a view, the index label is set to None.

Example 1: Create a DataFrame from an existing table "sales" in Vantage

The index label is not specified, the primary index "accounts" is used.

```
>>> df = DataFrame("sales")
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |

```
>>>
```

Example 2: Create a DataFrame from an existing table "sales" in Vantage with an index label "Feb"

```
>>> df = DataFrame("sales", index_label="Feb")
>>> df
```

| | accounts | Jan | Mar | Apr | datetime |
|-------|------------|------|------|------|------------|
| Feb | | | | | |
| 210.0 | Alpha Co | 200 | 215 | 250 | 04/01/2017 |
| 200.0 | Red Inc | 150 | 140 | None | 04/01/2017 |
| 210.0 | Orange Inc | None | None | 250 | 04/01/2017 |
| 200.0 | Jones LLC | 150 | 140 | 180 | 04/01/2017 |
| 90.0 | Yellow Inc | None | None | None | 04/01/2017 |
| 90.0 | Blue Inc | 50 | 95 | 101 | 04/01/2017 |

Example 3: Create a DataFrame from an existing table "sales" in Vantage with an index label "Jan" and "Feb"

```
>>> df = DataFrame("sales", index_label=["Jan", "Feb"])
>>> df
```

| | | accounts | Mar | Apr | datetime |
|-----|-------|------------|------|------|------------|
| Jan | Feb | | | | |
| 200 | 210.0 | Alpha Co | 215 | 250 | 04/01/2017 |
| 150 | 200.0 | Red Inc | 140 | None | 04/01/2017 |
| NaN | 210.0 | Orange Inc | None | 250 | 04/01/2017 |
| 150 | 200.0 | Jones LLC | 140 | 180 | 04/01/2017 |

```
NaN 90.0    Yellow Inc    None    None    04/01/2017
50  90.0      Blue Inc      95    101    04/01/2017
```

Example 4: Creates a DataFrame from an existing view "salesv" in Vantage with an index_label "Mar"

Note:

To use this example, one must create a view on sales table at the backend with name 'salesv'. If not created, user will not be able to run this example.

```
>>> get_context().execute("CREATE VIEW salesv AS SELECT * FROM sales")
<sqlalchemy.engine.result.ResultProxy object at 0x11bbc3668>
```

```
>>> df = DataFrame("salesv", index_label="Mar")
>>> df
```

| | accounts | Feb | Jan | Apr | datetime |
|-----|------------|-------|------|------|------------|
| Mar | | | | | |
| 95 | Blue Inc | 90.0 | 50 | 101 | 04/01/2017 |
| NaN | Orange Inc | 210.0 | None | 250 | 04/01/2017 |
| 140 | Red Inc | 200.0 | 150 | None | 04/01/2017 |
| NaN | Yellow Inc | 90.0 | None | None | 04/01/2017 |
| 140 | Jones LLC | 200.0 | 150 | 180 | 04/01/2017 |
| 215 | Alpha Co | 210.0 | 200 | 250 | 04/01/2017 |

DataFrame.from_table() Function

You can also use the DataFrame.from_table() function to create a teradataml DataFrame from an existing table or view in Vantage.

Example 1: Create a teradataml DataFrame

```
>>> df = DataFrame.from_table("sales")
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |

Example 2: Create a teradataml DataFrame with 'index_label'

```
>>> df = DataFrame.from_table("sales", index_label="Feb")
>>> df
```

| | accounts | Jan | Mar | Apr | datetime |
|-------|------------|------|------|------|------------|
| Feb | | | | | |
| 90.0 | Blue Inc | 50 | 95 | 101 | 04/01/2017 |
| 210.0 | Orange Inc | None | None | 250 | 04/01/2017 |
| 200.0 | Red Inc | 150 | 140 | None | 04/01/2017 |
| 90.0 | Yellow Inc | None | None | None | 04/01/2017 |
| 200.0 | Jones LLC | 150 | 140 | 180 | 04/01/2017 |
| 210.0 | Alpha Co | 200 | 215 | 250 | 04/01/2017 |

DataFrame.from_query() Function

The DataFrame.from_query() function constructs a teradataml DataFrame from the result of a SQL query.

Arguments:

The function takes a SQL query as an argument and creates a DataFrame based on the query.

The function also takes an index label as an optional argument. The index label is used for sorting. The value of the index label can be a column name or a list of column names. If the index label is not specified when creating a DataFrame for a query, the index label is set to None.

Example 1: DataFrame.from_query with no Index Label specified

This example creates the DataFrame "df" from the result of a SQL query of the table or view "sales".

```
>>> df = DataFrame.from_query("select Jan, Feb, datetime from sales")
>>> df
```

| | Jan | Feb | datetime |
|---|-------|-------|------------|
| 0 | NaN | 90.0 | 2017-04-01 |
| 1 | 150.0 | 200.0 | 2017-04-01 |
| 2 | NaN | 210.0 | 2017-04-01 |
| 3 | 200.0 | 210.0 | 2017-04-01 |
| 4 | 50.0 | 90.0 | 2017-04-01 |
| 5 | 150.0 | 200.0 | 2017-04-01 |

Example 2: DataFrame.from_query with One-Column Index Label

This example creates the DataFrame "df" from the result of a SQL query of the table or view "sales". The index_label is composed of one column of "sales", "Jan".

```
>>> df = DataFrame.from_query("select Jan, Feb, datetime from
sales", index_label="Jan")
```

```
>>> df
      Feb    datetime
Jan
150.0  200.0  2017-04-01
NaN     90.0  2017-04-01
NaN    210.0  2017-04-01
50.0    90.0  2017-04-01
200.0  210.0  2017-04-01
150.0  200.0  2017-04-01
```

Access Tables in a Non-Default Database in Vantage

When you create a connection to Vantage, you have the option to either specify a default database, or rely on the default database of the connecting user. However, it is also possible to interact with the tables in a database other than the default using the following approaches.

Using the `in_schema()` function

The `in_schema()` function takes a schema name and a table name and creates a database object name in the format "schema"."table_name".

Note:

Teradata recommends using this function to access tables and views from a database other than the default database.

The following example creates a DataFrame from the existing Vantage view "dbcinfo" in the non-default database "dbc" using the `in_schema()` function.

```
>>> from teradataml.dataframe.dataframe import in_schema
```

```
>>> df = DataFrame(in_schema("dbc", "dbcinfo"))
>>> df
```

| | InfoKey | InfoData |
|---|-----------------------|-------------|
| 0 | RELEASE | 16.20.27.01 |
| 1 | LANGUAGE SUPPORT MODE | Standard |
| 2 | VERSION | 16.20.27.01 |

Using fully qualified table name

Teradata recommends using `in_schema()` to create a dataframe, as shown in the previous section.

Teradata does not recommend using fully qualified table name while creating a dataframe.

Note:

If you want to continue using fully qualified table name, then the schema name and table name should be quoted, that is, "schema_name"."table_name".

You can use fully qualified table name that can be passed to the `from_table()` function or fully qualified table in SQL that can be passed to the `from_query()` function.

- Example: Using `from_table()` and fully qualified table name

```
>>> from teradataml.dataframe.dataframe import DataFrame
```

```
>>> df = DataFrame.from_table('"dbc"."dbcinfo"')
>>> df
```

| | InfoKey | InfoData |
|---|-----------------------|-------------|
| 0 | LANGUAGE SUPPORT MODE | Standard |
| 1 | RELEASE | 16.20.27.01 |
| 2 | VERSION | 16.20.27.01 |

```
>>>
```

- Example: Using `from_query()` and fully qualified table name in SQL

```
>>> from teradataml.dataframe.dataframe import in_schema
```

```
>>> df = DataFrame.from_query("SELECT * FROM dbc.dbcinfo")
>>> df
```

| | InfoKey | InfoData |
|---|-----------------------|-------------|
| 0 | RELEASE | 16.20.27.01 |
| 1 | LANGUAGE SUPPORT MODE | Standard |
| 2 | VERSION | 16.20.27.01 |

```
>>>
```

Input Classes for UAF Functions

TDSeries

Create a `TDSeries` object from a `teradataml` `DataFrame` representing a `SERIES` in time series which is used as input to Unbounded Array Framework (UAF), time series functions.

A series is a one-dimensional array. They are the basic input of UAF functions. A series is identified by its series ID, that is, the *id* argument, and indexed by its *row_index* argument.

Series is passed to and returned from UAF functions as wavelets. Wavelets are collections of rows, grouped by one or more fields, and ordered on the *row_axis* argument.

Any operations like filter, select, sum, and so on, over TDSeries returns a teradataml DataFrame.

Required Arguments:

- *data*: Specifies the teradataml DataFrame.
- *id*: Specifies the name of the column in *data* containing the identifier values.
- *row_index*: Specifies the name of the column in *data* containing the row indexing values.

Optional Arguments:

- *row_index_style*: Specifies the style of row indexing.

Permitted values are "TIMECODE" (default value), "SEQUENCE".

- *in_sequence*: Specifies a sequence of series to plot.
- *payload_field*: Specifies the names of the fields for payload.
- *payload_content*: Specifies the payload content type.

Permitted values:

- "REAL"
- "COMPLEX"
- "AMPL_PHASE"
- "AMPL_PHASE_RADIANS"
- "AMPL_PHASE_DEGREES"
- "MULTIVAR_REAL"
- "MULTIVAR_COMPLEX"
- "MULTIVAR_ANYTYPE"
- "MULTIVAR_AMPL_PHASE"
- "MULTIVAR_AMPL_PHASE_RADIANS "
- "MULTIVAR_AMPL_PHASE_DEGREES"
- *layer*: Specifies the layer name of the ART table, if dataframe is created on ART table.
- *interval*: Specifies the indicator to divide a series into a collection of intervals along its row-axis.

interval is categorized in to 4 types:

- Values represent time-duration.

Allowed values include:

- CAL_YEARS
- CAL_MONTHS
- CAL_DAYS
- WEEKS
- DAYS
- HOURS
- MINUTES
- SECONDS

- `MILLISECONDS`
- `MICROSECONDS`
- Values represent time-zero.
Allowed values include:
 - `DATE`
 - `TIMESTAMP`
 - `TIMESTAMP WITH TIME ZONE`
- Values represent an integer or floating number.
Allowed values include a positive integer or float ranging from 1 to 32767, inclusively.
- `sequence-zero`.
This is an expression which evaluates to an `INTEGER` or `FLOAT`. Used when `row_index_style` is `SEQUENCE`.

Permitted values for argument *interval* include individual values or combined values from the following:

- `time-duration`
- `time-duration, time-zero`
- `integer`
- `float, integer`
- `sequence-zero`
- `float, sequence-zero`

Example 1: Create TDSeries

```
>>> from teradataml import create_context, load_example_data,
DataFrame, TDSeries

>>> con = create_context(host = host, user=user, password=passw)

>>> load_example_data("dataframe", "admissions_train")

# Create a DataFrame to be passed as input to TDSeries.
>>> data = DataFrame("admissions_train")

# Create TDSeries object which can be used as Series_Spec input in
UAF functions.
>>> result = TDSeries(data=data,
                      id="admitted",
                      row_index="admitted",
```

```
payload_field="abc",
payload_content="REAL")
```

```
>>> result
      masters  gpa  stats programming  admitted
id
5         no  3.44  Novice      Novice         0
34        yes  3.85  Advanced  Beginner         0
13        no  4.00  Advanced    Novice         1
40        yes  3.95  Novice    Beginner         0
22        yes  3.46  Novice    Beginner         0
19        yes  1.98  Advanced  Advanced         0
36        no  3.00  Advanced    Novice         0
15        yes  4.00  Advanced  Advanced         1
7         yes  2.33  Novice    Novice         1
17        no  3.83  Advanced  Advanced         1
```

TDAnalyticResult

Create a `TDAnalyticResult` object from a teradataml DataFrame created on an Analytic Result Table (ART) which can be used as input to Unbounded Array Framework functions.

The primary use of an analytical result table (ART) is to associate function results with a name label, enabling users to easily retrieve the result data and pass the result to another UAF function.

An ART can have multiple layers. Each layer has its own dedicated row composition for the series or matrix.

Any operations like filter, select, sum, and so on, over `TDAnalyticResult` returns a teradataml DataFrame.

Required Arguments:

- *data*: Specifies the teradataml DataFrame.

Optional Arguments

- *id_sequence*: Specifies a sequence of series to plot.
- *payload_field*: Specifies the names of the fields for payload.
- *payload_content*: Specifies the payload content type.

Permitted values:

- "REAL"
- "COMPLEX"
- "AMPL_PHASE"
- "AMPL_PHASE_RADIANS"
- "AMPL_PHASE_DEGREES"
- "MULTIVAR_REAL"
- "MULTIVAR_COMPLEX"

- "MULTIVAR_ANYTYPE"
 - "MULTIVAR_AMPL_PHASE"
 - "MULTIVAR_AMPL_PHASE_RADIANS "
 - "MULTIVAR_AMPL_PHASE_DEGREES"
- *layer*: Specifies the layer name of the ART table, if dataframe is created on ART table.

Example 1: Prepare input for UAF function using an Analytic Result Table (ART)

```
>>> from teradataml import create_context
>>> con = create_context(host=host, user= user, password=password)

# Create a Analytic Result Table(ART) by executing SInfo function.
>>> from teradataml import load_example_data, SInfo
>>> load_example_data("uaf", ["ocean_buoys2"])

# Create teradataml DataFrame object.
>>> data = DataFrame.from_table("ocean_buoys2")

# Create teradataml TDSeries object.
>>> data_series_df = TDSeries(data=data,
                               id=["ocean_name","buoyid"],
                               row_index="TD_TIMECODE",
                               row_index_style="TIMECODE",
                               payload_field="jsoncol.Measure.salinity",
                               payload_content="REAL")

# Execute SInfo function and store the output in 'TSINFO_RESULTS'.
>>> uaf_out = SInfo(data=data_series_df, output_table_name='TSINFO_RESULTS')

# Create a teradataml dataframe on 'TSINFO_RESULTS' ART.
>>> art_df = DataFrame('TSINFO_RESULTS')

# Check if the DataFrame 'art_table' is created on an ART.
>>> art_df.is_art
True

# Create TDAnalyticResult object which can be used as input in UAF functions.
>>> result = TDAnalyticResult(data=art_df)

# Check if 'result' is created on an ART.
>>> result.is_art
True
```

```
>>> result
```

| | buoyid | ROW_I | INDEX_DT | | |
|----------------------------|------------------|------------------|------------------|----------------------------|-------------------------|
| INDEX_BEGIN | | | INDEX_END | NUM_ENTRIES | DISCRETE |
| SAMPLE_INTERVAL | CONTENT | MIN_MAG_salinity | MAX_MAG_salinity | AVG_MAG_salinity | |
| RMS_MAG_salinity | HAS_NULL_NAN_INF | | | | |
| 0 | 44 | 1 | TIMESTAMP(6) | 2014-01-06 10:00:24.000000 | |
| 2014-01-06 10:52:00.000009 | | | 13 | 0 | MICROSECONDS(258000001) |
| REAL | 55.0 | | 55.0 | | 55.0 |
| 55.0 | N | | | | |
| 1 | 0 | 1 | TIMESTAMP(6) | 2014-01-06 08:00:00.000000 | |
| 2014-01-06 08:10:00.000000 | | | 5 | 0 | SECONDS(150) |
| REAL | 55.0 | | 55.0 | | 55.0 |
| 55.0 | N | | | | |
| 2 | 2 | 1 | TIMESTAMP(6) | 2014-01-06 21:01:25.122200 | |
| 2014-01-06 21:03:25.122200 | | | 3 | 1 | MINUTES(1) |
| REAL | 55.0 | | 55.0 | | 55.0 |
| 55.0 | N | | | | |
| 3 | 1 | 1 | TIMESTAMP(6) | 2014-01-06 09:01:25.122200 | |
| 2014-01-06 09:03:25.122200 | | | 6 | 0 | SECONDS(24) |
| REAL | 55.0 | | 55.0 | | 55.0 |
| 55.0 | N | | | | |

TDMatrix

Create a TDMatrix object from a teradataml DataFrame representing a MATRIX in time series which is used as input to Unbounded Array Framework time series functions.

A matrix is a two-dimensional array that has rows and columns. A matrix is identified by its matrix id, that is, the *id* argument, and is indexed by its *row_index* and *column_index* arguments.

A matrix can be a one of the following types:

- Row-major matrix: Each row is a wavelet that is grouped by its matrix id and *row_index*, and ordered by its *column_index*.
- Column-major matrix: Each column is a wavelet that is grouped by its matrix id and *column_index*, and ordered by its *row_index*.

Required Arguments:

- *data*: Specifies the teradataml DataFrame.
- *id*: Specifies the name of the column in *data* containing the identifier values.
- *row_index*: Specifies the name of the column in *data* containing the row indexing values.
- *column_index*: Specifies the name of the column in *data* containing the column indexing values.

Optional Arguments:

- *row_index_style*: Specifies the style of row indexing.

Permitted values are "TIMECODE" (default value), "SEQUENCE".

- *column_index_style*: Specifies the style of column indexing.

Permitted values are "TIMECODE" (default value), "SEQUENCE".

- *id_sequence*: Specifies a sequence of series to plot.
- *payload_field*: Specifies the names of the fields for payload.
- *payload_content*: Specifies the payload content type.

Permitted values:

- "REAL"
 - "COMPLEX"
 - "AMPL_PHASE"
 - "AMPL_PHASE_RADIANS"
 - "AMPL_PHASE_DEGREES"
 - "MULTIVAR_REAL"
 - "MULTIVAR_COMPLEX"
 - "MULTIVAR_ANYTYPE"
 - "MULTIVAR_AMPL_PHASE"
 - "MULTIVAR_AMPL_PHASE_RADIANS "
 - "MULTIVAR_AMPL_PHASE_DEGREES"
- *layer*: Specifies the layer name of the ART table, if dataframe is created on ART table.

Example 1: Create a TDMatrix object

```
>>> from teradataml import create_context, load_example_data,
DataFrame, TDMatrix

>>> con = create_context(host = host, user=user, password=passw)

>>> load_example_data("dataframe", "admissions_train")

# Create a DataFrame to be passed as input to TDMatrix.
>>> data = DataFrame("admissions_train")

# Create a TDMatrix object to be passed as input to UAF functions.
>>> res = TDMatrix(data=data,
                    id='admitted',
                    row_index='id',
                    column_index = 'admitted',
                    row_index_style="TIMECODE",
                    payload_field='payload_field',
                    payload_content='REAL')
```

```
>>> res
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 40 | yes | 3.95 | Novice | Beginner | 0 |
| 7 | yes | 2.33 | Novice | Novice | 1 |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |

TDGenSeries

Generate a series to a UAF function rather than using a pre-existing series instance. It contains all the information that would have been derivable from a [TDSeries](#) as well as the information required to generate the series.

Note:

The TDGenSeries can only be passed to a function that accepts a single series as input.

Generated series have a indexing mechanism which starts at integer 0 and increments by 1 for each additional generated entry.

Required Arguments:

- *instances*: Specifies the columns and values for the generated series.
- *data_types*: Specifies the data types of the identifying columns for the generated series.
- *start*: Specifies the start value for the information about how the series payload, containing successive real magnitude values.
- *offset*: Specifies the offset value for the information about how the series payload, containing successive real magnitude values.
- *num_entries*: Specifies the number of entries for the information about how the series payload, containing successive real magnitude values.

Example 1: Create a TDGenSeries object

```
# Import TDGenSeries.
>>> from teradataml import TDGenSeries

# Import INTEGER type from teradatasqlalchemy.
>>> from teradatasqlalchemy.types import INTEGER
```



```
# Create a TDGenSeries object to be passed as input to UAF functions.
>>> series = TDGenSeries(instances = {"BuoyID": 3}, data_types = INTEGER(),
start=0, offset=1, num_entries=5)
```

DataFrame Manipulation

You can manipulate a DataFrame with methods and operators. The DataFrames created using the DataFrame() constructor, or the DataFrame() and DataFrame.from_table() and DataFrame.from_query() functions have the same methods and operators.

DataFrame Methods

A DataFrame method has the basic syntax *DataFrame.method(arguments)*. Using the specified DataFrame and arguments, the method returns a new DataFrame. The specified DataFrame remains unchanged.

| DataFrame Method | Description |
|--|---|
| assign() Method | Assigns new column expressions in a DataFrame. |
| concat() Method | Concatenate two teradataml DataFrame objects along the index axis. |
| describe() Method | Generates statistics for numeric columns. Computes the count, mean, std, min, percentiles, and max for numeric columns. |
| drop() Method | Drops specified labels from rows or columns in a DataFrame. |
| dropna() Method | Removes rows with null values in a DataFrame. |
| filter() Method | Returns only the filtered columns or rows (based on the index) of a DataFrame. Filter is item, like, or regex. Other filters are operators index[] and loc[]. |
| get() Method | Retrieves required columns from a DataFrame using column names as key. |
| get_values() Method | Retrieves all values (only) present in a DataFrame. |
| groupby() Method | Returns all columns of a DataFrame, grouped as specified. |
| head() Method | Returns the first <i>n</i> rows of a DataFrame. |
| join() Method | Joins two different teradataml DataFrames together. |
| map_row() Method | Applies a function to every row in a teradataml DataFrame and returns a teradataml DataFrame. |
| map_partition() Method | Applies a function to a group or partition of rows in a teradataml DataFrame and returns a teradataml DataFrame. |
| merge() Method | Merges two teradataml DataFrames together. |
| sample() Method | Samples rows from a DataFrame, directly or based on conditions. |
| select() Method | Returns only the selected columns of a DataFrame. |

| DataFrame Method | Description |
|-------------------------------------|--|
| set_index() Method | Assigns one or more existing columns as the new index to a DataFrame. |
| show_query() Method | Returns underlying SQL for the teradataml DataFrame. |
| sort() Method | Returns all columns of a DataFrame, sorted as specified. |
| sort_index() Method | Returns sorted objects by labels (along an axis) in either ascending or descending order for a DataFrame. |
| squeeze() Method | Squeezes one-dimensional axis objects into a scalar for DataFrames with a single element, or a Series object for a DataFrame with a single column. |
| tail() Method | Returns the last n rows of the sorted DataFrame. |

For a list of regular aggregate functions supported by DataFrame, see [Regular Aggregate Functions Supported by DataFrame](#).

For a list of time series aggregate functions, see [Time Series Aggregate Functions](#).

Note:

- The `describe()` and `get_values()` methods do not return a teradataml DataFrame.
- The `groupby()` method returns instance of teradataml DataFrameGroupBy, which is inherited from teradataml DataFrame.
- The `groupby_time()` and `resample()` methods return instance of teradataml DataFrameGroupByTime, which is inherited from teradataml DataFrame.

Operators

A DataFrame operator has the basic syntax as follows:

- `DataFrame.loc[arguments]`
- `DataFrame.iloc[arguments]`
- `DataFrame[arguments]`

All operators are filters. Another filter is the `filter()` method.

| DataFrame Operator | Description |
|----------------------------------|--|
| index[] Operator | Returns only the filtered rows of a DataFrame. Filter uses logical expressions composed of DataFrame columns and Python literals. |
| loc[] Operator | Returns new DataFrame that has only the filtered columns and rows of a DataFrame accessed by labels. |
| iloc[] Operator | Returns new DataFrame that has only the filtered columns and rows of a DataFrame accessed by integer values. |

select() Method

Use the `select()` method to select columns in a `DataFrame`. The function takes a select expression as an argument and returns a new `DataFrame` with the selected columns. The expression can be a single column name, a list of column names, or a list of column name lists.

Note:

Multicolumn selection of the same column (for example, `df.select(['col1', 'col1'])`) is not supported.

Examples Prerequisite

Assume the table "admissions_train" exists and its index column is id. And a `DataFrame` "df" is created based on this table using the command:

```
>>> df = DataFrame("admissions_train")

>>> df
  masters  gpa  stats programming admitted
id
5       no  3.44  novice      novice        0
7       yes  2.33  novice      novice        1
22      yes  3.46  novice  beginner        0
17      no  3.83  advanced  advanced        1
13      no  4.00  advanced  novice         1
19      yes  1.98  advanced  advanced        0
36      no  3.00  advanced  novice         0
15      yes  4.00  advanced  advanced        1
34      yes  3.85  advanced  beginner        0
40      yes  3.95  novice   beginner        0
```

Example 1: Expression is single column name

```
>>> df.select("id")

Empty DataFrame
Columns: []
Index: [22, 34, 13, 19, 15, 38, 26, 5, 36, 17]
```

Example 2: Expression is list of column names

```
>>> df.select(["id", "masters", "gpa"])
```

| | masters | gpa |
|----|---------|------|
| id | | |
| 5 | no | 3.44 |
| 36 | no | 3.00 |
| 15 | yes | 4.00 |
| 17 | no | 3.83 |
| 13 | no | 4.00 |
| 40 | yes | 3.95 |
| 7 | yes | 2.33 |
| 22 | yes | 3.46 |
| 34 | yes | 3.85 |
| 19 | yes | 1.98 |

Example 3: Expression is list of column name lists

```
>>> df.select(['id', 'masters', 'gpa'])
```

| | masters | gpa |
|----|---------|------|
| id | | |
| 5 | no | 3.44 |
| 34 | yes | 3.85 |
| 13 | no | 4.00 |
| 40 | yes | 3.95 |
| 22 | yes | 3.46 |
| 19 | yes | 1.98 |
| 36 | no | 3.00 |
| 15 | yes | 4.00 |
| 7 | yes | 2.33 |
| 17 | no | 3.83 |

drop and dropna

drop() Method

Use the `drop()` method to drop specified labels from rows or columns in a `DataFrame`. The method takes a single label or a list of labels as an argument and returns a new `DataFrame` with the specified rows or columns removed.

Arguments:

- *labels*

This is an optional argument. It could be single label or list-like. It also can be Index or column labels to drop depending on the *axis* argument.

- *axis*

This is an optional argument. Use the `axis` argument to specify whether the labels refer to index labels or column labels:

- 0 or 'index' for index labels
- 1 or 'column' for column labels

The default is 0.

- *columns*

This is an optional argument. Use the `columns` argument to specify a single column label or a list of column labels. The `columns` argument is an alternative to specifying `axis=1` with labels. You cannot specify both labels and columns.

Examples Prerequisite

Assume the table "admissions_train" exists and its index column is "id". And a DataFrame "df" is created based on this table using the command:

```
>>> df = DataFrame("admissions_train")

>>> df
   masters  gpa  stats programming admitted
id
5         no  3.44  novice      novice      0
7         yes  2.33  novice      novice      1
22        yes  3.46  novice  beginner      0
17         no  3.83  advanced  advanced      1
13         no  4.00  advanced  novice      1
19         yes  1.98  advanced  advanced      0
36         no  3.00  advanced  novice      0
15         yes  4.00  advanced  advanced      1
34         yes  3.85  advanced  beginner      0
40         yes  3.95  novice  beginner      0
```

Example 1: Drop columns 'stats' and 'admitted' with axis=1

```
>>> df.drop(['stats', 'admitted'], axis=1)

   programming masters  gpa
id
5         novice      no  3.44
34    beginner      yes  3.85
13         novice      no  4.00
40    beginner      yes  3.95
```

| | | | |
|----|----------|-----|------|
| 22 | beginner | yes | 3.46 |
| 19 | advanced | yes | 1.98 |
| 36 | novice | no | 3.00 |
| 15 | advanced | yes | 4.00 |
| 7 | novice | yes | 2.33 |
| 17 | advanced | no | 3.83 |

Example 2: Drop columns 'stats' and 'admitted' with columns argument

```
>>> df.drop(columns=['stats', 'admitted'])
```

| | programming masters | | gpa |
|----|---------------------|-----|------|
| id | | | |
| 5 | novice | no | 3.44 |
| 34 | beginner | yes | 3.85 |
| 13 | novice | no | 4.00 |
| 19 | advanced | yes | 1.98 |
| 15 | advanced | yes | 4.00 |
| 40 | beginner | yes | 3.95 |
| 7 | novice | yes | 2.33 |
| 22 | beginner | yes | 3.46 |
| 36 | novice | no | 3.00 |
| 17 | advanced | no | 3.83 |

Example 3: Drop rows with index values 13 and 15 with axis=0

```
>>> df1 = df[df.gpa == 4.00]
>>> df1
```

| | masters | gpa | stats | programming | admitted |
|----|---------|-----|----------|-------------|----------|
| id | | | | | |
| 13 | no | 4.0 | Advanced | Novice | 1 |
| 29 | yes | 4.0 | Novice | Beginner | 0 |
| 15 | yes | 4.0 | Advanced | Advanced | 1 |

```
>>> df1.drop([13,15], axis=0)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|-----|--------|-------------|----------|
| id | | | | | |
| 29 | yes | 4.0 | Novice | Beginner | 0 |

```
>>>
```

dropna() Method

Use the `dropna()` method to remove rows with null values in a DataFrame.

Arguments:

- *how*: optional argument specifies how rows are removed. It has options 'any' or 'all'.
 - 'any': Removes rows with at least one null value.
 - 'all': Removes rows with all null values.

The default is 'any'.
- *thresh*: optional argument specifies the minimum number of non-null values in a row to include.
- *subset*: optional argument specifies list of column names to include, in array-like format. Use this argument to limit the search for null values to specific columns.

Examples Prerequisite

Assume the table "sales" exists. And a DataFrame "df" is created using the command:

```
>>> df = DataFrame("sales")

>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Yellow Inc | 90.0 | None | None | None | 2017-04-01 |
| Orange Inc | 210.0 | None | None | 250 | 2017-04-01 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2017-04-01 |

Example 1: Drop rows with at least one Null value

```
>>> df.dropna()
```

| | Feb | Jan | Mar | Apr | datetime |
|-----------|-------|-----|-----|-----|------------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |

Example 2: Keep rows with at least four Non-Null values

```
>>> df.dropna(thresh=4)
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Orange Inc | 210.0 | None | None | 250 | 2017-04-01 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2017-04-01 |

Example 3: Keep rows with at least five Non-Null values

```
>>> df.dropna(thresh=5)
```

| | Feb | Jan | Mar | Apr | datetime |
|-----------|-------|-----|-----|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2017-04-01 |

Example 4: Drop rows with all Null values in columns 'Jan' and 'Mar'

```
>>> df.dropna(how='all', subset=['Jan', 'Mar'])
```

| | Feb | Jan | Mar | Apr | datetime |
|-----------|-------|-----|-----|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2017-04-01 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |

assign() Method

Use the `assign()` method to assign new column expressions in a teradataml DataFrame. A new DataFrame is returned without modifying the existing DataFrame.

```
assign(self, drop_columns = False, **kwargs)
```

The expressions are given as key value pairs where the keys are column names and the values are column expressions. The values can include arithmetic expressions that involve supported python literals and columns (ColumnExpression instances) from the DataFrame.

When the 'drop_columns' parameter is True, it removes columns from the resulting DataFrame if they are not specified in assign. It is False by default, so columns from the previous DataFrame are retained.

Note:

Refer to [teradataml DataFrame Column](#) for more details about ColumnExpressions in teradataml.

Note:

- The values in kwargs is not callable for now.
- Since kwargs is a dictionary, the order of your arguments may not be preserved. To make things predictable, the columns are inserted in alphabetical order, at the end of your DataFrame. Assigning multiple columns within the same assign() is possible, but you cannot reference other columns created within the same assign call.
- If no kwargs are given, the function returns self.
- The maximum number of columns in a DataFrame is 2048.

Supported Types and Operators

Python int, float, Decimal, str and None literals can be used in assign expressions. All arithmetic expressions except floor division (//) and power (**) are supported.

Example 1: Add new columns, retaining original DataFrame columns in resulting DataFrame

This example adds new columns created using arithmetic operations and constants, retaining original DataFrame columns in resulting DataFrame.

```
>>> df = DataFrame("iris_test")
>>> df
```

| | sepal_length | sepal_width | petal_length | petal_width | species |
|-----|--------------|-------------|--------------|-------------|---------|
| id | | | | | |
| 120 | 6.0 | 2.2 | 5.0 | 1.5 | 3 |
| 20 | 5.1 | 3.8 | 1.5 | 0.3 | 1 |
| 60 | 5.2 | 2.7 | 3.9 | 1.4 | 2 |
| 10 | 4.9 | 3.1 | 1.5 | 0.1 | 1 |
| 15 | 5.8 | 4.0 | 1.2 | 0.2 | 1 |
| 30 | 4.7 | 3.2 | 1.6 | 0.2 | 1 |
| 70 | 5.6 | 2.5 | 3.9 | 1.1 | 2 |
| 65 | 5.6 | 2.9 | 3.6 | 1.3 | 2 |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | 1 |
| 80 | 5.7 | 2.6 | 3.5 | 1.0 | 2 |

Alias the columns to use in assign:

```
>>> s_len = df.sepal_length
>>> p_len = df.petal_length
```

Add new column expressions to DataFrame:

```
>>> df.select(['sepal_length', 'petal_length']).\
...     assign(sum = s_len + p_len,
...           diff = s_len - p_len,
...           prod = s_len * p_len,
...           div = s_len / p_len,
...           mod = s_len % p_len,
...           num_constant = 1,
...           str_constant = 'string')
```

| | sepal_length | petal_length | diff | div | mod | num_constant | prod | str_constant | sum |
|---|--------------|--------------|------|----------|-----|--------------|-------|--------------|------|
| 0 | 5.6 | 3.9 | 1.7 | 1.435897 | 1.7 | 1 | 21.84 | string | 9.5 |
| 1 | 5.7 | 3.5 | 2.2 | 1.628571 | 2.2 | 1 | 19.95 | string | 9.2 |
| 2 | 6.0 | 5.0 | 1.0 | 1.200000 | 1.0 | 1 | 30.00 | string | 11.0 |
| 3 | 4.9 | 1.5 | 3.4 | 3.266667 | 0.4 | 1 | 7.35 | string | 6.4 |
| 4 | 5.0 | 1.4 | 3.6 | 3.571429 | 0.8 | 1 | 7.00 | string | 6.4 |
| 5 | 5.1 | 1.5 | 3.6 | 3.400000 | 0.6 | 1 | 7.65 | string | 6.6 |
| 6 | 5.2 | 3.9 | 1.3 | 1.333333 | 1.3 | 1 | 20.28 | string | 9.1 |
| 7 | 5.6 | 3.6 | 2.0 | 1.555556 | 2.0 | 1 | 20.16 | string | 9.2 |
| 8 | 5.1 | 1.5 | 3.6 | 3.400000 | 0.6 | 1 | 7.65 | string | 6.6 |
| 9 | 4.7 | 1.6 | 3.1 | 2.937500 | 1.5 | 1 | 7.52 | string | 6.3 |

Example 2: Add new columns, dropping original DataFrame columns in resulting DataFrame

This example adds new columns created using arithmetic operations and constants, dropping original DataFrame columns in resulting DataFrame.

```
>>> df.assign(drop_columns = True,
...           sum = s_len + p_len,
...           diff = s_len - p_len,
...           prod = s_len * p_len,
...           div = s_len / p_len,
...           mod = s_len % 2,
...           num_constant = 1,
...           str_constant = 'string'
... )
```

| | diff | div | mod | num_constant | prod | str_constant | sum |
|---|------|----------|-----|--------------|-------|--------------|------|
| 0 | 1.0 | 1.200000 | 0.0 | 1 | 30.00 | string | 11.0 |
| 1 | 3.1 | 2.937500 | 0.7 | 1 | 7.52 | string | 6.3 |
| 2 | 1.7 | 1.435897 | 1.6 | 1 | 21.84 | string | 9.5 |
| 3 | 3.6 | 3.571429 | 1.0 | 1 | 7.00 | string | 6.4 |
| 4 | 1.3 | 1.333333 | 1.2 | 1 | 20.28 | string | 9.1 |
| 5 | 3.4 | 3.266667 | 0.9 | 1 | 7.35 | string | 6.4 |
| 6 | 2.0 | 1.555556 | 1.6 | 1 | 20.16 | string | 9.2 |
| 7 | 3.6 | 3.400000 | 1.1 | 1 | 7.65 | string | 6.6 |
| 8 | 4.6 | 4.833333 | 1.8 | 1 | 6.96 | string | 7.0 |
| 9 | 2.2 | 1.628571 | 1.7 | 1 | 19.95 | string | 9.2 |

Regular Aggregate Functions Supported by DataFrame

teradataml DataFrame supports following set of regular aggregate functions which can be used with and without DataFrame.groupby().

See the [DataFrame Aggregate Functions](#) section of *Teradata Package for Python Function Reference*, B700-4008) at <https://docs.teradata.com/> for detailed description and usage examples of these functions.

| Sr. No. | Function Name | Description |
|---------|---------------|---|
| 1 | corr() | Returns the Sample Pearson product moment correlation coefficient of its arguments for all non-null data point pairs. |
| 2 | count() | Returns column-wise count of the dataframe. |
| 3 | covar_pop() | Returns the column-wise population covariance of its arguments for all non-null data point pairs. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |
| 4 | covar_samp() | Returns the column-wise sample covariance of its arguments for all non-null data point pairs. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |
| 5 | kurtosis() | Returns column-wise kurtosis value of the dataframe. Kurtosis is the fourth moment of the distribution of the standardized (z) values. It is a measure of the outlier (rare, extreme observation) character of the distribution as compared with the normal (or Gaussian) distribution. <ul style="list-style-type: none"> • The normal distribution has a kurtosis of 0. • Positive kurtosis indicates that the distribution is more outlier-prone than the normal distribution. • Negative kurtosis indicates that the distribution is less outlier-prone than the normal distribution. |
| 6 | max() | Returns column-wise maximum value of the dataframe. |
| 7 | mean() | Returns column-wise mean value of the dataframe. |
| 8 | median() | Returns column-wise median value of the dataframe. |
| 9 | min() | Returns column-wise minimum value of the dataframe. |
| 10 | percentile() | Return the value which represents the desired percentile. |
| 11 | regr_avgx() | Returns the column-wise mean of the independent variable for all non-null data pairs of the dependent and an independent variable arguments. |
| 12 | regr_avgy() | Returns the column-wise mean of the dependent variable for all non-null data pairs of the dependent and independent variable arguments. |

| Sr. No. | Function Name | Description |
|---------|-------------------------------|---|
| 13 | <code>regr_count()</code> | Returns the column-wise count of all non-null data pairs of the dependent and independent variable arguments. |
| 14 | <code>regr_intercept()</code> | Returns the column-wise intercept of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments. The intercept is the point at which the regression line through the non-null data pairs in the sample intersects the ordinate, or y-axis, of the graph. |
| 15 | <code>regr_r2()</code> | Returns the column-wise coefficient of determination for all non-null data pairs of the dependent and independent variable arguments. |
| 16 | <code>regr_slope()</code> | Returns the column-wise coefficient slope of the univariate linear regression line through all non-null data pairs of the dependent and an independent variable arguments. |
| 17 | <code>regr_sxx()</code> | Returns the column-wise sum of the squares of the independent variable expression for all non-null data pairs of dependent and an independent variable arguments. |
| 18 | <code>regr_sxy()</code> | Returns the column-wise sum of the products of the independent variable and the dependent variable for all non-null data pairs of the dependent and independent variable arguments. |
| 19 | <code>regr_syy()</code> | Returns the column-wise sum of the squares of the dependent variable expression for all non-null data pairs of dependent and an independent variable arguments. |
| 20 | <code>skew()</code> | Returns column-wise skewness of the distribution of the dataframe. Skewness is the third moment of a distribution. It is a measure of the asymmetry of the distribution about its mean compared with the normal (or Gaussian) distribution. <ul style="list-style-type: none"> • The normal distribution has a skewness of 0. • Positive skewness indicates a distribution having an asymmetric tail extending toward more positive values. • Negative skewness indicates an asymmetric tail extending toward more negative values. |
| 21 | <code>std()</code> | Returns column-wise sample or population standard deviation value of the dataframe. The standard deviation is the second moment of a distribution. <ul style="list-style-type: none"> • For a sample, it is a measure of dispersion from the mean of that sample. • For a population, it is a measure of dispersion from the mean of that population. The computation is more conservative for the population standard deviation to minimize the effect of outliers on the computed value. |
| 22 | <code>sum()</code> | Returns column-wise sum value of the dataframe. |
| 23 | <code>var()</code> | Returns column-wise sample or population variance of the columns in a dataframe. |

| Sr. No. | Function Name | Description |
|---|---------------|---|
| | | <ul style="list-style-type: none"> The variance of a population is a measure of dispersion from the mean of that population. The variance of a sample is a measure of dispersion from the mean of that sample. It is the square of the sample standard deviation. |
| 24 | agg() | Perform aggregates using one or more operations. |
| teradataml special aggregate functions | | |
| 25 | csum() | Returns column-wise cumulative sum value for rows in the partition of the dataframe. |
| 26 | msum() | Computes the moving sum for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |
| 27 | mavg() | Computes the moving average for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |
| 28 | mdiff() | Computes the moving difference for the current row and the preceding "width" rows in a partition, by sorting the rows according to "sort_columns". |
| 29 | mlinearreg() | Computes the moving linear regression for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |

groupby() Method

Use the groupby() method to group one or more columns of a DataFrame.

The method takes a column name or a list of column names to group by.

Note:

- You can still apply teradataml DataFrame methods (filters/sort/etc) on top of the result of this one.
- Consecutive operations of grouping, i.e., groupby_time(), resample() and groupby() are not permitted. An exception will be raised. Following are some cases where exception will be raised as "Invalid operation applied, check documentation for correct usage."
 - df.groupby().groupby()
 - df.groupby().resample()
 - df.groupby().groupby_time()

Examples Prerequisite

Assume a teradata DataFrame "df" is created based on the table "admissions_train", using the command:

```
>>> df = DataFrame("admissions_train")
```

```
>>> df
   masters  gpa  stats programming admitted
id
13      no  4.00  Advanced      Novice      1
26     yes  3.57  Advanced    Advanced      1
5       no  3.44   Novice      Novice      0
19     yes  1.98  Advanced    Advanced      0
15     yes  4.00  Advanced    Advanced      1
40     yes  3.95   Novice    Beginner      0
7       yes  2.33   Novice      Novice      1
22     yes  3.46   Novice    Beginner      0
36      no  3.00  Advanced      Novice      0
38     yes  2.65  Advanced    Beginner      1
```

Example 1: Groups by one column and finds the min

The following example groups by column "masters" and finds the min for the groups in "masters":

```
>>> df2 = df.groupby("masters")

>>> df2.min()
   masters  min_id  min_gpa  min_stats  min_programming  min_admitted
0      no         3    1.87  advanced      advanced         0
1     yes         1    1.98  advanced      advanced         0
```

Example 2: Groups by two columns and finds the min and max

This example finds the min and max grouped by "masters" and "programming":

```
>>> df3 = df.groupby(["masters", "programming"])

>>> df3.min()
   masters programming  min_id  min_gpa  min_stats  min_admitted
0      no    Advanced      8    3.13  Advanced         1
1     yes    Beginner      1    2.65  Advanced         0
2     yes     Novice      4    2.33  Advanced         0
3      no    Beginner      3    3.68   Novice         1
4     yes    Advanced      6    1.98  Advanced         0
5      no     Novice      5    1.87  Advanced         0

>>> df3.max()
   masters programming  max_id  max_gpa  max_stats  max_admitted
0      no     Novice      37    4.00   Novice         1
1     no    Advanced      28    3.96  Beginner         1
```

| | | | | | | |
|---|-----|----------|----|------|----------|---|
| 2 | no | Beginner | 35 | 3.87 | Novice | 1 |
| 3 | yes | Advanced | 27 | 4.00 | Beginner | 1 |
| 4 | yes | Novice | 30 | 3.79 | Novice | 1 |
| 5 | yes | Beginner | 40 | 4.00 | Novice | 1 |

Example 3: Select multiple columns, followed by groupby and find the min

This example selects "id", "masters", "gpa", "stats", followed by a groupby on "masters", and finds the min.

```
>>> df1 = df.select(["id", "masters", "gpa", "stats"])
```

```
>>> df2 = df1.groupby("masters")
```

```
>>> df2.min()
  masters min_id min_gpa min_stats
0      no      3    1.87  advanced
1     yes      1    1.98  advanced
```

Time Series Aggregates

The Time Series aggregate functions help perform aggregate operations on time series data.

In this section, you can find examples using APIs to group or resample time series data and other Time Series Aggregate functions.

Example Setup

Examples for Time Series Aggregates functions share same steps to set up the environment.

The following lists the procedure to load the example datasets and create required DataFrames to prepare for the examples of Time Series Aggregate functions.

- Load the example datasets

```
>>> load_example_data("dataframe", ["ocean_buoys",
  "ocean_buoys_seq", "ocean_buoys_nonpti"])
```

- Create required DataFrames:
 - DataFrame on non-sequenced PTI table

```
>>> ocean_buoys = DataFrame("ocean_buoys")
```

Check the DataFrame columns:

```
>>> ocean_buoys.columns
['buoyid', 'TD_TIMECODE', 'temperature', 'salinity']
```

Check the head of the DataFrame:

```
>>> ocean_buoys.head()
buoyid  TD_TIMECODE  temperature  salinity
0      2014-01-06 08:10:00.000000    100.0      55
0      2014-01-06 08:08:59.999999      NaN      55
1      2014-01-06 09:01:25.122200     77.0      55
1      2014-01-06 09:03:25.122200     79.0      55
1      2014-01-06 09:01:25.122200     70.0      55
1      2014-01-06 09:02:25.122200     71.0      55
1      2014-01-06 09:03:25.122200     72.0      55
0      2014-01-06 08:09:59.999999     99.0      55
0      2014-01-06 08:00:00.000000     10.0      55
0      2014-01-06 08:10:00.000000     10.0      55
```

- DataFrame on sequenced PTI table

```
>>> ocean_buoys_seq = DataFrame("ocean_buoys_seq")
```

Check the DataFrame columns:

```
>>> ocean_buoys_seq.columns
['TD_TIMECODE', 'TD_SEQNO', 'buoyid', 'salinity',
'temperature', 'dates']
```

Check the head of the DataFrame:

```
>>> ocean_buoys_seq.head()
buoyid  TD_TIMECODE  TD_SEQNO  salinity
0      2014-01-06 08:00:00.000000    26      55
10.0    2016-02-26
0      2014-01-06 08:08:59.999999    18      55
NaN     2015-06-18
1      2014-01-06 09:02:25.122200    24      55
78.0    2015-12-24
1      2014-01-06 09:01:25.122200    23      55
77.0    2015-11-23
1      2014-01-06 09:02:25.122200    12      55
71.0    2014-12-12
1      2014-01-06 09:03:25.122200    13      55
72.0    2015-01-13
1      2014-01-06 09:01:25.122200    11      55
70.0    2014-11-11
0      2014-01-06 08:10:00.000000    19      55
10.0    2015-07-19
```



```

0      2014-01-06 08:09:59.999999      17      55
99.0  2015-05-17
0      2014-01-06 08:10:00.000000      27      55
100.0  2016-03-27

```

- DataFrame on non-PTI table

```
>>> ocean_buoys_nonpti = DataFrame("ocean_buoys_nonpti")
```

Check the DataFrame columns:

```
>>> ocean_buoys_nonpti.columns
['buoyid', 'timecode', 'temperature', 'salinity']
```

Check the head of the DataFrame:

```
>>> ocean_buoys_nonpti.head()
           buoyid  temperature  salinity
timecode
2014-01-06 08:09:59.999999      0      99.0      55
2014-01-06 08:10:00.000000      0      10.0      55
2014-01-06 09:01:25.122200      1      70.0      55
2014-01-06 09:01:25.122200      1      77.0      55
2014-01-06 09:02:25.122200      1      71.0      55
2014-01-06 09:03:25.122200      1      72.0      55
2014-01-06 09:02:25.122200      1      78.0      55
2014-01-06 08:10:00.000000      0     100.0      55
2014-01-06 08:08:59.999999      0       NaN      55
2014-01-06 08:00:00.000000      0      10.0      55

```

groupby_time()

The `groupby_time()` function resamples time series data to group the same by time on a datetime column of a teradataml DataFrame. It also allows grouping based on teradataml DataFrame columns.

Although the grouping is optimized for DataFrames created for PTI tables, it is also supported on non-PTI tables when the argument `'timecode_column'` is specified.

Note:

- This API is similar to `resample()`.
- You can still apply teradataml DataFrame methods (filters/sort/etc) on top of the result of this one.
- Consecutive operations of grouping, i.e., `groupby_time()`, `resample()` and `groupby()` are not permitted. An exception will be raised. Following are some cases where exception will be raised as "Invalid operation applied, check documentation for correct usage."
 - `df.groupby_time().groupby()`
 - `df.groupby_time().resample()`
 - `df.groupby_time().groupby_time()`

Examples Prerequisite

- Load the example datasets:

```
>>> load_example_data("dataframe", ["ocean_buoys", "ocean_buoys_nonpti"])
```

- Create required DataFrames:

- DataFrame on non-sequenced PTI table

```
>>> ocean_buoys = DataFrame("ocean_buoys")
```

Check the DataFrame columns:

```
>>> ocean_buoys.columns
['buoyid', 'TD_TIMECODE', 'temperature', 'salinity']
```

Check the head of the DataFrame:

```
>>> ocean_buoys.head()
          TD_TIMECODE  temperature  salinity
buoyid
0      2014-01-06 08:10:00.000000      100.0      55
0      2014-01-06 08:08:59.999999        NaN      55
1      2014-01-06 09:01:25.122200       77.0      55
1      2014-01-06 09:03:25.122200       79.0      55
1      2014-01-06 09:01:25.122200       70.0      55
1      2014-01-06 09:02:25.122200       71.0      55
1      2014-01-06 09:03:25.122200       72.0      55
0      2014-01-06 08:09:59.999999       99.0      55
```

```
0      2014-01-06 08:00:00.000000      10.0      55
0      2014-01-06 08:10:00.000000      10.0      55
```

- DataFrame on non-PTI table

```
>>> ocean_buoys_nonpti = DataFrame("ocean_buoys_nonpti")
```

Check the DataFrame columns:

```
>>> ocean_buoys_nonpti.columns
['buoyid', 'timecode', 'temperature', 'salinity']
```

Check the head of the DataFrame:

```
>>> ocean_buoys_nonpti.head()
           buoyid  temperature  salinity
timecode
2014-01-06 08:09:59.999999      0      99.0      55
2014-01-06 08:10:00.000000      0      10.0      55
2014-01-06 09:01:25.122200      1      70.0      55
2014-01-06 09:01:25.122200      1      77.0      55
2014-01-06 09:02:25.122200      1      71.0      55
2014-01-06 09:03:25.122200      1      72.0      55
2014-01-06 09:02:25.122200      1      78.0      55
2014-01-06 08:10:00.000000      0     100.0      55
2014-01-06 08:08:59.999999      0       NaN      55
2014-01-06 08:00:00.000000      0      10.0      55
```

Example 1: Group by timebucket of two calendar years, on DataFrame created on non-sequenced PTI table

Use formal notation and 'buoyid' column on DataFrame created on non-sequenced PTI table.

Fill missing values with Nulls.

```
>>> ocean_buoys_grpby1 =
ocean_buoys.groupby_time(timebucket_duration="CAL_YEARS(2)",
value_expression="buoyid", fill="NULLS")
```

```
>>> number_of_values_to_column = {2: "temperature"}
```

```
>>> ocean_buoys_grpby1.bottom(number_of_values_to_column).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(CAL_YEARS(2))  buoyid  bottom2temperature
0  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      0      10
1  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      0      10
2  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      1      71
3  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      1      70
4  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      2      80
5  ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      2      81
```

| | | | | |
|---|---|---|----|----|
| 6 | ('2014-01-01 00:00:00.000000-00:00', '2016-01-... | 2 | 44 | 43 |
| 7 | ('2014-01-01 00:00:00.000000-00:00', '2016-01-... | 2 | 44 | 43 |

Example 2: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with Nulls.

Note:

'timecode_column' must be specified for non-PTI table.

```
>>> ocean_buoys_nonpti_grpby2 =
ocean_buoys_nonpti.groupby_time(timebucket_duration="2m",
value_expression="buoyid", timecode_column="timecode", fill="NULLS")

>>> number_of_values_to_column = {2: "temperature"}

>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])
buoyid bottom_with_ties2temperature TIMECODE_RANGE GROUP BY TIME(MINUTES(2))
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10.0
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 NaN
2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 NaN
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 NaN
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99.0
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100.0
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10.0
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70.0
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77.0
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71.0
```

Example 3: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with previous values.

Note:

'timecode_column' must be specified for non-PTI table.

```
>>> ocean_buoys_nonpti_grpby2 =
ocean_buoys_nonpti.groupby_time(timebucket_duration="2mins",
value_expression="buoyid", timecode_column="timecode", fill="prev")

>>> number_of_values_to_column = {2: "temperature"}
```

```
>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(MINUTES(2))
buoyid bottom_with_ties2temperature
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 10
2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 10
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 10
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71
```

Example 4: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with numeric constant 12345.

Note:

'timecode_column' must be specified for non-PTI table.

```
>>> ocean_buoys_nonpti_grpby2 =
ocean_buoys_nonpti.groupby_time(timebucket_duration="2minute",
value_expression="buoyid", timecode_column="timecode", fill=12345)
```

```
>>> number_of_values_to_column = {2: "temperature"}
```

```
>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(MINUTES(2))
buoyid bottom_with_ties2temperature
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 12345
2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 12345
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 12345
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71
```

resample()

The `resample()` function resamples time series data to group the same by time on a datetime column of a teradataml DataFrame. It also allows grouping done based on teradataml DataFrame columns.

This function applies Group By Time to one or more columns of a teradataml DataFrame. Outcome of this function can be used to run Time Series Aggregate functions.

Although the grouping is optimized for DataFrames created for PTI tables, it is also supported on non-PTI tables when the argument 'timecode_column' is specified.

Note:

- This API is similar to `groupby_time()`.
- You can still apply teradataml DataFrame methods (filters/sort/etc) on top of the result of this one.
- Consecutive operations of grouping, i.e., `groupby_time()`, `resample()` and `groupby()` are not permitted. An exception will be raised. Following are some cases where exception will be raised as "Invalid operation applied, check documentation for correct usage."
 - `df.resample().groupby()`
 - `df.resample().resample()`
 - `df.resample().groupby_time()`

Examples Prerequisite

See [Example Setup](#) to set up the environment for the following examples.

Example 1: Group by timebucket of two calendar years, on DataFrame created on non-sequenced PTI table

Use formal notation and 'buoyid' column on DataFrame created on non-sequenced PTI table.

Fill missing values with Nulls.

```
>>> ocean_buoys_grpby1 = ocean_buoys.resample(rule="CAL_YEARS(2)",
value_expression="buoyid", fill_method="NULLS")
```

```
>>> number_of_values_to_column = {2: "temperature"}
```

```
>>> ocean_buoys_grpby1.bottom(number_of_values_to_column).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(CAL_YEARS(2))  buoyid  bottom2temperature
0 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      0      10
1 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      0      10
2 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      1      71
3 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      1      70
4 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      2      80
5 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2      2      81
6 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2     44      43
7 ('2014-01-01 00:00:00.000000-00:00', '2016-01-...  2     44      43
```

Example 2: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with Nulls.

Note:

Time column must be specified for non-PTI table to 'on' argument.

```
>>> ocean_buoys_nonpti_grpby2 = ocean_buoys_nonpti.resample(rule="2m",
value_expression="buoyid", on="timecode", fill_method="NULLS")

>>> number_of_values_to_column = {2: "temperature"}

>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(MINUTES(2))
buoyid bottom_with_ties2temperature
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10.0
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 NaN
2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 NaN
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 NaN
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99.0
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100.0
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10.0
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70.0
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77.0
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71.0
```

Example 3: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with previous values.

Note:

Time column must be specified for non-PTI table to 'on' argument.

```
>>> ocean_buoys_nonpti_grpby2 = ocean_buoys_nonpti.resample(rule="2m",
value_expression="buoyid", on="timecode", fill_method="prev")

>>> number_of_values_to_column = {2: "temperature"}

>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])
TIMECODE_RANGE  GROUP BY TIME(MINUTES(2))
buoyid bottom_with_ties2temperature
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 10
```

```

2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 10
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 10
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71

```

Example 4: Group by timebucket of two minutes, on DataFrame created on non-PTI table

Use shorthand notation to specify timebucket, on DataFrame created on non-PTI table.

Fill missing values with numeric constant 12345.

Note:

Time column must be specified for non-PTI table to 'on' argument.

```
>>> ocean_buoys_nonpti_grpby2 = ocean_buoys_nonpti.resample(rule="2minute",
value_expression="buoyid", on="timecode", fill_method=12345)
```

```
>>> number_of_values_to_column = {2: "temperature"}
```

```
>>> ocean_buoys_nonpti_grpby2.bottom(number_of_values_to_column, with_ties=True).sort(["TIMECODE_RANGE", "buoyid"])

buoyid bottom_with_ties2temperature TIMECODE_RANGE GROUP BY TIME(MINUTES(2))
0 ('2014-01-06 08:00:00.000000+00:00', '2014-01-... 11574961
0 10
1 ('2014-01-06 08:02:00.000000+00:00', '2014-01-... 11574962
0 12345
2 ('2014-01-06 08:04:00.000000+00:00', '2014-01-... 11574963
0 12345
3 ('2014-01-06 08:06:00.000000+00:00', '2014-01-... 11574964
0 12345
4 ('2014-01-06 08:08:00.000000+00:00', '2014-01-... 11574965
0 99
5 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 10
6 ('2014-01-06 08:10:00.000000+00:00', '2014-01-... 11574966
0 100
7 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 77
8 ('2014-01-06 09:00:00.000000+00:00', '2014-01-... 11574991
1 70
9 ('2014-01-06 09:02:00.000000+00:00', '2014-01-... 11574992
1 71

```

Time Series Aggregate Functions

teradataml supports following set of time series aggregate functions which can be invoked on DataFrame.groupby_time() or DataFrame.resample().

See the [teradataml: Time Series Functions](#) section of *Teradata Package for Python Function Reference*, B700-4008) at <https://docs.teradata.com/> for detailed description and usage examples of these functions.

| Sr. No. | Function Name | Description |
|---------|---------------|---|
| 1 | bottom() | Returns the smallest number of values in the columns for each group, with or without ties. |
| 2 | count() | Returns column-wise count for each group. |
| 3 | describe() | Generates statistics for numeric columns. It computes max, mean, min, std, median, mode, and percentiles for numeric columns. Default statistics include: 'max', 'mean', 'min', 'std'. |
| 4 | delta_t() | Calculates the time difference, or DELTA_T, between a starting and an ending event. The calculation is performed against a time-ordered time series data set. |
| 5 | first() | Returns the oldest value, determined by the timecode, for each group. |
| 6 | kurtosis() | Returns column-wise kurtosis value for each group. Kurtosis is the fourth moment of the distribution of the standardized (z) values. It is a measure of the outlier (rare, extreme observation) character of the distribution as compared with the normal (or Gaussian) distribution. <ul style="list-style-type: none"> • The normal distribution has a kurtosis of 0. • Positive kurtosis indicates that the distribution is more outlier-prone than the normal distribution. • Negative kurtosis indicates that the distribution is less outlier-prone than the normal distribution. |
| 7 | last() | Returns the newest value, determined by the timecode, for each group. |
| 8 | mad() | Median Absolute Deviation (MAD) returns the median of the set of values defined as the absolute value of the difference between each value and the median of all values in each group. |
| 9 | max() | Returns column-wise maximum value for each group. |
| 10 | mean() | Returns column-wise mean value for each group. |
| 11 | median() | Returns column-wise median value for each group. |
| 12 | min() | Returns column-wise minimum value for each group. |
| 13 | mode() | Returns the column-wise mode of all values in each group. |
| 14 | percentile() | Return the value which represents the desired percentile from each group. The result value is determined by the desired index (di) in an ordered list of values. The following equation is for the di: $di = (\text{number of values in group} - 1) * \text{percentile}/100$ When di is a whole number, that value is the returned result. The di can also be between two data points, i and j, where $i < j$. In that case, the result is interpolated according to the value specified in interpolation argument. |
| 15 | skew() | Returns column-wise skewness of the distribution for each group. |

| Sr. No. | Function Name | Description |
|---------|---------------|--|
| | | <p>Skewness is the third moment of a distribution. It is a measure of the asymmetry of the distribution about its mean compared with the normal (or Gaussian) distribution.</p> <ul style="list-style-type: none"> • The normal distribution has a skewness of 0. • Positive skewness indicates a distribution having an asymmetric tail extending toward more positive values. • Negative skewness indicates an asymmetric tail extending toward more negative values. |
| 16 | std() | <p>Returns column-wise sample or population standard deviation value for each group. The standard deviation is the second moment of a distribution.</p> <ul style="list-style-type: none"> • For a sample, it is a measure of dispersion from the mean of that sample. • For a population, it is a measure of dispersion from the mean of that population. <p>The computation is more conservative for the population standard deviation to minimize the effect of outliers on the computed value.</p> |
| 17 | sum() | Returns column-wise sum value for each group. |
| 18 | var() | <p>Returns column-wise sample or population variance of the columns for each group.</p> <ul style="list-style-type: none"> • The variance of a population is a measure of dispersion from the mean of that population. • The variance of a sample is a measure of dispersion from the mean of that sample. It is the square of the sample standard deviation. |
| 19 | top() | Returns the largest number of values in the columns for each group, with or without ties. |

concat() Method

Use the `concat()` method to concatenate two teradataml DataFrame objects along the index axis. The operation is performed by carrying out a database-style union or union all operation.

Examples Prerequisite

Assume the table "admissions_train" exists and a DataFrame "df" is created based on this table using the command:

```
>>> df = DataFrame("admissions_train")

>>> df
  masters  gpa  stats programming admitted
id
22    yes  3.46   Novice    Beginner      0
36     no  3.00  Advanced     Novice      0
```

| | | | | | |
|----|-----|------|----------|----------|---|
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

```
>>> df1 = df[df.gpa == 4].select(['id', 'stats', 'masters', 'gpa'])
```

```
>>> df1
```

| | stats | masters | gpa |
|----|----------|---------|-----|
| id | | | |
| 13 | Advanced | no | 4.0 |
| 29 | Novice | yes | 4.0 |
| 15 | Advanced | yes | 4.0 |

```
>>> df2 = df[df.gpa < 2].select(['id', 'stats', 'programming', 'admitted'])
```

```
>>> df2
```

| | stats | programming | admitted |
|----|----------|-------------|----------|
| id | | | |
| 24 | Advanced | Novice | 1 |
| 19 | Advanced | Advanced | 0 |

Example 1: Default behavior with default values for the optional arguments.

```
>>> # Default options
```

```
>>> cdf = df1.concat(df2)
```

```
>>> cdf
```

| | stats | masters | gpa | programming | admitted |
|----|----------|---------|-----|-------------|----------|
| id | | | | | |
| 19 | Advanced | None | NaN | Advanced | 0 |
| 24 | Advanced | None | NaN | Novice | 1 |
| 13 | Advanced | no | 4.0 | None | None |
| 29 | Novice | yes | 4.0 | None | None |
| 15 | Advanced | yes | 4.0 | None | None |

Example 2: concat() operation with 'join = inner'.

```
>>> cdf = df1.concat(df2, join='inner')
```

```
>>> cdf
```

| | stats |
|----|----------|
| id | |
| 19 | Advanced |

```

24 Advanced
13 Advanced
29 Novice
15 Advanced

```

Example 3: concat() operation with 'allow_duplicates=False'.

```

>>> # allow_duplicates = True (default)
>>> cdf = df1.concat(df2)
>>> cdf
      stats masters  gpa programming admitted
id
19  Advanced    None  NaN    Advanced         0
24  Advanced    None  NaN     Novice         1
13  Advanced     no  4.0      None        None
29   Novice     yes  4.0      None        None
15  Advanced     yes  4.0      None        None

>>> cdf = cdf.concat(df2)
>>> cdf
      stats masters  gpa programming admitted
id
19  Advanced    None  NaN    Advanced         0
13  Advanced     no  4.0      None        None
24  Advanced    None  NaN     Novice         1
24  Advanced    None  NaN     Novice         1
19  Advanced    None  NaN    Advanced         0
29   Novice     yes  4.0      None        None
15  Advanced     yes  4.0      None        None

>>> # allow_duplicates = False
>>> cdf = cdf.concat(df2, allow_duplicates=False)
>>> cdf
      stats masters  gpa programming admitted
id
19  Advanced    None  NaN    Advanced         0
29   Novice     yes  4.0      None        None
24  Advanced    None  NaN     Novice         1
15  Advanced     yes  4.0      None        None
13  Advanced     no  4.0      None        None

```

Example 4: concat() operation with 'sort=True'.

```
>>> cdf = df1.concat(df2, sort=True)
>>> cdf
```

| | admitted | gpa | masters | programming | stats |
|----|----------|-----|---------|-------------|----------|
| id | | | | | |
| 19 | 0 | NaN | None | Advanced | Advanced |
| 24 | 1 | NaN | None | Novice | Advanced |
| 13 | None | 4.0 | no | None | Advanced |
| 29 | None | 4.0 | yes | None | Novice |
| 15 | None | 4.0 | yes | None | Advanced |

describe() Method

The describe() function generates statistics for numeric columns. This function can be used in two modes:

- Regular Aggregate Mode

It computes the count, mean, std, min, percentiles, and max for numeric columns.

Default statistics include: "count", "mean", "std", "min", "percentile", "max".

Note:

If describe() is used on the output of any DataFrame API or groupby(), then it is used in regular aggregate mode.

- Time Series Aggregate Mode

It computes the max, mean, min, std, median, mode, and percentiles for numeric columns.

Default statistics include: 'max', 'mean', 'min', 'std'.

Note:

If describe() is used on the output of groupby_time(), then it is used in time series aggregate mode, where time series aggregates are used to calculate the statistics.

Optional arguments:

- *percentiles*: A list of values between 0 and 1 used for computing percentiles.
The default value is [.25, .5, .75], which generates the 25th, 50th, and 75th percentiles.
- *include*: The values for this argument can be either 'None' or 'all', used to specify if non-numeric columns are included in the computation.
 - If the value is 'all': Both numeric and non-numeric columns are included. The function computes count, mean, std, min, percentiles, and max for numeric columns, and computes count and unique for non-numeric columns.

- If the value is 'None': Only numeric columns are used for collecting statics.

The default value is 'None'.

Note:

Value 'all' is not applicable for Time Series Aggregate Mode.

- *verbose*: Specifies a boolean value to be used for time series aggregation, stating whether to get verbose output or not. When this argument is set to 'True', function calculates median, mode, and percentile values on top of its default statistics.

Note:

Default and the only acceptable value for this argument when used in Regular Aggregate Mode is 'False'.

verbose as 'True' is not applicable for Regular Aggregate Mode.

- *distinct*: Specifies a boolean value to decide whether to consider duplicate rows in statistic calculation or not.

Note:

By default, this argument is set to 'False', which means that duplicate values are considered for statistic calculation.

When this is set to 'True', only distinct rows are considered for statistic calculation.

1. Examples for describe() as Regular Aggregate function

Example Prerequisite

```
>>> df = DataFrame('sales')
```

```
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |

Example 1.1: Generates statistics for DataFrame "sales"

Use default values to computes count, mean, std, min, percentiles, and max for numeric columns.

```
>>> df.describe()
      Apr      Feb      Mar      Jan
func
count      4       6       4       4
mean    195.25  166.667  147.5  137.5
std     70.971   59.554  49.749  62.915
min      101      90      95      50
25%     160.25   117.5  128.75   125
50%      215     200     140     150
75%      250     207.5  158.75   162.5
max      250     210     215     200
```

Example 1.2: Use argument percentiles to compute the 30th and 60th percentiles

```
>>> df.describe(percentiles=[.3, .6])
      Apr      Feb      Mar      Jan
func
count      4       6       4       4
mean    195.25  166.667  147.5  137.5
std     70.971   59.554  49.749  62.915
min      101      90      95      50
30%     172.1     145   135.5   140
60%      236     200     140     150
max      250     210     215     200
```

Example 1.3: Use groupby to compute statistics for specific groups

```
>>> df1 = df.groupby(["datetime", "Feb"])
```

```
>>> df1.describe()
      datetime  Feb  func      Jan      Mar      Apr
04/01/2017  90.0  25%      50      95     101
              50%      50      95     101
              75%      50      95     101
              count      1      1      1
              max      50      95     101
              mean      50      95     101
              min      50      95     101
```

| | | | | |
|-------|-------|------|------|------|
| | std | None | None | None |
| 200.0 | 25% | 150 | 140 | 180 |
| | 50% | 150 | 140 | 180 |
| | 75% | 150 | 140 | 180 |
| | count | 2 | 2 | 1 |
| | max | 150 | 140 | 180 |
| | mean | 150 | 140 | 180 |
| | min | 150 | 140 | 180 |
| | std | 0 | 0 | None |
| 210.0 | 25% | 200 | 215 | 250 |
| | 50% | 200 | 215 | 250 |
| | 75% | 200 | 215 | 250 |
| | count | 1 | 1 | 2 |
| | max | 200 | 215 | 250 |
| | mean | 200 | 215 | 250 |
| | min | 200 | 215 | 250 |
| | std | None | None | 0 |

Example 1.4: Use argument include value 'all' to compute statistics for all columns

Computes count, mean, std, min, percentiles, and max for numeric columns and computes count and unique for non-numeric columns.

```
>>> df.describe(include="all")
```

| | accounts | Feb | Jan | Mar | Apr | datetime |
|--------|----------|---------|--------|--------|--------|----------|
| func | | | | | | |
| 25% | None | 117.5 | 125 | 128.75 | 160.25 | None |
| 75% | None | 207.5 | 162.5 | 158.75 | 250 | None |
| count | 6 | 6 | 4 | 4 | 4 | 6 |
| mean | None | 166.667 | 137.5 | 147.5 | 195.25 | None |
| max | None | 210 | 200 | 215 | 250 | None |
| min | None | 90 | 50 | 95 | 101 | None |
| 50% | None | 200 | 150 | 140 | 215 | None |
| std | None | 59.554 | 62.915 | 49.749 | 70.971 | None |
| unique | 6 | None | None | None | None | 1 |

2. Examples for describe() as Time Series Aggregate function

Examples Prerequisite

See [Example Setup](#) to set up the environment for the following examples.

Example 2.1: Get the basic statistics

Get the basic statistics for time series aggregation for all the numeric columns, use default settings. This example returns max, mean, min and std values.

```
>>> ocean_buoys_grpby.describe()
```

| TIMECODE_RANGE | GROUP BY TIME(CAL_YEARS(2)) | buoyid | func | temperature | salinity |
|--|-----------------------------|--------|------|-------------|----------|
| ('2014-01-01 00:00:00.000000-00:00', '2016-01-0... 2 | | 0 | max | 100 | 55 |
| | | | mean | 54.75 | 55 |
| | | | min | 10 | 55 |
| | | | std | 51.674 | 0 |
| | | 1 | max | 79 | 55 |
| | | | mean | 74.5 | 55 |
| | | | min | 70 | 55 |
| | | | std | 3.937 | 0 |
| | | 2 | max | 82 | 55 |
| | | | mean | 81 | 55 |
| | | | min | 80 | 55 |
| | | | std | 1 | 0 |
| | | 44 | max | 56 | 55 |
| | | | mean | 48.077 | 55 |
| | | | min | 43 | 55 |
| | | | std | 5.766 | 0 |

Example 2.2: Get the verbose statistics

Get the verbose statistics for time series aggregation for all the numeric columns, use default settings. This example returns max, mean, min, std, median, mode, 25th, 50th and 75th percentile.

```
>>> ocean_buoys_grpby.describe(verbose=True)
```

| TIMECODE_RANGE | GROUP BY TIME(CAL_YEARS(2)) | buoyid | func | temperature | salinity |
|--|-----------------------------|--------|--------|-------------|----------|
| ('2014-01-01 00:00:00.000000-00:00', '2016-01-0... 2 | | 0 | 25% | 10 | 55 |
| | | | 50% | 54.5 | 55 |
| | | | 75% | 99.25 | 55 |
| | | | max | 100 | 55 |
| | | | mean | 54.75 | 55 |
| | | | median | 54.5 | 55 |
| | | | min | 10 | 55 |
| | | | mode | 10 | 55 |
| | | | std | 51.674 | 0 |
| | | 1 | 25% | 71.25 | 55 |
| | | | 50% | 74.5 | 55 |
| | | | 75% | 77.75 | 55 |
| | | | max | 79 | 55 |
| | | | mean | 74.5 | 55 |
| | | | median | 74.5 | 55 |
| | | | min | 70 | 55 |
| | | | mode | 71 | 55 |
| | | | mode | 72 | 55 |
| | | | mode | 77 | 55 |
| | | | mode | 78 | 55 |
| | | 2 | mode | 79 | 55 |
| | | | mode | 70 | 55 |
| | | | std | 3.937 | 0 |
| | | | 25% | 80.5 | 55 |
| | | | 50% | 81 | 55 |
| | | | 75% | 81.5 | 55 |
| | | | max | 82 | 55 |
| | | | mean | 81 | 55 |
| | | | median | 81 | 55 |
| | | | min | 80 | 55 |
| | | | mode | 80 | 55 |
| | | | mode | 81 | 55 |
| | | 44 | mode | 82 | 55 |
| | | | std | 1 | 0 |
| | | | 25% | 43 | 55 |
| | | | 50% | 43 | 55 |
| | | | 75% | 53 | 55 |
| | | | max | 56 | 55 |
| | | | mean | 48.077 | 55 |
| | | | median | 43 | 55 |
| | | | min | 43 | 55 |

| | | |
|------|-------|----|
| mode | 43 | 55 |
| std | 5.766 | 0 |

Example 2.3: Get the basic statistics, consider only unique values

Get the basic statistics for time series aggregation for all the numeric columns, consider only unique values. This example returns max, mean, min and std values.

```
>>> ocean_buoys_grpby.describe(distinct=True)
```

| TIMECODE_RANGE | GROUP BY TIME(CAL_YEARS(2)) | buoyid | func | temperature | salinity |
|--|-----------------------------|--------|------|-------------|----------|
| ('2014-01-01 00:00:00.000000-00:00', '2016-01-0... 2 | | 0 | max | 100 | 55 |
| | | | mean | 69.667 | 55 |
| | | | min | 10 | 55 |
| | | | std | 51.675 | None |
| | | 1 | max | 79 | 55 |
| | | | mean | 74.5 | 55 |
| | | | min | 70 | 55 |
| | | | std | 3.937 | None |
| | | 2 | max | 82 | 55 |
| | | | mean | 81 | 55 |
| | | | min | 80 | 55 |
| | | | std | 1 | None |
| | | 44 | max | 56 | 55 |
| | | | mean | 52.2 | 55 |
| | | | min | 43 | 55 |
| | | | std | 5.263 | None |

Example 2.4: Get the verbose statistics, select non-default percentiles

Get the verbose statistics for time series aggregation for all the numeric columns. In this example, you select non-default percentiles 33rd and 66th. This example returns max, mean, min, std, median, mode, 33rd, and 66th percentile.

```
>>> ocean_buoys_grpby.describe(verbose=True, percentiles=[0.33, 0.66])
```

| TIMECODE_RANGE | GROUP BY TIME(CAL_YEARS(2)) | buoyid | func | temperature | salinity |
|--|-----------------------------|--------|--------|-------------|----------|
| ('2014-01-01 00:00:00.000000-00:00', '2016-01-0... 2 | | 0 | 33% | 10 | 55 |
| | | | 66% | 97.22 | 55 |
| | | | max | 100 | 55 |
| | | | mean | 54.75 | 55 |
| | | | median | 54.5 | 55 |
| | | | min | 10 | 55 |
| | | | mode | 10 | 55 |
| | | | std | 51.674 | 0 |
| | | 1 | 33% | 71.65 | 55 |
| | | | 66% | 77.3 | 55 |
| | | | max | 79 | 55 |
| | | | mean | 74.5 | 55 |
| | | | median | 74.5 | 55 |
| | | | min | 70 | 55 |
| | | | mode | 70 | 55 |
| | | | mode | 71 | 55 |
| | | 2 | mode | 77 | 55 |
| | | | mode | 78 | 55 |
| | | | mode | 79 | 55 |
| | | | mode | 72 | 55 |
| | | | std | 3.937 | 0 |
| | | | 33% | 80.66 | 55 |
| | | | 66% | 81.32 | 55 |
| | | | max | 82 | 55 |
| | | 44 | mean | 81 | 55 |
| | | | median | 81 | 55 |
| | | | min | 80 | 55 |
| | | | mode | 80 | 55 |
| | | | mode | 81 | 55 |
| | | | mode | 82 | 55 |
| | | | std | 1 | 0 |
| | | | 33% | 43 | 55 |
| | | | 66% | 53 | 55 |
| | | | max | 56 | 55 |
| | | | mean | 48.077 | 55 |
| | | | median | 43 | 55 |
| | | | min | 43 | 55 |

| | | |
|------|-------|----|
| mode | 43 | 55 |
| std | 5.766 | 0 |

Filtering Rows and Columns

Filtering a teradataml DataFrame can be done using the [index\[\] Operator](#), the [loc\[\] Operator](#), the [iloc\[\] Operator](#), or the [filter\(\) Method](#).

filter() Method

The filter method subsets rows based on the index values of a DataFrame or columns of a DataFrame according to the labels in the specified index.

The filter is applied to the columns of the DataFrame when *axis* equals 'columns' (or 1). Otherwise, the filter is applied to the values of the DataFrame index when *axis* equals 'rows' (or 0).

There are three ways to filter using the filter method: item based, substring based and using regular expression. These ways are mutually exclusive in the sense that you must only use one of the following parameters: *items*, *like*, or *regex*.

item based Filtering

Example: Filtering with axis set to 'rows' or 0

When *axis* is 'rows' or 0, the items list specifies the values of the column(s) specified in the *index_label* of the DataFrame. Only rows where the values of the index column(s) in the *items* list are returned.

```
>>> df = DataFrame('admissions_train', index_label = ['programming'])
>>> df
```

| | id | masters | gpa | stats | admitted |
|-------------|----|---------|------|----------|----------|
| programming | | | | | |
| Advanced | 15 | yes | 4.00 | Advanced | 1 |
| Beginner | 34 | yes | 3.85 | Advanced | 0 |
| Novice | 13 | no | 4.00 | Advanced | 1 |
| Beginner | 38 | yes | 2.65 | Advanced | 1 |
| Novice | 5 | no | 3.44 | Novice | 0 |
| Beginner | 40 | yes | 3.95 | Novice | 0 |
| Novice | 7 | yes | 2.33 | Novice | 1 |
| Beginner | 22 | yes | 3.46 | Novice | 0 |
| Advanced | 26 | yes | 3.57 | Advanced | 1 |
| Advanced | 17 | no | 3.83 | Advanced | 1 |

```
>>> df.filter(items = ['Advanced', 'Novice'], axis = 0)
           id masters  gpa  stats admitted
programming
```

| | | | | | |
|----------|----|-----|------|----------|---|
| Advanced | 15 | yes | 4.00 | Advanced | 1 |
| Novice | 37 | no | 3.52 | Novice | 1 |
| Novice | 12 | no | 3.65 | Novice | 1 |
| Advanced | 17 | no | 3.83 | Advanced | 1 |
| Advanced | 11 | no | 3.13 | Advanced | 1 |
| Advanced | 26 | yes | 3.57 | Advanced | 1 |
| Novice | 5 | no | 3.44 | Novice | 0 |
| Novice | 24 | no | 1.87 | Advanced | 1 |
| Novice | 13 | no | 4.00 | Advanced | 1 |
| Novice | 7 | yes | 2.33 | Novice | 1 |

Example: Filtering with axis set to 'columns' or 1

When *axis* is 'columns' or 1, then column names provided in the *items* list are selected from the DataFrame.

```
>>> df.filter(items = ['programming', 'id', 'masters', 'gpa'])
      programming  id masters  gpa
Advanced      15    yes  4.00
Novice         7    yes  2.33
Beginner      22    yes  3.46
Advanced      17    no   3.83
Novice        13    no   4.00
Beginner      38    yes  2.65
Advanced      26    yes  3.57
Novice         5    no   3.44
Beginner      34    yes  3.85
Beginner      40    yes  3.95
```

Supported types for python literals in items

You can specify float, decimal.Decimal, str, bytes, datetime.time, datetime.date, and datetime.datetime python literal types in the items list.

substring based Filtering

Example 1: Filtering with axis set to 'columns' or 1

When *axis* is 'columns' or 1, then the like substring pattern is applied to the column names of the DataFrame.

Note:

The substring pattern is case sensitive.

```
>>> df.filter(like = 'st', axis = 'columns')
  masters  stats
0    yes  Advanced
1    yes   Novice
2    yes   Novice
3     no  Advanced
4     no  Advanced
5    yes  Advanced
6    yes  Advanced
7     no   Novice
8    yes  Advanced
9    yes   Novice
```

Example 2: Filtering with axis set to 'rows' or 0

When axis is 'rows' or 0, then the like substring pattern is applied to the values of the columns specified in the index_label of the DataFrame.

Note:

The index columns will be cast into VARCHAR columns in order to perform the match.

```
>>> df.filter(like = 'vice', axis = 'rows')
      id masters  gpa  stats admitted
programming
Novice    23    yes 3.59  Advanced      1
Novice    37    no 3.52   Novice      1
Novice    12    no 3.65   Novice      1
Novice    13    no 4.00  Advanced      1
Novice     5    no 3.44   Novice      0
Novice    24    no 1.87  Advanced      1
Novice    33    no 3.55   Novice      1
Novice     7    yes 2.33   Novice      1
Novice    30    yes 3.79  Advanced      0
Novice    36    no 3.00  Advanced      0
```

regex Filtering

Example 1: Filtering with axis set to 'columns' or 1

When axis is 'columns' or 1, then the regex pattern is applied to the column names of the DataFrame.

```
>>> df.filter(regex='[a-z].*s', axis = 'columns')
  masters  stats
```

```

0    yes  Advanced
1    yes   Novice
2    yes   Novice
3     no  Advanced
4     no  Advanced
5    yes  Advanced
6    yes  Advanced
7     no   Novice
8    yes  Advanced
9    yes   Novice

```

Example 2: Filtering with axis set to 'rows' or 0

When axis is 'rows' or 0, then the regex pattern is applied to the values of the columns specified in the index_label of the DataFrame.

Note:

The index columns will be cast into VARCHAR columns in order to perform the match.

```

>>> df.filter(regex = '^Beg.+ ', axis = 0)
           id masters   gpa   stats admitted
programming
Beginner   21     no  3.87   Novice         1
Beginner   22    yes  3.46   Novice         0
Beginner   39    yes  3.75  Advanced         0
Beginner   34    yes  3.85  Advanced         0
Beginner   38    yes  2.65  Advanced         1
Beginner    3     no  3.70   Novice         1
Beginner    1    yes  3.95  Beginner         0
Beginner   32    yes  3.46  Advanced         0
Beginner   40    yes  3.95   Novice         0
Beginner   29    yes  4.00   Novice         0

```

index[] Operator

Use the index operator ([]) to filter rows based on logical expressions given by Python literals and DataFrame columns.

Note:

Refer to [teradataml DataFrame Column](#) for more details about ColumnExpressions in teradataml.

Examples Prerequisite

```
>>> df = DataFrame('iris_test')
>>> df
```

| | sepal_length | sepal_width | petal_length | petal_width | species |
|-----|--------------|-------------|--------------|-------------|---------|
| id | | | | | |
| 120 | 6.0 | 2.2 | 5.0 | 1.5 | 3 |
| 30 | 4.7 | 3.2 | 1.6 | 0.2 | 1 |
| 70 | 5.6 | 2.5 | 3.9 | 1.1 | 2 |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | 1 |
| 60 | 5.2 | 2.7 | 3.9 | 1.4 | 2 |
| 10 | 4.9 | 3.1 | 1.5 | 0.1 | 1 |
| 65 | 5.6 | 2.9 | 3.6 | 1.3 | 2 |
| 20 | 5.1 | 3.8 | 1.5 | 0.3 | 1 |
| 15 | 5.8 | 4.0 | 1.2 | 0.2 | 1 |
| 80 | 5.7 | 2.6 | 3.5 | 1.0 | 2 |

```
>>> s_len = df.sepal_length
>>> p_len = df.petal_length
```

Example: Inequality operations

The operators supported are <, <=, >, >=.

```
>>> df[s_len > p_len]
```

| | sepal_length | sepal_width | petal_length | petal_width | species |
|-----|--------------|-------------|--------------|-------------|---------|
| id | | | | | |
| 120 | 6.0 | 2.2 | 5.0 | 1.5 | 3 |
| 30 | 4.7 | 3.2 | 1.6 | 0.2 | 1 |
| 70 | 5.6 | 2.5 | 3.9 | 1.1 | 2 |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | 1 |
| 60 | 5.2 | 2.7 | 3.9 | 1.4 | 2 |
| 10 | 4.9 | 3.1 | 1.5 | 0.1 | 1 |
| 65 | 5.6 | 2.9 | 3.6 | 1.3 | 2 |
| 20 | 5.1 | 3.8 | 1.5 | 0.3 | 1 |
| 15 | 5.8 | 4.0 | 1.2 | 0.2 | 1 |
| 80 | 5.7 | 2.6 | 3.5 | 1.0 | 2 |

```
>>> df[s_len < p_len + 1]
```

| | sepal_length | sepal_width | petal_length | petal_width | species |
|-----|--------------|-------------|--------------|-------------|---------|
| id | | | | | |
| 105 | 6.5 | 3.0 | 5.8 | 2.2 | 3 |
| 85 | 5.4 | 3.0 | 4.5 | 1.5 | 2 |
| 115 | 5.8 | 2.8 | 5.1 | 2.4 | 3 |

| | | | | | |
|-----|-----|-----|-----|-----|---|
| 150 | 5.9 | 3.0 | 5.1 | 1.8 | 3 |
| 135 | 6.1 | 2.6 | 5.6 | 1.4 | 3 |

Example: Equality operations

The operators supported are == and !=.

```
>>> df[s_len == p_len + 1]
   sepal_length  sepal_width  petal_length  petal_width  species
id
125          6.7          3.3          5.7          2.1         3
145          6.7          3.3          5.7          2.5         3
120          6.0          2.2          5.0          1.5         3
```

```
>>> df[s_len != p_len + 2]
   sepal_length  sepal_width  petal_length  petal_width  species
id
120          6.0          2.2          5.0          1.5         3
20           5.1          3.8          1.5          0.3         1
60           5.2          2.7          3.9          1.4         2
10           4.9          3.1          1.5          0.1         1
30           4.7          3.2          1.6          0.2         1
70           5.6          2.5          3.9          1.1         2
15           5.8          4.0          1.2          0.2         1
5            5.0          3.6          1.4          0.2         1
80           5.7          2.6          3.5          1.0         2
40           5.1          3.4          1.5          0.2         1
```

Example: Boolean operations

The operators supported are &, |, ~.

Note:

Do not use 'and' operator, instead use '&' operator. Using 'and' boolean operator instead of '&' will return incorrect results.

For example, if condition used is as length == 4.0 and width == 3.5, the first condition in previous case (length == 4.0), in 'and' condition is completely ignored and data satisfying only second condition (width == 3.5) will be returned.

Note:

Do not use 'or' operator, instead use '|' operator. Using 'or' boolean operator instead of '|' will return incorrect results.

For example, if condition used is as `length == 4.0 or width == 3.5`, the second condition in previous case (`width == 3.5`), in 'or' condition is completely ignored and data satisfying only first condition (`length == 4.0`) will be returned.

```
>>> df[~(s_len != p_len + 1)]
      sepal_length  sepal_width  petal_length  petal_width  species
id
125           6.7           3.3           5.7           2.1         3
145           6.7           3.3           5.7           2.5         3
120           6.0           2.2           5.0           1.5         3
```

```
>>> df[(s_len > 5.5) & (p_len < 5)]
      sepal_length  sepal_width  petal_length  petal_width  species
id
95             5.6           2.7           4.2           1.3         2
70             5.6           2.5           3.9           1.1         2
100            5.7           2.8           4.1           1.3         2
75             6.4           2.9           4.3           1.3         2
65             5.6           2.9           3.6           1.3         2
15             5.8           4.0           1.2           0.2         1
55             6.5           2.8           4.6           1.5         2
80             5.7           2.6           3.5           1.0         2
```

```
>>> df[(s_len >= 6.3) | (p_len == 4.2)]
      sepal_length  sepal_width  petal_length  petal_width  species
id
110            7.2           3.6           6.1           2.5         3
75             6.4           2.9           4.3           1.3         2
130            7.2           3.0           5.8           1.6         3
145            6.7           3.3           5.7           2.5         3
140            6.9           3.1           5.4           2.1         3
125            6.7           3.3           5.7           2.1         3
105            6.5           3.0           5.8           2.2         3
95             5.6           2.7           4.2           1.3         2
55             6.5           2.8           4.6           1.5         2
```

Note:

It is recommended to surround expressions in parentheses in order to avoid ambiguity in operator precedence, similar to pandas.

For example, in previous example, if the expression is taken without the parenthesis, an error is given.

```
df[2 >= s_len | 4 <= p_len]
```

```
-----
ArgumentError                                Traceback (most recent call last)
...
ArgumentError: SQL expression object or string expected, got object of
type <class 'int'> instead
...
TeradataMLException: [Teradata][teradataml](TDML_2021) Unable to retrieve
information for the DataFrame.
```

Supported Types and Operators

Python int, float, Decimal, str, and None literals can be used in filtering expressions.

All arithmetic expressions except floor division (//) and power (**) can be used.

All logical operators are supported.

get() Method

Use the get() function to retrieve required columns from a teradataml DataFrame.

The function takes a *key* representing a column name as an argument and returns a new DataFrame with the the appropriate columns. The *key* can be a single column name or a list of column names.

Note:

Multicolumn retrieval of the same column such as df.get(['col1', 'col1']) is not supported.

Examples Prerequisite

Assume a teradataml DataFrame is created based on the table "admissions_train".

```
>>> df = DataFrame("admissions_train")

>>> df
  masters    gpa    stats programming admitted
```

| id | | | | | |
|----|-----|------|----------|----------|---|
| 5 | no | 3.44 | novice | novice | 0 |
| 7 | yes | 2.33 | novice | novice | 1 |
| 22 | yes | 3.46 | novice | beginner | 0 |
| 17 | no | 3.83 | advanced | advanced | 1 |
| 13 | no | 4.00 | advanced | novice | 1 |
| 19 | yes | 1.98 | advanced | advanced | 0 |
| 36 | no | 3.00 | advanced | novice | 0 |
| 15 | yes | 4.00 | advanced | advanced | 1 |
| 34 | yes | 3.85 | advanced | beginner | 0 |
| 40 | yes | 3.95 | novice | beginner | 0 |

Example 1: Retrieve a single column

This example retrieves a single column "id" from the table "admissions_train". Column "id" is the index column.

```
>>>df.get("id")
Empty DataFrame
Columns: []
Index: [22, 34, 13, 19, 15, 38, 26, 5, 36, 17]
```

Example 2: Retrieve multiple columns using a list of columns names

Use a list of columns names for multicolumn retrieval.

```
>>>df.get(["id", "masters", "gpa"])
  masters  gpa
id
5       no  3.44
36      no  3.00
15     yes  4.00
17      no  3.83
13      no  4.00
40     yes  3.95
7       yes  2.33
22     yes  3.46
34     yes  3.85
19     yes  1.98
```

Example 3: Retrieve multiple columns using a list of list of columns names

Use a list of list of columns names for multicolumn retrieval.

```
>>> df.get(['id', 'masters', 'gpa'])
  masters  gpa
```

```

id
5      no  3.44
34     yes  3.85
13     no  4.00
40     yes  3.95
22     yes  3.46
19     yes  1.98
36     no  3.00
15     yes  4.00
7      yes  2.33
17     no  3.83

```

get_values() Method

Use the `get_values()` function to retrieve all values (only) present in a teradataml DataFrame.

The values are retrieved as per a `numpy.ndarray` representation of the teradataml DataFrame. This format is equivalent to the `get_values()` representation of a Pandas DataFrame.

An optional integer valued argument `num_rows` allows the user to specify the number of rows to retrieve values for from the teradataml DataFrame.

Note:

- Row and column indexing starts from 0, so the first column = index 0, second column = index 1, and so on.
 - When a Pandas DataFrame is saved to the database and then retrieved back as a teradataml DataFrame, the `get_values()` method on a Pandas DataFrame, and the corresponding teradataml DataFrames have the following type differences:
 - teradataml DataFrame `get_values()` retrieves 'bool' type Pandas DataFrame values ('True' or 'False') as BYTEINTS ('1' or '0');
 - teradataml DataFrame `get_values()` retrieves 'Timedelta' type Pandas DataFrame values as equivalent values in seconds.
-

Example Prerequisite

```

>>> df = DataFrame("admissions_train")

>>> df
  masters  gpa  stats programming admitted
id
5      no  3.44  novice      novice        0
7      yes  2.33  novice      novice        1
22     yes  3.46  novice  beginner        0

```

| | | | | | |
|----|-----|------|----------|----------|---|
| 17 | no | 3.83 | advanced | advanced | 1 |
| 13 | no | 4.00 | advanced | novice | 1 |
| 19 | yes | 1.98 | advanced | advanced | 0 |
| 36 | no | 3.00 | advanced | novice | 0 |
| 15 | yes | 4.00 | advanced | advanced | 1 |
| 34 | yes | 3.85 | advanced | beginner | 0 |
| 40 | yes | 3.95 | novice | beginner | 0 |

Example 1: Retrieves values present in a teradataml DataFrame

```
>>> vals = df.get_values()

>>> vals

array([[ 'yes', 4.0, 'advanced', 'advanced', 1],
       [ 'yes', 3.45, 'advanced', 'advanced', 0],
       [ 'yes', 3.5, 'advanced', 'beginner', 1],
       [ 'yes', 4.0, 'novice', 'beginner', 0],
       . . .
       [ 'no', 3.68, 'novice', 'beginner', 1],
       [ 'yes', 3.5, 'beginner', 'advanced', 1],
       [ 'yes', 3.79, 'advanced', 'novice', 0],
       [ 'no', 3.0, 'advanced', 'novice', 0],
       [ 'yes', 1.98, 'advanced', 'advanced', 0]], dtype=object)
```

Example 2: Retrieve values for a given number of rows from a teradataml DataFrame

```
>>> vals = df1.get_values(num_rows = 3)
>>> vals

array([[ 'yes', 4.0, 'advanced', 'advanced', 1],
       [ 'yes', 3.45, 'advanced', 'advanced', 0],
       [ 'yes', 3.5, 'advanced', 'beginner', 1]], dtype=object)
```

Example 3: Access specific values from the entire set received

```
# Retrieve all values from an entire row (for example, the first row):
>>> vals[0]
array([ 'yes', 4.0, 'advanced', 'advanced', 1], dtype=object)

# Specify a range to retrieve values from a subset of rows (For example, first
3 rows):
>>> vals[0:3]
```

```
array([[ 'yes', 4.0, 'advanced', 'advanced', 1],
       [ 'yes', 3.45, 'advanced', 'advanced', 0],
       [ 'yes', 3.5, 'advanced', 'beginner', 1]], dtype=object)

# Retrieve all values from an entire column (For example, the first column):
>>> vals[:, 0]
array([ 'yes', 'yes', 'yes', 'yes', 'yes', 'no', 'yes', 'yes', 'yes',
       'yes', 'no', 'no', 'yes', 'yes', 'no', 'yes', 'no', 'yes', 'no',
       'no', 'no', 'no', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes',
       'yes', 'yes', 'no', 'no', 'yes', 'no', 'no', 'yes', 'yes',
       'no', 'yes'], dtype=object)

# Retrieve a single value from a given row and column (For example, 3rd row,
and 2nd column):
>>> vals[2,1]
3.5
```

head and tail

The examples in this section use the DataFrame "df" created using the following command.

Examples Prerequisite

Assume a teradataml DataFrame "df" is created from a table "admissions_train", using command:

```
>>> df = DataFrame("admissions_train")
```

head() Method

Use the head() method to print the first n rows of a teradataml DataFrame.

The method takes the optional argument n number of rows to print. Default value is dependent on the value of the teradataml DataFrame display option *max_rows*, which is by default 10.

Note:

The DataFrame is sorted on the index column or the first column if there is no index column. The column type must support sorting.

Unsupported types include: 'BLOB', 'CLOB', 'ARRAY', 'VARARRAY'.

Example 1: Print default number of rows

This example prints the default 10 rows of "admissions_train":

```
>>> df.head()
   masters  gpa  stats programming admitted
id
3        no  3.70  novice    beginner         1
5        no  3.44  novice      novice         0
6       yes  3.50  beginner  advanced         1
7       yes  2.33  novice      novice         1
9        no  3.82  advanced  advanced         1
10       no  3.71  advanced  advanced         1
8        no  3.60  beginner  advanced         1
4       yes  3.50  beginner      novice         1
2       yes  3.76  beginner  beginner         0
1       yes  3.95  beginner  beginner         0
```

Example 2: Print n rows

The following example prints 5 rows of "admissions_train":

```
>>> df.head(5)
   masters  gpa  stats programming admitted
id
3        no  3.70  novice    beginner         1
5        no  3.44  novice      novice         0
4       yes  3.50  beginner      novice         1
2       yes  3.76  beginner  beginner         0
1       yes  3.95  beginner  beginner         0
```

The following example prints 15 rows of "admissions_train":

```
>>> df.head(15)
   masters  gpa  stats programming admitted
id
3        no  3.70  novice    beginner         1
5        no  3.44  novice      novice         0
6       yes  3.50  beginner  advanced         1
7       yes  2.33  novice      novice         1
9        no  3.82  advanced  advanced         1
10       no  3.71  advanced  advanced         1
11       no  3.13  advanced  advanced         1
12       no  3.65  novice      novice         1
13       no  4.00  advanced      novice         1
14      yes  3.45  advanced  advanced         0
15      yes  4.00  advanced  advanced         1
8        no  3.60  beginner  advanced         1
4       yes  3.50  beginner      novice         1
```

| | | | | | |
|---|-----|------|----------|----------|---|
| 2 | yes | 3.76 | beginner | beginner | 0 |
| 1 | yes | 3.95 | beginner | beginner | 0 |

tail() Method

Use the `tail()` method to print the last n rows of a DataFrame.

The method takes the optional argument n number of rows to print. Default value is dependent on the value of the teradataml DataFrame display option `max_rows`, which is by default 10.

Note:

The DataFrame is sorted on the index column or the first column if there is no index column. The column type must support sorting.

Unsupported types include: 'BLOB', 'CLOB', 'ARRAY', 'VARRAY'.

Example 1: Print default number of rows

The following example prints the default last 10 rows of "admissions_train" sorted by id.

```
>>>df.tail()
  masters  gpa  stats programming admitted
id
38    yes  2.65  advanced    beginner      1
36     no  3.00  advanced     novice      0
35     no  3.68   novice    beginner      1
34    yes  3.85  advanced    beginner      0
32    yes  3.46  advanced    beginner      0
31    yes  3.50  advanced    beginner      1
33     no  3.55   novice     novice      1
37     no  3.52   novice     novice      1
39    yes  3.75  advanced    beginner      0
40    yes  3.95   novice    beginner      0
```

Example 2: Print n rows

The following example prints the last 3 rows of "admissions_train":

```
>>>df.tail(3)
  masters  gpa  stats programming admitted
id
38    yes  2.65  advanced    beginner      1
39    yes  3.75  advanced    beginner      0
40    yes  3.95   novice    beginner      0
```


The following example prints the last 15 rows of "admissions_train":

```
>>>df.tail(15)
   masters  gpa  stats programming admitted
id
38    yes  2.65  advanced    beginner        1
36     no  3.00  advanced      novice        0
35     no  3.68   novice    beginner        1
34    yes  3.85  advanced    beginner        0
32    yes  3.46  advanced    beginner        0
31    yes  3.50  advanced    beginner        1
30    yes  3.79  advanced      novice        0
29    yes  4.00   novice    beginner        0
28     no  3.93  advanced  advanced        1
27    yes  3.96  advanced  advanced        0
26    yes  3.57  advanced  advanced        1
33     no  3.55   novice      novice        1
37     no  3.52   novice      novice        1
39    yes  3.75  advanced    beginner        0
40    yes  3.95   novice    beginner        0
```

join() Method

Use the `join()` function to join two teradataml DataFrame objects together. The operation is performed by carrying out a database-style join using teradataml specified as keys.

The required argument *other* specifies right teradataml DataFrame on which join is to be performed.

The optional *how* argument specifies the type of the join performed.

Supported join operations are:

- `left`: Default value for this parameter. Returns all matching rows, plus non-matching rows from the left teradataml DataFrame;
- `inner`: Returns only matching rows, non-matching rows are eliminated;
- `right`: Returns all matching rows, plus non-matching rows from the right teradataml DataFrame;
- `full`: Returns all rows from both teradataml DataFrames, including non matching rows;
- `cross`: Returns all rows from both tables where each row from the first table is joined with each row from the second table. The result of the join is a Cartesian cross product.

Valid forms of on conditions

The optional *on* argument specifies the list of conditions that indicate the columns to be join keys.

Supported join operators are `=`, `==`, `<`, `<=`, `>`, `>=`, `<>` and `!=`.

Note:

= and <> operators are not supported when using DataFrame columns as operands.

The on conditions can take the following forms:

- String comparisons, in the form of "col1 <= col2", where col1 is the column of left dataframe df1 and col2 is the column of right dataframe df2. For example:
 - ["a","b"] indicates df1.a = df2.a and df1.b = df2.b
 - ["a = b", "c == d"] indicates df1.a = df2.b and df1.c = df2.d
 - ["a <= b", "c > d"] indicates df1.a <= df2.b and df1.c > df2.d
 - ["a < b", "c >= d"] indicates df1.a < df2.b and df1.c >= df2.d
 - ["a <> b"] indicates df1.a != df2.b. Same is the case for ["a != b"]
- Column comparisons, in the form of df1.col1 <= df1.col2, where col1 is the column of left dataframe df1 and col2 is the column of right dataframe df2. For example:
 - [df1.a == df2.a, df1.b == df2.b] indicates df1.a = df2.a and df1.b = df2.b
 - [df1.a == df2.b, df1.c == df2.d] indicates df1.a = df2.b and df1.c = df2.d
 - [df1.a <= df2.b and df1.c > df2.d] indicates df1.a <= df2.b and df1.c > df2.d
 - [df1.a < df2.b and df1.c >= df2.d] indicates df1.a < df2.b and df1.c >= df2.d
 - df1.a != df2.b indicates df1.a != df2.b
- The combination of both string comparisons and comparisons as column expressions. For example:
 - ["a", df1.b == df2.b] indicates df1.a = df2.a and df1.b = df2.b
 - [df1.a <= df2.b, "c > d"] indicates df1.a <= df2.b and df1.c > df2.d

The optional arguments *lsuffix* and *rsuffix* specify the suffix to be added to the left and right table columns, respectively.

The optional arguments *lprefix* and *rprefix* specify the prefix to be added to the left and right table columns, respectively.

Note:

Teradata recommends using the *lprefix* and *rprefix* arguments, as behavior of *lsuffix* and *rsuffix* arguments will be changed.

Note:

- When multiple join conditions are given, they are joined using AND boolean operator. Other boolean operators are not supported.
- Nesting of join on conditions in column expressions using & and | is not supported. For example, on = [(df1.a == df1.b) & (df1.c == df1.d)] is unsupported nested join on condition. You can use [df1.a == df1.b, df1.c == df1.d] instead.
- For a cross join operation, the 'on' argument is ignored.

Refer to [teradataml DataFrame Column](#) for more information about ColumnExpressions in teradataml.

Example Prerequisite

Set up teradataml DataFrames for join operation(s).

```
>>> from datetime import datetime, timedelta

>>> dob = datetime.strptime('31101991', '%d%m%Y').date()

>>> pdf1 = pd.DataFrame(data={'col1': [1, 2,3],
                              'col2': ['teradata','analytics','platform'],
                              'col3': [1.3, 2.3, 3.3],
                              'col5': ['a','b','c'],
                              'col 6': [dob, dob + timedelta(1), dob
+ timedelta(2)]},
                        "'col8'": [3,4,5]})

>>> pdf2 = pd.DataFrame(data={'col1': [1, 2, 3],
                              'col4': ['teradata', 'analytics', 'are you'],
                              'col3': [1.3, 2.3, 4.3],
                              'col7': ['a','b','d'],
                              'col 6': [dob, dob + timedelta(1), dob
+ timedelta(3)]},
                        "'col8'": [3, 4, 5]})

>>> copy_to_sql(pdf1, "t1", primary_index="col1")

>>> copy_to_sql(pdf2, "t2", primary_index="col1")

>>> df1 = DataFrame("t1")

>>> df2 = DataFrame("t2")
```

Display the DataFrames.

```
>>> df1
      'col8'      col 6      col2  col3 col5
col1
2          4  1991-11-01  analytics  2.3    b
1          3  1991-10-31   teradata  1.3    a
3          5  1991-11-02   platform  3.3    c

>>> df2
```

| | 'col8' | col 6 | col3 | col4 | col7 |
|------|--------|------------|------|-----------|------|
| col1 | | | | | |
| 2 | 4 | 1991-11-01 | 2.3 | analytics | b |
| 1 | 3 | 1991-10-31 | 1.3 | teradata | a |
| 3 | 5 | 1991-11-03 | 4.3 | are you | d |

Example 1: Use 'on' and 'how' parameters to join the DataFrames

Specify a join condition using the 'on' parameter and the type of join using the 'how' parameter.

Note:

If teradataml DataFrames have columns with the same names, suffixes are required.

```
# Using string expressions for on conditions.
>>> df3 = df1.join(other = df2, on = ["col3", "col5=col7"], how = "left", lsuffix = "t1", rsuffix = "t2")
>>> df3
```

| | t2_'col8' | t1_col1 | col5 | t2_col 6 | col2 | t1_'col8' | col7 | t1_col3 | t2_col3 | t1_col 6 | t2_col1 | col4 |
|---|-----------|---------|------|------------|-----------|-----------|------|---------|---------|------------|---------|-----------|
| 0 | None | 3 | c | None | platform | 5 | None | 3.3 | NaN | 1991-11-02 | None | None |
| 1 | 4 | 2 | b | 1991-11-01 | analytics | 4 | b | 2.3 | 2.3 | 1991-11-01 | 2 | analytics |
| 2 | 3 | 1 | a | 1991-10-31 | teradata | 3 | a | 1.3 | 1.3 | 1991-10-31 | 1 | teradata |

```
# Using dataframe column expressions instead of string expressions for on conditions.
>>> df3 = df1.join(other = df2, on = [df1.col3 == df2.col3, df1.col5 == df2.col7], how = "left", lsuffix = "t1",
rsuffix = "t2")
>>> df3
```

| | t2_'col8' | t1_col1 | col5 | t2_col 6 | col2 | t1_'col8' | col7 | t1_col3 | t2_col3 | t1_col 6 | t2_col1 | col4 |
|---|-----------|---------|------|------------|-----------|-----------|------|---------|---------|------------|---------|-----------|
| 0 | None | 3 | c | None | platform | 5 | None | 3.3 | NaN | 1991-11-02 | None | None |
| 1 | 4 | 2 | b | 1991-11-01 | analytics | 4 | b | 2.3 | 2.3 | 1991-11-01 | 2 | analytics |
| 2 | 3 | 1 | a | 1991-10-31 | teradata | 3 | a | 1.3 | 1.3 | 1991-10-31 | 1 | teradata |

Example 2: Specify a required type of join using the 'how' parameter

```
# Using string expressions for on conditions.
>>> df4 = df1.join(other = df2, on = ["col3", "'col8'"], how = "right", lsuffix = "t1", rsuffix = "t2")
>>> df4
```

| | t2_'col8' | t1_col1 | col5 | t2_col 6 | col2 | t1_'col8' | col7 | t1_col3 | t2_col3 | t1_col 6 | t2_col1 | col4 |
|---|-----------|---------|------|------------|-----------|-----------|------|---------|---------|------------|---------|-----------|
| 0 | 5 | None | None | 1991-11-03 | None | None | d | NaN | 4.3 | None | 3 | are you |
| 1 | 4 | 2 | b | 1991-11-01 | analytics | 4 | b | 2.3 | 2.3 | 1991-11-01 | 2 | analytics |
| 2 | 3 | 1 | a | 1991-10-31 | teradata | 3 | a | 1.3 | 1.3 | 1991-10-31 | 1 | teradata |

```
# Using dataframe column expressions instead of string expressions for some of on conditions.
>>> df3 = df1.join(other = df2, on = [df1.col3 == df2.col3, "'col8'"], how = "right", lsuffix = "t1", rsuffix = "t2")
>>> df3
```

| | t2_'col8' | t1_col1 | col5 | t2_col 6 | col2 | t1_'col8' | col7 | t1_col3 | t2_col3 | t1_col 6 | t2_col1 | col4 |
|---|-----------|---------|------|------------|-----------|-----------|------|---------|---------|------------|---------|-----------|
| 0 | 5 | None | None | 1991-11-03 | None | None | d | NaN | 4.3 | None | 3 | are you |
| 1 | 4 | 2 | b | 1991-11-01 | analytics | 4 | b | 2.3 | 2.3 | 1991-11-01 | 2 | analytics |
| 2 | 3 | 1 | a | 1991-10-31 | teradata | 3 | a | 1.3 | 1.3 | 1991-10-31 | 1 | teradata |

Example 3: Use the default value for 'how' parameter

When the 'how' parameter is not provided, a left join is used by default.

```
>>> df5 = df1.join(other = df2, on = 'col3', lsuffix = "t1", rsuffix = "t2")
```

```
>>> df5
```

| | t2_'col8' | t1_col1 | col5 | t2_col 6 | col2 | t1_'col8' | col7 | t1_col3 | t2_col3 | t1_col 6 | t2_col1 | col4 |
|---|-----------|---------|------|------------|-----------|-----------|------|---------|---------|------------|---------|-----------|
| 0 | None | 3 | c | None | platform | 5 | None | 3.3 | NaN | 1991-11-02 | None | None |
| 1 | 4 | 2 | b | 1991-11-01 | analytics | 4 | b | 2.3 | 2.3 | 1991-11-01 | 2 | analytics |
| 2 | 3 | 1 | a | 1991-10-31 | teradata | 3 | a | 1.3 | 1.3 | 1991-10-31 | 1 | teradata |

Example 4: How cross join works

```
# cross join "admissions_train" with "admissions_train".
>>> df1 = DataFrame("admissions_train").head(3).sort("id")
>>> print(df1)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 1 | yes | 3.95 | Beginner | Beginner | 0 |
| 2 | yes | 3.76 | Beginner | Beginner | 0 |
| 3 | no | 3.70 | Novice | Beginner | 1 |

```
>>> df2 = DataFrame("admissions_train").head(3).sort("id")
>>> print(df2)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 1 | yes | 3.95 | Beginner | Beginner | 0 |
| 2 | yes | 3.76 | Beginner | Beginner | 0 |
| 3 | no | 3.70 | Novice | Beginner | 1 |

```
>>> df3 = df1.join(other=df2, how="cross", lsuffix="l", rsuffix="r")
>>> df3.set_index("l_id").sort("l_id")
```

| | r_id | l_masters | r_masters | l_gpa | r_gpa | l_stats | r_stats | l_programming | r_programming | l_admitted | r_admitted |
|------|------|-----------|-----------|-------|-------|----------|----------|---------------|---------------|------------|------------|
| l_id | | | | | | | | | | | |
| 1 | 3 | yes | no | 3.95 | 3.70 | Beginner | Novice | Beginner | Beginner | 0 | 1 |
| 1 | 1 | yes | yes | 3.95 | 3.95 | Beginner | Beginner | Beginner | Beginner | 0 | 0 |
| 1 | 2 | yes | yes | 3.95 | 3.76 | Beginner | Beginner | Beginner | Beginner | 0 | 0 |
| 2 | 1 | yes | yes | 3.76 | 3.95 | Beginner | Beginner | Beginner | Beginner | 0 | 0 |
| 2 | 2 | yes | yes | 3.76 | 3.76 | Beginner | Beginner | Beginner | Beginner | 0 | 0 |
| 2 | 3 | yes | no | 3.76 | 3.70 | Beginner | Novice | Beginner | Beginner | 0 | 1 |
| 3 | 1 | no | yes | 3.70 | 3.95 | Novice | Beginner | Beginner | Beginner | 1 | 0 |
| 3 | 2 | no | yes | 3.70 | 3.76 | Novice | Beginner | Beginner | Beginner | 1 | 0 |
| 3 | 3 | no | no | 3.70 | 3.70 | Novice | Novice | Beginner | Beginner | 1 | 1 |

loc and iloc**iloc[] Operator**

Use the `iloc[]` operator to access a group of rows and columns by integer values.

The operator takes a single integer, a list of integers, and a slice with integers as valid inputs. It also takes a list of booleans for column access. The list must include a boolean value for each column.

Note:

For integer indexing on row access, the integer index values are applied to a sorted teradataml DataFrame on the index column or the first column if there is no index column.

Examples Prerequisite

Assume a teradataml DataFrame "df" is created from a Vantage table "sales" and sorted using commands:

```
>>> df = DataFrame('sales')

>>> df.sort("accounts")
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Orange Inc | 210.0 | None | None | 250 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2018-10-15 |
| Yellow Inc | 90.0 | None | None | None | 2017-04-01 |

Example 1 : Retrieve a row using a single integer value

The following example retrieves a row using a single integer value.

```
>>> df.iloc[1]
```

| | Feb | Jan | Mar | Apr | datetime |
|----------|------|-----|-----|-----|------------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |

Example 2: Retrieve using a list of integers

Note:

Using "[[]]" to include a list of integers.

```
>>> df.iloc[[1, 2]]
```

| | Feb | Jan | Mar | Apr | datetime |
|-----------|-------|-----|-----|-----|------------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50 | 95 | 101 | 2017-04-01 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |

Example 3: Use a single integer for both row and column

```
>>> df.iloc[5, 0]
Empty DataFrame
Columns: []
Index: [Yellow Inc]
```

```
>>> df.iloc[(5, 1)]
      Feb
0  90.0
```

Example 4: Use a slice for row and a single integer for column access

Note:

The stop for the slice is excluded.

```
>>> df.iloc[2:5, 2]
      Jan
0  None
1  150
2  150
```

Example 5: Use a slice for row and column access

Note:

The stop for the slice is excluded.

```
>>> df.iloc[2:5, 0:5]
      Mar  Jan  Feb  Apr
accounts
Orange Inc  None  None  210.0  250
Red Inc     140  150  200.0  None
Jones LLC   140  150  200.0  180
```

Example 6: Use an empty slice for row and column access

```
>>> df.iloc[:, :]
      Feb  Jan  Mar  datetime  Apr
accounts
Jones LLC  200.0  150  140  2017-04-01  180
Blue Inc   90.0   50   95  2017-04-01  101
Yellow Inc  90.0  None  None  2017-04-01  None
Orange Inc 210.0  None  None  2017-04-01  250
Alpha Co   210.0  200  215  2017-04-01  250
Red Inc    200.0  150  140  2017-04-01  None
```

Example 7: Use a list of integers and boolean array for column access

```
>>> df.iloc[[0, 2, 3, 4], [True, True, False, False, True, True]]
      datetime  Apr  Feb
accounts
Jones LLC    2017-04-01    180  200.0
Orange Inc   2017-04-01    250  210.0
Alpha Co     2017-04-01    250  210.0
Red Inc      2017-04-01    None  200.0
```

loc[] Operator

Use the `loc[]` operator to access a group of rows and columns by labels.

The operator takes a single label, a list of column or index labels, and a slice with labels as valid inputs. It also takes a conditional expression for row access and a list of booleans for column access. The list must include a boolean value for each column.

Examples Prerequisite

Assume a teradataml DataFrame "df" is created from a Vantage table "sales", using command:

```
>>> df = DataFrame('sales')

>>> df
      Feb  Jan  Mar  Apr  datetime
accounts
Blue Inc   90.0   50   95  101  2017-04-01
Alpha Co  210.0  200  215  250  2017-04-01
Jones LLC  200.0  150  140  180  2017-04-01
Yellow Inc  90.0  None  None  None  2017-04-01
Orange Inc 210.0  None  None  250  2017-04-01
Red Inc    200.0  150  140  None  2017-04-01
```

Example 1: Retrieve a row using a single index label

This example retrieves a row using a single index label "Blue Inc":

```
>>> df.loc['Blue Inc']
      Feb Jan Mar  Apr  datetime
accounts
Blue Inc  90.0  50  95  101  2017-04-01
```


Example 2: Retrieve multiple rows using a list of labels

This example uses a list of labels to retrieve the rows for "Blue Inc" and "Jones LLC":

```
>>> df.loc[['Blue Inc', 'Jones LLC']]
          Feb  Jan  Mar  Apr  datetime
accounts
Blue Inc   90.0   50   95  101  2017-04-01
Jones LLC  200.0  150  140  180  2017-04-01
```

Example 3: Retrieve using a single index label and a single index column label

This example uses a single index label and a single column label (index column label) for row and column access:

```
>>> df.loc['Yellow Inc', 'accounts']
Empty DataFrame
Columns: []
Index: [Yellow Inc]
```

Example 4: Retrieve using a single index label and a single non-index column label

This example uses a single index label "Yellow Inc" and a single column label "Feb" (non-index column label) for row and column access:

```
>>> df.loc['Yellow Inc', 'Feb']
Feb
0    90.0
```

Example 5: Retrieve using a slice with labels for row access and single label for column access

This example uses a slice with labels for row access and single label for column access:

```
>>> df.loc['Jones LLC':'Red Inc', 'accounts']
Empty DataFrame
Columns: []
Index: [Orange Inc, Jones LLC, Red Inc]
```

Note:

Both the start and stop of the slice are included.

Example 6: Retrieve using a slice with labels for row access and for column access

This example uses a slice with labels for row access and for column access:

```
>>> df.loc['Jones LLC':'Red Inc', 'accounts':'Apr']
```

| | Mar | Jan | Feb | Apr |
|------------|------|------|-------|------|
| accounts | | | | |
| Orange Inc | None | None | 210.0 | 250 |
| Red Inc | 140 | 150 | 200.0 | None |
| Jones LLC | 140 | 150 | 200.0 | 180 |

Example 7: Retrieve using an empty slice for row access and for column access

This example uses an empty slice for row access and for column access:

```
>>> df.loc[:, :]
```

| | Feb | Jan | Mar | datetime | Apr |
|------------|-------|------|------|------------|------|
| accounts | | | | | |
| Jones LLC | 200.0 | 150 | 140 | 2017-04-01 | 180 |
| Blue Inc | 90.0 | 50 | 95 | 2017-04-01 | 101 |
| Yellow Inc | 90.0 | None | None | 2017-04-01 | None |
| Orange Inc | 210.0 | None | None | 2017-04-01 | 250 |
| Alpha Co | 210.0 | 200 | 215 | 2017-04-01 | 250 |
| Red Inc | 200.0 | 150 | 140 | 2017-04-01 | None |

Example 8: Retrieve rows using a conditional expression

This example uses a conditional expression to retrieve rows where the value for "Feb" is greater than 90:

```
>>> df.loc[df['Feb'] > 90]
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Jones LLC | 200.0 | 150 | 140 | 180 | 2017-04-01 |
| Red Inc | 200.0 | 150 | 140 | None | 2017-04-01 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 2017-04-01 |
| Orange Inc | 210.0 | None | None | 250 | 2017-04-01 |

Example 9: Retrieve using a conditional expression for row access with multiple column labels for column access

This examples uses a conditional expression for row access with multiple column labels for column access:

```
>>> df.loc[df['accounts'] == 'Jones LLC', ['accounts', 'Jan', 'Feb']]
           Jan    Feb
accounts
Jones LLC  150  200.0
```

Example 10: Retrieve using a conditional expression for row access and boolean array for column access

This example uses a conditional expression for row access and boolean array for column access:

```
>>> df.loc[df['Feb'] > 90, [True, True, False, False, True, True]]
           datetime    Apr    Feb
accounts
Jones LLC  2017-04-01   180  200.0
Orange Inc 2017-04-01   250  210.0
Alpha Co   2017-04-01   250  210.0
Red Inc    2017-04-01  None  200.0
```

merge() Method

Use the `merge()` function to merge two teradataml DataFrame objects together. The merge is done by performing a database-style join operation by columns or indexes.

Supported merge operations are:

- `inner`: Returns only matching rows, non-matching rows are eliminated;
- `left`: Returns all matching rows, plus non-matching rows from the left teradataml DataFrame;
- `right`: Returns all matching rows, plus non-matching rows from the right teradataml DataFrame;
- `full`: Returns all rows from both teradataml DataFrames, including non matching rows.

Note:

- When multiple merge conditions are given, they are joined using AND boolean operator. Other boolean operators are not supported.
- Nesting of merge on conditions in column expressions using `&` and `|` is not supported.

For example: `on = [(df1.a == df1.b) & (df1.c == df1.d)]`. You can use `[df1.a == df1.b, df1.c == df1.d]` instead.

The valid form of *on* conditions

The *on* parameter specifies the list of conditions that indicate the columns to be join keys. Supported join operators are `=`, `==`, `<`, `<=`, `>`, `>=`, `<>` and `!=`.

`=` and `<>` operators are not supported when using DataFrame columns as operands.

The on conditions can take the following forms:

- String comparisons, in the form of `col1 <= col2`, where `col1` is the column of left dataframe `df1` and `col2` is the column of right dataframe `df2`. For example:
 1. `["a", "b"]` indicates `df1.a = df2.a` and `df1.b = df2.b`.
 2. `["a = b", "c == d"]` indicates `df1.a = df2.b` and `df1.c = df2.d`.
 3. `["a <= b", "c > d"]` indicates `df1.a <= df2.b` and `df1.c > df2.d`.
 4. `["a < b", "c >= d"]` indicates `df1.a < df2.b` and `df1.c >= df2.d`.
 5. `["a <> b"]` indicates `df1.a != df2.b`. Same is the case for `["a != b"]`.
- Column comparisons, in the form of `df1.col1 <= df1.col2`, where `col1` is the column of left dataframe `df1` and `col2` is the column of right dataframe `df2`. For example:
 1. `[df1.a == df2.a, df1.b == df2.b]` indicates `df1.a = df2.a` and `df1.b = df2.b`.
 2. `[df1.a == df2.b, df1.c == df2.d]` indicates `df1.a = df2.b` and `df1.c = df2.d`.
 3. `[df1.a <= df2.b and df1.c > df2.d]` indicates `df1.a <= df2.b` and `df1.c > df2.d`.
 4. `[df1.a < df2.b and df1.c >= df2.d]` indicates `df1.a < df2.b` and `df1.c >= df2.d`.
 5. `df1.a != df2.b` indicates `df1.a != df2.b`.
- The combination of both string comparisons and comparisons as column expressions. For example:
 1. `["a", df1.b == df2.b]` indicates `df1.a = df2.a` and `df1.b = df2.b`.
 2. `[df1.a <= df2.b, "c > d"]` indicates `df1.a <= df2.b` and `df1.c > df2.d`.

Refer to [teradataml DataFrame Column](#) for more information about ColumnExpressions in teradataml.

Example Prerequisite

Set up teradataml DataFrames for merge.

```
>>> from datetime import datetime, timedelta

>>> dob = datetime.strptime('31101991', '%d%m%Y').date()

>>> df1 = pd.DataFrame(data={'col1': [1, 2, 3],
                             'col2': ['teradata', 'analytics', 'platform'],
                             'col3': [1.3, 2.3, 3.3],
                             'col5': ['a', 'b', 'c'],
                             'col 6': [dob, dob + timedelta(1), dob
+ timedelta(2)],
                             "'col8'": [3, 4, 5]})

>>> df2 = pd.DataFrame(data={'col1': [1, 2, 3],
                             'col4': ['teradata', 'analytics', 'are you'],
                             'col3': [1.3, 2.3, 4.3],
                             'col7': ['a', 'b', 'd'],
```

```

        'col 6': [dob, dob + timedelta(1), dob
+ timedelta(3)],
        "'col8'": [3, 4, 5]})

```

Persist the Pandas DataFrames in Vantage:

```

>>> copy_to_sql(df1, "t1", primary_index="col1")

>>> copy_to_sql(df2, "t2", primary_index="col1")

>>> df1 = DataFrame("table1")

>>> df2 = DataFrame("table2")

```

Display the DataFrames.

```

>>> df1
      'col8'      col 6      col2  col3  col5
col1
2          4  1991-11-01  analytics    2.3    b
1          3  1991-10-31   teradata    1.3    a
3          5  1991-11-02   platform    3.3    c

>>> df2
      'col8'      col 6  col3      col4  col7
col1
2          4  1991-11-01    2.3  analytics    b
1          3  1991-10-31    1.3   teradata    a
3          5  1991-11-03    4.3    are you    d

```

Example 1: Specify merge conditions as string using 'on' argument as well as DataFrame indexes as merge keys

```

>>> df1.merge(right = df2, how = "left", on = ["col3", "col2=col4"], use_index = True, lsuffix = "t1", rsuffix
= "t2")

```

| | t2_col1 | col5 | t2_col 6 | t1_col1 | t2_'col8' | t1_col3 | col4 | t2_col3 | col7 | col2 | t1_col 6 | t1_'col8' |
|---|---------|------|------------|---------|-----------|---------|-----------|---------|------|-----------|------------|-----------|
| 0 | 2 | b | 1991-11-01 | 2 | 4 | 2.3 | analytics | 2.3 | b | analytics | 1991-11-01 | 4 |
| 1 | 1 | a | 1991-10-31 | 1 | 3 | 1.3 | teradata | 1.3 | a | teradata | 1991-10-31 | 3 |
| 2 | None | c | None | 3 | None | 3.3 | None | NaN | None | platform | 1991-11-02 | 5 |

Example 2: Specify 'on' conditions as ColumnExpression and DataFrame indexes as merge keys

```

>>> df1.merge(right = df2, how = "left", on = [df1.col1, df1.col3], use_index = True, lsuffix = "t1", rsuffix
= "t2")

```

| | t1_col1 | t2_col1 | col2 | t1_col3 | t2_col3 | col5 | t1_col 6 | t2_col 6 | t1_'col8' | t2_'col8' | col4 | col7 |
|---|---------|---------|-----------|---------|---------|------|------------|------------|-----------|-----------|-----------|------|
| 0 | 2 | 2.0 | analytics | 2.3 | 2.3 | b | 1991-06-23 | 1991-06-23 | 4 | 4.0 | analytics | b |
| 1 | 1 | 1.0 | teradata | 1.3 | 1.3 | a | 1991-06-22 | 1991-06-22 | 3 | 3.0 | teradata | a |
| 2 | 3 | NaN | platform | 3.3 | NaN | c | 1991-06-24 | None | 5 | NaN | None | None |

Example 3: Specify left_on, right_on conditions along with DataFrame indexes as merge keys

```
>>> df1.merge(right = df2, how = "right", left_on = "col2", right_on = "col4", use_index = True, lsuffix = "t1",
rsuffix = "t2")
   t1_col1 t2_col1   col2 t1_col3 t2_col3 col5 t1_col 6 t2_col 6 t1_'col8' t2_'col8'   col4 col7
0         2         2  analytics  2.3    2.3    b  1991-11-01  1991-11-01         4         4  analytics  b
1         1         1  teradata  1.3    1.3    a  1991-10-31  1991-10-31         3         3  teradata  a
2      None         3      None   NaN    4.3  None      None  1991-11-03      None         5  are you  d
```

Example 4: DataFrames to be merged do not contain common columns

If DataFrames to be merged do not contain common columns, lsuffix and rsuffix are not required.

```
>>> new_df1 = df1.select(['col2', 'col5'])
```

```
>>> new_df2 = df2.select(['col4', 'col7'])
```

```
>>> new_df1
   col5   col2
0     b  analytics
1     a  teradata
2     c  platform
```

```
>>> new_df2
   col7   col4
0     b  analytics
1     a  teradata
2     d  are you
```

```
>>> new_df1.merge(right = new_df2, how = "inner", on = "col5=col7")
   col5   col4   col2 col7
0     b  analytics  analytics  b
1     a  teradata  teradata  a
```

Example 5: No merge conditions are specified

When no merge conditions are specified, teradataml DataFrame indexes are used as merge keys.

```
>>> df1.merge(right = df2, how = "full", lsuffix = "t1", rsuffix = "t2")
   t2_col1 col5   t2_col 6 t1_col1 t2_'col8' t1_col3   col4 t2_col3 col7   col2 t1_col 6 t1_'col8'
0         2     b  1991-11-01         2         4     2.3  analytics     2.3     b  analytics  1991-11-01         4
1         1     a  1991-10-31         1         3     1.3  teradata     1.3     a  teradata  1991-10-31         3
2         3     c  1991-11-03         3         5     3.3  are you     4.3     d  platform  1991-11-02         5
```

sample() Method

The sample() function samples rows from a DataFrame, directly or based on conditions. The function creates a new column 'sampleid' which has a unique ID for each sample, helping identify each sample.

Note:

If more than one sample() operations are performed on teradataml DataFrame, then 'sampleid' from the latest call is projected. Previous 'sampleid' columns are ignored.

In this example, 'sampleid' column shown is from the latest sample() operation (sample(frac = [0.5, 0.1])).

```
>>> from teradataml import *
>>> load_example_data("dataframe", "admissions_train")
>>> df = DataFrame("admissions_train")

>>> df.sample(n = 20).sample(frac = [0.5, 0.1])
   masters  gpa  stats programming  admitted  sampleid
id
15    yes  4.00  Advanced    Advanced         1         1
37     no  3.52   Novice     Novice         1         1
35     no  3.68   Novice    Beginner         1         2
17     no  3.83  Advanced    Advanced         1         1
9      no  3.82  Advanced    Advanced         1         1
3      no  3.70   Novice    Beginner         1         1
34    yes  3.85  Advanced    Beginner         0         1
39    yes  3.75  Advanced    Beginner         0         1
36     no  3.00  Advanced     Novice         0         2
19    yes  1.98  Advanced    Advanced         0         1
```

In the following example, two sample() operations are performed on a DataFrame, but not consecutively. The 'sampleid' column in the result is from the latest sample operation (sample(n = [5, 4])). 'sampleid' column from the previous call is ignored.

```
>>> from teradataml import*
>>> load_example_data("dataframe", "admissions_train")
>>> df = DataFrame.from_table('admissions_train')

>>> df.sample(frac = 0.8).filter(items = ["masters"]).sample(n = [5, 4])
   masters  sampleid
0     yes         1
1     no         1
2     yes         1
3     no         2
4     no         1
5     no         2
6     yes         2
7     yes         1
8     yes         2
```

Example Prerequisite

```
>>> from teradataml import *
>>> load_example_data("dataframe", "admissions_train")
>>> df = DataFrame("admissions_train")
```

```
>>> df
   masters  gpa  stats programming admitted
id
13      no  4.00  Advanced      Novice        1
26     yes  3.57  Advanced    Advanced        1
5       no  3.44    Novice      Novice         0
19     yes  1.98  Advanced    Advanced         0
15     yes  4.00  Advanced    Advanced         1
40     yes  3.95    Novice  Beginner         0
7       yes  2.33    Novice      Novice         1
22     yes  3.46    Novice  Beginner         0
36      no  3.00  Advanced      Novice         0
38     yes  2.65  Advanced  Beginner         1
```

Example 1: Sample one specific number of rows

This example randomly samples 2 rows from the teradataml DataFrame. As there is only one sample, the 'sampleid' is 1.

```
>>> df.sample(n = 2)
   masters  gpa  stats programming admitted SampleId
id
18     yes  3.81  Advanced    Advanced         1         1
19     yes  1.98  Advanced    Advanced         0         1
```

Example 2: Sample multiple values for the number of rows to be sampled

This example creates two samples, one with 2 rows and one with 1 row. There are two values (1,2) for 'sampleid', each indicates one sample.

```
>> df.sample(n = [2, 1])
   masters  gpa  stats programming admitted SampleId
id
1       yes  3.95  Beginner  Beginner         0         1
10      no  3.71  Advanced    Advanced         1         1
11      no  3.13  Advanced    Advanced         1         2
```


Example 3: Sample one specific percentage of rows

This example randomly samples 20% of the total rows in the input teradataml DataFrame.

```
>>> df.sample(frac = 0.2)
   masters  gpa  stats programming admitted SampleId
id
18    yes  3.81  Advanced    Advanced         1         1
15    yes  4.00  Advanced    Advanced         1         1
14    yes  3.45  Advanced    Advanced         0         1
35     no  3.68   Novice    Beginner         1         1
27    yes  3.96  Advanced    Advanced         0         1
25     no  3.96  Advanced    Advanced         1         1
10     no  3.71  Advanced    Advanced         1         1
9      no  3.82  Advanced    Advanced         1         1
```

Example 4: Sample multiple values for the percentage of rows to be sampled

This example creates two samples, one with 4% of total rows and one with 2% of total rows.

```
>>> df.sample(frac = [0.04, 0.02])
   masters  gpa  stats programming admitted SampleId
id
29    yes  4.00   Novice    Beginner         0         1
19    yes  1.98  Advanced    Advanced         0         2
11     no  3.13  Advanced    Advanced         1         1
```

Example 5: Sample specific number of rows, replace and randomization

This example creates two samples, one with 2 rows and one with 1 row, with possible redundant sampling as replace is True and also selects rows from different AMPs as randomize is True.

```
>>> df.sample(n = [2, 1], replace = True, randomize = True)
   masters  gpa  stats programming admitted SampleId
id
12     no  3.65   Novice    Novice         1         1
39    yes  3.75  Advanced    Beginner         0         1
20    yes  3.90  Advanced    Advanced         1         2
```

Example 6: Sample specific percentage of rows, replace and randomization

This example creates two samples, one with 4% of total rows and one with 2% of total rows in teradataml DataFrame, with possible redundant sampling and also selects rows from different AMPs.

```
>>> df.sample(frac = [0.04, 0.02], replace = True, randomize = True)
  masters  gpa  stats programming admitted SampleId
id
7      yes  2.33  Novice      Novice         1         2
30     yes  3.79  Advanced    Novice         0         1
33     no   3.55  Novice      Novice         1         1
```

Example 7: Sample with condition and number of samples to be sampled

This example creates two samples, with 1, 2 rows respectively from rows which satisfy `df.gpa < 2` and 2.5% of rows from rows which satisfy `df.stats == 'Advanced'`.

```
>>> df.sample(case_when_then={df.gpa < 2 : [1, 2], df.stats ==
'Advanced' : 0.025})
  masters  gpa  stats programming admitted SampleId
id
19     yes  1.98  Advanced    Advanced         0         1
24     no   1.87  Advanced    Novice          1         1
11     no   3.13  Advanced    Advanced         1         3
```

Example 8: Sample with condition and number of samples to be sampled, replace and randomization

This example creates two samples with 1 and 2 rows respectively from rows which satisfy `df.gpa < 2` and 2.5% of rows from rows which satisfy `df.stats == 'Advanced'` and selects rows from different AMPs with replacement.

```
>>> df.sample(replace = True, randomize = True, case_when_then={df.gpa < 2 :
[1, 2], df.stats == 'Advanced' : 0.025})
  masters  gpa  stats programming admitted SampleId
id
24     no   1.87  Advanced    Novice          1         1
24     no   1.87  Advanced    Novice          1         2
24     no   1.87  Advanced    Novice          1         2
24     no   1.87  Advanced    Novice          1         2
24     no   1.87  Advanced    Novice          1         2
24     no   1.87  Advanced    Novice          1         1
31     yes  3.50  Advanced    Beginner         1         3
```

Example 9: Sample with different conditions and numbers of samples to be sampled

This example creates creates seven samples:

- Two with 1, 3 rows from rows which satisfy `df.gpa > 2`

- One with 5 rows from rows which satisfy `df.programming == 'Novice'`
- One with 5 rows from rows which satisfy `df.masters == 'no'`
- One with 1 row from rows which does not meet all the previous conditions

```
>>> df.sample(case_when_then = {df.gpa > 2 : [1, 3], df.stats == 'Novice' : [1, 2], df.programming == 'Novice' : 5, df.masters == 'no': 5}, case_else = 1)
```

| | masters | gpa | stats | programming | admitted | SampleId |
|----|---------|------|----------|-------------|----------|----------|
| id | | | | | | |
| 24 | no | 1.87 | Advanced | Novice | 1 | 5 |
| 2 | yes | 3.76 | Beginner | Beginner | 0 | 1 |
| 12 | no | 3.65 | Novice | Novice | 1 | 2 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 | 2 |
| 36 | no | 3.00 | Advanced | Novice | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 7 |

Example 10: Sample with different conditions and numbers of samples to be sampled, replace and randomization

This example creates Four samples:

- Two with 1, 3 rows from rows which satisfy `df.gpa > 2`
- Two with 2.5%, 5% of rows from rows which does not meet the previous condition with possible redundant replacement and also select rows from different AMPs

```
>>> df.sample(case_when_then = {df.gpa < 2 : [1, 3]}, replace = True, randomize = True, case_else = [0.025, 0.05])
```

| | masters | gpa | stats | programming | admitted | SampleId |
|----|---------|------|----------|-------------|----------|----------|
| id | | | | | | |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 40 | yes | 3.95 | Novice | Beginner | 0 | 3 |
| 3 | no | 3.70 | Novice | Beginner | 1 | 4 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 2 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 1 |

set_index() Method

Use the `set_index()` function to assign an appropriate index to a teradataml DataFrame.

The `keys` parameter is used to assign one or more existing columns as the new index to a teradataml DataFrame. The argument can be a single column name or a list of column names.

Arguments:

- *append*: Allows the user to specify whether or not to append requested columns to an already existing index (if any).
- *drop*: Allows the user to specify whether or not to display the requested index being assigned as a column of the teradataml DataFrame.

Examples Prerequisite

Assume a teradataml DataFrame is created based on the table "df_admissions_train".

```
>>> df1 = DataFrame.from_table('df_admissions_train')

>>> df1
```

| | id | masters | gpa | stats | programming | admitted |
|---|----|---------|------|----------|-------------|----------|
| 0 | 26 | yes | 3.57 | advanced | advanced | 1 |
| 1 | 34 | yes | 3.85 | advanced | beginner | 0 |
| 2 | 40 | yes | 3.95 | novice | beginner | 0 |
| 3 | 14 | yes | 3.45 | advanced | advanced | 0 |
| 4 | 29 | yes | 4.0 | novice | beginner | 0 |
| 5 | 6 | yes | 3.5 | beginner | advanced | 1 |
| 6 | 36 | no | 3.0 | advanced | novice | 0 |
| 7 | 32 | yes | 3.46 | advanced | beginner | 0 |
| 8 | 5 | no | 3.44 | novice | novice | 0 |

Example 1: Assign a single index column 'id' as the index

This example assigns a single index column 'id' as the index to a teradataml DataFrame without an index created from the 'admissions_train' table.

```
>>> df2 = df1.set_index("id")

>>> df2
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 26 | yes | 3.57 | advanced | advanced | 1 |
| 34 | yes | 3.85 | advanced | beginner | 0 |
| 40 | yes | 3.95 | novice | beginner | 0 |
| 14 | yes | 3.45 | advanced | advanced | 0 |
| 29 | yes | 4.0 | novice | beginner | 0 |
| 6 | yes | 3.5 | beginner | advanced | 1 |
| 36 | no | 3.0 | advanced | novice | 0 |
| 32 | yes | 3.46 | advanced | beginner | 0 |
| 5 | no | 3.44 | novice | novice | 0 |

Example 2: Assign a multicolumn index using a list of columns names

This examples uses a list of columns names to assign a multicolumn index.

```
>>> df3 = df1.set_index(["id", "masters"])
>>> df3
```

| | gpa | stats | programming | admitted |
|------------|------|----------|-------------|----------|
| id masters | | | | |
| 26 yes | 3.57 | advanced | advanced | 1 |
| 34 yes | 3.85 | advanced | beginner | 0 |
| 40 yes | 3.95 | novice | beginner | 0 |
| 14 yes | 3.45 | advanced | advanced | 0 |
| 29 yes | 4.0 | novice | beginner | 0 |
| 6 yes | 3.5 | beginner | advanced | 1 |
| 36 no | 3.0 | advanced | novice | 0 |
| 32 yes | 3.46 | advanced | beginner | 0 |
| 5 no | 3.44 | novice | novice | 0 |

Example 3: Add an additional column as an index

This example add an additional column as an index to a teradataml DataFrame that already has an index.

```
>>> df4 = df3.set_index("gpa", append = True, drop = True)
>>> df4
```

| | stats | programming | admitted |
|----------------|----------|-------------|----------|
| id masters gpa | | | |
| 26 yes 3.57 | advanced | advanced | 1 |
| 34 yes 3.85 | advanced | beginner | 0 |
| 40 yes 3.95 | novice | beginner | 0 |
| 14 yes 3.45 | advanced | advanced | 0 |
| 29 yes 4.0 | novice | beginner | 0 |
| 6 yes 3.5 | beginner | advanced | 1 |
| 36 no 3.0 | advanced | novice | 0 |
| 32 yes 3.46 | advanced | beginner | 0 |
| 5 no 3.44 | novice | novice | 0 |

Example 4: Display an assigned index as a column of a DataFrame

This example displays an assigned index as a column of a teradataml DataFrame after assigning it as an index.

```
>>> df5 = df3.set_index("gpa", append = True, drop = False)
>>> df5
```

| | gpa | stats | programming | admitted |
|--|-----|-------|-------------|----------|
|--|-----|-------|-------------|----------|

```

id masters gpa
26 yes      3.57 3.57 advanced advanced 1
34 yes      3.85 3.85 advanced beginner 0
40 yes      3.95 3.95 novice  beginner 0
14 yes      3.45 3.45 advanced advanced 0
29 yes      4.0  4.0  novice  beginner 0
6  yes      3.5  3.5  beginner advanced 1
36 no       3.0  3.0  advanced novice    0
32 yes      3.46 3.46 advanced beginner 0
5  no       3.44 3.44 novice  novice    0

```

show_query() Method

Use the `show_query()` method to return underlying SQL for a teradataml DataFrame. It is the same SQL that is used to view the data of the teradataml DataFrame.

Arguments:

The optional argument *full_query* specifies if the complete query for the dataframe should be returned.

- If set to True, query for the dataframe is returned with respect to the base dataframe's table ("from_table()" or "from_query()"), or from the output tables of analytical functions, if there are any in the workflow.

Note:

This query may or may not be directly used to retrieve data of the dataframe upon which the function is called.

- If set to False or not used, the string returned is the query already used or will be used to retrieve data for the teradataml DataFrame.

This is the default value.

Note:

`show_query()` API is not intended to be used on the output of the following APIs:

- `set_index()`
 - `sort()`
 - `sort_index()`
 - `squeeze()`
-

Note:

In most cases, queries returned by `show_query()` with `full_query` set to `False`, can be executed as is on Vantage. But there are certain functions which are lazy, some functions are executed at the time when actual data is requested. When such DataFrames are manipulated and `show_query()` is executed on the outcome, such queries may or may not be executable on Vantage.

Following are some analytic functions which creates lazy DataFrames:

- ML Engine functions:
 - Attribution
 - CoxPH
 - CoxHazardRatio
 - DecisionForest
 - DecisionForestPredict
 - FPGrowth
 - GLML1L2Predict
 - NaiveBayes
 - PageRank
 - POSTagger
 - Sessionize
 - XGBoost
- Analytics Database functions:
 - MovingAverage
 - NaiveBayesTextClassifierPredict

Examples Prerequisite

Load example data and create required DataFrame on it.

```
>>> load_example_data("dataframe", "admissions_train")
>>> load_example_data("NaiveBayes", "nb_iris_input_train")

>>> df = DataFrame.from_table("admissions_train")
```

Example 1: Show query on base (from_table) dataframe, with default option.

```
>>> df.show_query()
'select * from "admissions_train"'
```

Example 2: Show query on base (from_query) dataframe, with default option.

```
>>> df_from_query = DataFrame.from_query("select masters, gpa
from admissions_train")
```

```
>>> df_from_query.show_query()
'select masters, gpa from admissions_train'
```

Example 3: Show query on base (from_table) dataframe, with full_query option.

This will return same query as with default option because workflow only has one dataframe.

```
>>> df.show_query(full_query = True)
'select * from "admissions_train"
```

Example 4: Show query on base (from_query) dataframe, with full_query option.

```
>>> df_from_query = DataFrame.from_query("select masters, gpa
from admissions_train")
```

```
>>> df_from_query.show_query(full_query = True)
'select masters, gpa from admissions_train'
```

Example 5: Show query used in a workflow demonstrating default and full_query options.

- Assign operation on base dataframe:

```
# Workflow Step-1: Assign operation on base dataframe
>>> df1 = df.assign(temp_column=admissions_train_df.gpa
+ admissions_train_df.admitted)
```

- Select columns from assign's result:

```
# Workflow Step-2: Selecting columns from assign's result
>>> df2 = df1.select(["masters", "gpa", "programming", "admitted"])
```

- Filter on top of select's result:

```
# Workflow Step-3: Filtering on top of select's result
>>> df3 = df2[df2.admitted > 0]
```

- Sample 90% rows from filter's result:


```
# Workflow Step-4: Sampling 90% rows from filter's result
>>> df4 = df3.sample(frac=0.9)
```

- Show query with full_query option on df4:

```
# Show query with full_query option on df4.
# This will give full query upto base dataframe(df)
>>> df4.show_query(full_query = True)
'select masters,gpa,stats,programming,admitted,sampleid as "sampleid" from (
  select * from (select masters,gpa,stats,programming,admitted from (select
id AS
  id, masters AS masters, gpa AS gpa, stats AS stats, programming
AS programming,
  admitted AS admitted, gpa + admitted AS temp_column from
"admissions_train") as
  temp_table) as temp_table where admitted > 0) as temp_table SAMPLE 0.9'
```

- Show query with default option on df4:

```
# Show query with default option on df4. This will give same query as give
in above case.
>>> df4.show_query()
'select masters,gpa,stats,programming,admitted,sampleid as "sampleid" from
(select *
  from (select masters,gpa,stats,programming,admitted from (select id AS
id, masters
  AS masters, gpa AS gpa, stats AS stats, programming AS programming,
admitted AS admitted,
  gpa + admitted AS temp_column from "admissions_train") as temp_table)
as temp_table
  where admitted > 0) as temp_table SAMPLE 0.9'
```

- Execute intermediate dataframe df3:

```
>>> df2
  masters  gpa programming  admitted
0      no  4.00      Novice          1
1     yes  3.57    Advanced          1
2      no  3.44      Novice          0
3     yes  1.98    Advanced          0
4     yes  4.00    Advanced          1
5     yes  3.95    Beginner          0
6     yes  2.33      Novice          1
7     yes  3.46    Beginner          0
```

| | | | | |
|---|-----|------|----------|---|
| 8 | no | 3.00 | Novice | 0 |
| 9 | yes | 2.65 | Beginner | 1 |

- Show query with default option on df4:

```
# Show query with default option on df4. This will give query with respect
# to view/table created by the latest executed dataframe in the workflow
# (df2 in this scenario).
# This is the query teradataml internally uses to retrieve data for
# dataframe df4, if executed
# at this point.
>>> df4.show_query()
'select masters,gpa,stats,programming,admitted,sampleid as "sampleid" from
(select * from
"ALICE"."ml__select__1585722211621282" where admitted > 0) as temp_table
SAMPLE 0.9'
```

- Show query with full_query option on df4:

```
>>> df4.show_query(full_query = True)
'select masters,gpa,stats,programming,admitted,sampleid as "sampleid" from
(select *
from (select masters,gpa,stats,programming,admitted from (select id AS
id, masters
AS masters, gpa AS gpa, stats AS stats, programming AS programming,
admitted AS admitted,
gpa + admitted AS temp_column from "admissions_train") as temp_table)
as temp_table
where admitted > 0) as temp_table SAMPLE 0.9'
```

sort() Method

Use the `sort()` method to sort data on one or more columns in either ascending or descending order for a teradataml DataFrame.

The method takes a column name or a list of column names to sort on. Use the ascending parameter to specified ascending or descending order. True for ascending order and False for descending order.

Note:

The column used for sorting in sort must have type that supports sorting. Unsupported types include: 'BLOB', 'CLOB', 'ARRAY', 'VARRAY'.

Examples Prerequisite

Assume a teradataml DataFrame "df" is created from a Vantage table "admissions_train", using command:

```
>>> df = DataFrame("admissions_train")
```

Example 1: Sort in ascending order

This example sorts in ascending order on the column "id" of "admissions_train":

```
>>> df.sort("id")
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 1 | yes | 3.95 | beginner | beginner | 0 |
| 2 | yes | 3.76 | beginner | beginner | 0 |
| 3 | no | 3.70 | novice | beginner | 1 |
| 4 | yes | 3.50 | beginner | novice | 1 |
| 5 | no | 3.44 | novice | novice | 0 |
| 6 | yes | 3.50 | beginner | advanced | 1 |
| 7 | yes | 2.33 | novice | novice | 1 |
| 8 | no | 3.60 | beginner | advanced | 1 |
| 9 | no | 3.82 | advanced | advanced | 1 |
| 10 | no | 3.71 | advanced | advanced | 1 |

Example 2: Sort in descending order

This example sorts in descending order on the column "id" of "admissions_train":

```
>>> df.sort("id", ascending=False)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 40 | yes | 3.95 | novice | beginner | 0 |
| 39 | yes | 3.75 | advanced | beginner | 0 |
| 38 | yes | 2.65 | advanced | beginner | 1 |
| 37 | no | 3.52 | novice | novice | 1 |
| 36 | no | 3.00 | advanced | novice | 0 |
| 35 | no | 3.68 | novice | beginner | 1 |
| 34 | yes | 3.85 | advanced | beginner | 0 |
| 33 | no | 3.55 | novice | novice | 1 |
| 32 | yes | 3.46 | advanced | beginner | 0 |
| 31 | yes | 3.50 | advanced | beginner | 1 |

Example 3: Sort in ascending order on multiple columns

This example sorts in ascending order on the columns "masters" and "gpa":

```
>>> display.max_row = 20
```

```
>>> df.sort(["masters", "gpa"])
   masters  gpa  stats programming admitted
id
24      no  1.87  advanced      novice      1
36      no  3.00  advanced      novice      0
11      no  3.13  advanced  advanced      1
5       no  3.44   novice      novice      0
37      no  3.52   novice      novice      1
33      no  3.55   novice      novice      1
8       no  3.60  beginner  advanced      1
12      no  3.65   novice      novice      1
35      no  3.68   novice  beginner      1
3       no  3.70   novice  beginner      1
16      no  3.70  advanced  advanced      1
10      no  3.71  advanced  advanced      1
9       no  3.82  advanced  advanced      1
17      no  3.83  advanced  advanced      1
21      no  3.87   novice  beginner      1
28      no  3.93  advanced  advanced      1
25      no  3.96  advanced  advanced      1
13      no  4.00  advanced   novice      1
19     yes  1.98  advanced  advanced      0
7      yes  2.33   novice   novice      1
```

sort_index() Method

Use the `sort_index()` method to get object sorted by labels (along an axis) in either ascending or descending order for a teradataml DataFrame.

Example 1: Default behavior of `sort_index()` when no arguments is given.

```
>>> load_example_data("dataframe", "scale_housing_test")

>>> df = DataFrame.from_table('scale_housing_test')
>>> df
      id  price  lotsize  bedrooms  bathrms  stories
types
classic  14  36000.0   2880.0        3.0      1.0      1.0
bungalow 11  90000.0   7200.0        3.0      2.0      1.0
classic  15  37000.0   3600.0        2.0      1.0      1.0
classic  13  27000.0   1700.0        3.0      1.0      2.0
classic  12  30500.0   3000.0        2.0      1.0      1.0
```

```
>>> df.sort_index()
           id    price  lotsize  bedrooms  bathrms  stories
types
bungalow  11  90000.0   7200.0         3.0      2.0      1.0
classic   13  27000.0   1700.0         3.0      1.0      2.0
classic   12  30500.0   3000.0         2.0      1.0      1.0
classic   14  36000.0   2880.0         3.0      1.0      1.0
classic   15  37000.0   3600.0         2.0      1.0      1.0
```

Example 2: Use `sort_index()` with **DESCENDING for respective axis.**

```
>>> load_example_data("dataframe", "scale_housing_test")

>>> df = DataFrame.from_table('scale_housing_test')

>>> df.sort_index(1, False)
           stories    price  lotsize  id  bedrooms  bathrms
types
classic         1.0  36000.0   2880.0  14         3.0      1.0
bungalow        1.0  90000.0   7200.0  11         3.0      2.0
classic         1.0  37000.0   3600.0  15         2.0      1.0
classic         2.0  27000.0   1700.0  13         3.0      1.0
classic         1.0  30500.0   3000.0  12         2.0      1.0
```

Example 3: Use `sort_index()` with type of sorting algorithm.

```
>>> load_example_data("dataframe", "scale_housing_test")

>>> df = DataFrame.from_table('scale_housing_test')

>>> df.sort_index(1, True, 'mergesort')
           bathrms  bedrooms  id  lotsize    price  stories
types
classic         1.0        3.0  14   2880.0  36000.0      1.0
bungalow        2.0        3.0  11   7200.0  90000.0      1.0
classic         1.0        2.0  15   3600.0  37000.0      1.0
classic         1.0        3.0  13   1700.0  27000.0      2.0
classic         1.0        2.0  12   3000.0  30500.0      1.0
```

squeeze() Method

Use the `squeeze()` method to squeeze one-dimensional axis objects into a scalar for teradataml DataFrames with a single element, or a Series object for a teradataml DataFrame with a single column.

The teradataml DataFrame is returned unchanged when both dimensions are greater than one.

Arguments:

The *axis* argument specifies the axis along which the squeeze operation is to be attempted. The possible values are:

- 1 or 'columns': Return Series object if number of columns equals one.
- 0 or 'index': Return the unchanged teradataml DataFrame object.

When axis is not specified and both dimensions equal one, a scalar object is returned which is the only element in the DataFrame.

Examples Prerequisite

Assume the table "admissions_train" exists and its index column is id. And a DataFrame "df" is created based on this table using the command:

```
>>> df = DataFrame("admissions_train")

>>> df
   masters  gpa  stats programming admitted
id
22     yes  3.46  Novice    Beginner         0
36      no  3.00  Advanced    Novice         0
15     yes  4.00  Advanced  Advanced         1
38     yes  2.65  Advanced  Beginner         1
5       no  3.44  Novice    Novice         0
17      no  3.83  Advanced  Advanced         1
34     yes  3.85  Advanced  Beginner         0
13      no  4.00  Advanced    Novice         1
26     yes  3.57  Advanced  Advanced         1
19     yes  1.98  Advanced  Advanced         0
```

Example 1: Squeeze a teradataml DataFrame with both dimension greater than one.

```
>>> df.squeeze()
   masters  gpa  stats programming admitted
id
22     yes  3.46  Novice    Beginner         0
36      no  3.00  Advanced    Novice         0
15     yes  4.00  Advanced  Advanced         1
38     yes  2.65  Advanced  Beginner         1
5       no  3.44  Novice    Novice         0
17      no  3.83  Advanced  Advanced         1
34     yes  3.85  Advanced  Beginner         0
```

| | | | | | |
|----|-----|------|----------|----------|---|
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

Example 2: Squeeze a single-column teradataml DataFrame.

```
>>> gpa = df.select(["gpa"])
>>> gpa.squeeze()
0    4.00
1    2.33
2    3.46
3    3.83
4    4.00
5    2.65
6    3.57
7    3.44
8    3.85
9    3.95
Name: gpa, dtype: float64
```

```
>>> gpa.squeeze(axis = 1)
0    3.46
1    3.00
2    4.00
3    2.65
4    3.44
5    3.83
6    3.85
7    4.00
8    3.57
9    1.98
Name: gpa, dtype: float64
```

```
>>> gpa.squeeze(axis = 0)
      gpa
0  3.46
1  3.00
2  4.00
3  2.65
4  3.44
5  3.83
6  3.85
7  4.00
```

```
8  3.57
9  1.98
```

Example 3: Squeeze a teradataml DataFrame with multiple columns and a single row.

```
>>> df = DataFrame.from_query('select gpa, stats from admissions_train
where gpa=2.33')
```

```
>>> df
      gpa  stats
0  2.33  Novice
```

```
>>> s = df.squeeze()
>>> s
      gpa  stats
0  2.33  Novice
```

Example 4: Squeeze a teradataml DataFrame with a single element.

```
>>> single_gpa = DataFrame.from_query('select gpa from admissions_train
where gpa=2.33')
```

```
>>> single_gpa
      gpa
0  2.33
```

```
>>> single_gpa.squeeze()
2.33
```

```
>>> single_gpa.squeeze(axis = 1)
0    2.33
Name: gpa, dtype: float64
```

```
>>> single_gpa.squeeze(axis = 0)
      gpa
0  2.33
```

map_row() and map_partition() Methods

Use the `map_row()` and `map_partition()` methods to apply a Python function to each row or group of rows in a DataFrame, and return the result to the user as a teradataml DataFrame.

Python's built-in `map` function allows the application of a function to items in a list. The `map_row()` and `map_partition()` methods are similar to it, allowing the application of a function to each row or each group (partition) of rows, respectively.

For a user to run Python code in-database, Analytics Database offers Script Table Operator (STO). Both the `map_row()` and `map_partition()` methods leverage STO through the Script object in `teradataml`.

To use STO, a user needs to:

1. Write the Python code to process the rows in a script.
2. Make sure that the script takes care of parsing the input data from a stream to be able to access the values in the individual columns.
3. Install this script in Vantage.
4. Run the STO query to invoke this script on the required dataset on Analytics Database.
5. (Optional) Remove the installed script.

The `map_row()` and `map_partition()` methods take care of all these steps on user's behalf, so that the user can focus on the Python function and not the logistics involved in being able to leverage STO to use it.

Note:

- The Python function being applied to the row or group of rows using these two methods must be defined in the current Python session.

Any modules or packages being used by it must be available to use with SCRIPT Table Operator on the Vantage servers, and must be imported within the function scope. Modules or packages imported by the function need not be present on the client, unless the same function is also called in the current Python session outside the `map_row()` and `map_partition()` methods.

If the Python function being applied using `map_row()` or `map_partition()` is imported from a specific module or package, then the source module or package must also be available on the Vantage servers.

- The versions of any modules or packages being used by the applied function must be the same as or compatible with the version of the same packages installed on the Vantage servers.
- Both methods use `dill` to rewrite the function to be applied in a script format, which is sensitive to version differences.

User must make sure that the version of `dill` on the client machine is the same as that of the one installed on the Vantage servers.

Functions, Inputs, and Outputs

Both methods return a `teradataml` DataFrame when run on the Analytics Database (`exec_mode = 'IN-DB'`).

Inputs:

Both methods support user defined functions (regular Python function and lambda notation).

The user defined function can accept as many arguments as required, but must accept the following as its first (positional) argument:

- When the method called is `map_row()`: a pandas Series object corresponding to a row in the DataFrame.

That is, a row in a Pandas DataFrame corresponding to the teradataml DataFrame to which it is to be applied.

- When the method called is `map_partition()`: an iterator (TextFileReader object) to read data from the partition of rows in chunks (pandas DataFrames).

That is, an iterator on the partition of rows from the teradataml DataFrame represented as a Pandas DataFrame to which it is to be applied.

Thus, the user function has access to the data to process in a familiar format, without having to worry about how it will be streamed from the STO to the function itself.

Outputs:

The functions can either print the output directly to the standard output (just like STO) or return objects of supported types that can be printed to the standard output correctly.

User function can return an object of the following supported types:

- pandas DataFrame: having the same number of columns as expected in the output.
- pandas Series: represents a row in the output, having the same number of columns as expected in the output.
- numpy ndarray:
 - One-dimensional array: represents a row in the output, having the same number of columns as expected in the output.
 - Two-dimensional array: represents a dataset (like a pandas DataFrame), every array contained in the outer array having the same number of columns as expected in the output.

Then the returned object is printed to the standard output as delimited lines (rows), using specified *delimiter* and *quotechar*.

If the user function prints the output directly to the standard output (instead of returning an object of the supported type), then it must take care of using the *delimiter* and *quotechar*, if and when specified, to format the output printed.

The data printed to the standard output then gets converted to and saved in a table on the Analytics Database. The table is deleted as a part of garbage collection as soon as a `remove_context()` call is issued. To persist the results, the `DataFrame.to_sql()` method can be used.

Note:

It is recommended that empty values in characters columns are filled or replaced by a known placeholder value for better readability with `map_row()` and `map_function()` to avoid confusing NULL values with empty string.

Testing Mode

Users can test scripts using local system environment by setting the execution mode to local (`exec_mode='local'`).

Both `map_row()` and `map_partition()` methods return a pandas DataFrame, if the `exec_mode` is set to 'local'.

The sample data which is used to test the scripts should contain at most the number of rows as specified by `num_rows`.

map_row() Method

Use the `map_row()` method to apply a function to every row in a teradataml DataFrame and return a teradataml DataFrame.

Required arguments:

- **`user_function`:** Specifies the user defined function to apply to each row in the teradataml DataFrame.

This can be either a lambda function, a regular Python function, or an object of `functools.partial`.

A non-lambda function can be passed only when the user defined function does not accept any arguments other than the mandatory input - the input row.

A user can also use `functools.partial` and lambda functions for the same, when:

- For lambda function, there is a need to pass positional, or keyword, or both arguments.
- For `functools.partial`, there is a need to pass keyword arguments only.

See the "Functions, Inputs and Outputs" section in [map_row\(\) and map_partition\(\) Methods](#) for details about the input and output of this argument.

Optional arguments:

- **`exec_mode`:** Specifies the mode of execution for the user defined function.

Permitted values:

- **IN-DB:** Execute the function on data in the teradataml DataFrame in Vantage.

This is the default value.

- **LOCAL:** Execute the function locally on sample data (at most `num_rows` rows) from the teradataml DataFrame.

- **`chunk_size`:** Specifies the number of rows to be read in a chunk in each iteration using an iterator to apply the user defined function to each row in the chunk.

Varying the value passed to this argument affects the performance and the memory utilization. Default value is 1000.

- **`num_rows`:** Specifies the maximum number of sample rows to use from the teradataml DataFrame to apply the user defined function to when `exec_mode` is 'LOCAL'.
- **`returns`:** Specifies the output column definition corresponding to the output of `user_function`.

When not specified, the function assumes that the names and types of the output columns are the same as those of the input.

Note:

Teradata reserved keywords should not be provided as column names unless the column names of output dataframe are an exact match of input dataframe. User can find the list or check if the string is a reserved keyword or not using the [list_td_reserved_keywords\(\)](#) function.

- *delimiter*: Specifies a delimiter to use when reading columns from a row and writing result columns. The default value is '\t'.
-

Note:

- This argument cannot be the same as *quotechar* argument.
 - This argument cannot be newline character '\n'.
-

- *quotechar*: Specifies a character that forces all input and output of the user function to be quoted using this specified character.

Using this argument enables the Analytics Database to distinguish between NULL fields and empty strings. A string with length zero is quoted, while NULL fields are not.

If this character is found in the data, it will be escaped by a second quote character.

Note:

- This argument cannot be the same as *delimiter* argument.
 - This argument cannot be newline character '\n'.
-

- *auth*: Specifies an authorization to use when running the *user_function*.
- *charset*: Specifies the character encoding for data.

Permitted values are 'utf-16' and 'latin'.

- *data_order_column*: Specifies the Order By columns for the teradataml DataFrame.

Values to this argument can be provided as a list, if multiple columns are used for ordering.

This argument is used in both cases: "is_local_order = True" and "is_local_order = False".

Note:

is_local_order must be set to 'True' when *data_order_column* is used with *data_hash_column*.

- *is_local_order*: Specifies a boolean value to determine whether the input data is to be ordered locally or not. with

When this argument is set to 'False' (default), *data_order_columns* specifies the order in which the values in a group, or partition, are sorted.

When this argument is set to 'True', qualified rows on each AMP are ordered in preparation to be input to a table function.

This argument is ignored, if *data_order_column* is None.

Note:

- This argument cannot be specified along with *data_partition_column*.
 - When this argument is set to True, *data_order_column* should be specified, and the columns specified in *data_order_column* are used for local ordering.
-

- *nulls_first*: Specifies a boolean value to determine whether NULLS are listed first or last during ordering.

NULLS are listed first when this argument is set to 'True', and last when set to 'False'.

This argument is ignored, if *data_order_column* is None.

- *sort_ascending*: Specifies a boolean value to determine if the result set is to be sorted on the *data_order_column* column in ascending or descending order.

The sorting is ascending when this argument is set to 'True', and descending when set to 'False'.

This argument is ignored, if *data_order_column* is None.

This function returns:

- teradataml DataFrame if *exec_mode* is "IN-DB"
- Pandas DataFrame if *exec_mode* is "LOCAL".

The method also accepts the same arguments that Script accepts, except that with *returns* is optional and the method does not accept *data*, *data_hash_column*, and *data_partition_column*. When *returns* is not provided, the method assumes that the function's output has the columns with the same names and types as the input teradataml DataFrame.

Example Prerequisite

The examples use the 'admissions_train' dataset, calculates the average 'gpa' per partition based on the value in 'admitted' column.

- Load the example data.

```
>>> load_example_data("dataframe", "admissions_train")
```

- Create a DataFrame.

```
>>> df = DataFrame('admissions_train')
```

```
>>> print(df)
   masters  gpa  stats programming  admitted
id
5         no  3.44   Novice      Novice         0
34        yes  3.85  Advanced   Beginner         0
13        no  4.00  Advanced     Novice         1
40        yes  3.95   Novice   Beginner         0
22        yes  3.46   Novice   Beginner         0
19        yes  1.98  Advanced   Advanced         0
36        no  3.00  Advanced     Novice         0
15        yes  4.00  Advanced   Advanced         1
7         yes  2.33   Novice     Novice         1
17        no  3.83  Advanced   Advanced         1
```

Example 1: Create a user defined function to increase the 'gpa' by the percentage provided

In this example, the input to and the output from the function is a Pandas Series object.

1. Create a user defined function.

```
>>> def increase_gpa(row, p=20):
    row['gpa'] = row['gpa'] + row['gpa'] * p/100
    return row
```

2. Apply the user defined function to the DataFrame.

Since the output of the user defined function expects the same columns with the same types, you can skip passing the *returns* argument.

```
>>> increase_gpa_20 = df.map_row(increase_gpa)
```

3. Print the result.

```
>>> print(increase_gpa_20)
   masters  gpa  stats programming  admitted
id
13        no  4.800  Advanced     Novice         1
36        no  3.600  Advanced     Novice         0
15        yes  4.800  Advanced   Advanced         1
40        yes  4.740   Novice   Beginner         0
22        yes  4.152   Novice   Beginner         0
38        yes  3.180  Advanced   Beginner         1
26        yes  4.284  Advanced   Advanced         1
5         no  4.128   Novice     Novice         0
```

| | | | | | |
|----|-----|-------|----------|----------|---|
| 7 | yes | 2.796 | Novice | Novice | 1 |
| 19 | yes | 2.376 | Advanced | Advanced | 0 |

Example 2: Use the same user defined function with a lambda notation to pass the percentage 'p = 40'

1. Apply the user defined function to the DataFrame with a lambda notation.

```
>>> increase_gpa_40 = df.map_row(lambda row: increase_gpa(row, p = 40))
```

2. Print the result.

```
>>> print(increase_gpa_40)
  masters    gpa    stats programming  admitted
id
5       no  4.816   Novice      Novice         0
34      yes  5.390  Advanced   Beginner         0
13      no  5.600  Advanced     Novice         1
40      yes  5.530   Novice   Beginner         0
22      yes  4.844   Novice   Beginner         0
19      yes  2.772  Advanced   Advanced         0
36      no  4.200  Advanced     Novice         0
15      yes  5.600  Advanced   Advanced         1
7       yes  3.262   Novice     Novice         1
17      no  5.362  Advanced   Advanced         1
```

Example 3: Use the same user defined function with functools.partial to pass the percentage 'p = 50'

1. Load the necessary module.

```
>>> from functools import partial
```

2. Apply the user defined function to the DataFrame with functools.partial.

```
>>> increase_gpa_50 = df.map_row(partial(increase_gpa, p = 50))
```

3. Print the result.

```
>>> print(increase_gpa_50)
  masters    gpa    stats programming  admitted
id
5       no  5.160   Novice      Novice         0
34      yes  5.775  Advanced   Beginner         0
13      no  6.000  Advanced     Novice         1
40      yes  5.925   Novice   Beginner         0
22      yes  5.190   Novice   Beginner         0
```

| | | | | | |
|----|-----|-------|----------|----------|---|
| 19 | yes | 2.970 | Advanced | Advanced | 0 |
| 36 | no | 4.500 | Advanced | Novice | 0 |
| 15 | yes | 6.000 | Advanced | Advanced | 1 |
| 7 | yes | 3.495 | Novice | Novice | 1 |
| 17 | no | 5.745 | Advanced | Advanced | 1 |

Example 4: Use a lambda function to increase the 'gpa' by 100 percent, and return numpy ndarray

1. Load the necessary module.

```
>>> from numpy import asarray
```

2. Create a lambda function.

```
>>> increase_gpa_lambda = lambda row, p=20: asarray([row['id'],
row['masters'], row['gpa'] + row['gpa'] * p/100, row['stats'],
row['programming'], row['admitted']])
```

3. Apply the lambda function to the DataFrame.

```
>>> increase_gpa_100 = df.map_row(lambda row:
increase_gpa_lambda(row, p=100))
```

4. Print the result.

```
>>> print(increase_gpa_100)
  masters  gpa  stats programming  admitted
id
5      no  6.88  Novice      Novice         0
34     yes  7.70  Advanced  Beginner         0
13     no  8.00  Advanced    Novice         1
40     yes  7.90  Novice     Beginner         0
22     yes  6.92  Novice     Beginner         0
19     yes  3.96  Advanced  Advanced         0
36     no  6.00  Advanced    Novice         0
15     yes  8.00  Advanced  Advanced         1
7      yes  4.66  Novice     Novice         1
17     no  7.66  Advanced  Advanced         1
```

map_partition() Method

Use the `map_partition()` method to apply a function to a group or partition of rows in a teradataml DataFrame and return a teradataml DataFrame.

Required arguments:

- *user_function*: Specifies the user defined function to apply to each group or partition of rows in the teradataml DataFrame.

This can be either a lambda function, a regular Python function, or an object of `functools.partial`.

A non-lambda function can be passed only when the user defined function does not accept any arguments other than the mandatory input - the iterator on the partition of rows.

A user can also use `functools.partial` and lambda functions for the same, when:

- For lambda function, there is a need to pass positional, or keyword, or both arguments.
- For `functools.partial`, there is a need to pass keyword arguments only.

See the "Functions, Inputs and Outputs" section in [map_row\(\) and map_partition\(\) Methods](#) for details about the input and output of this argument.

Optional arguments:

- *exec_mode*: Specifies the mode of execution for the user defined function.

Permitted values:

- IN-DB: Execute the function on data in the teradataml DataFrame in Vantage.
This is the default value.
- LOCAL: Execute the function locally on sample data (at most *num_rows* rows) from the teradataml DataFrame.

- *chunk_size*: Specifies the number of rows to be read in a chunk in each iteration using an iterator to apply the user defined function to each row in the chunk.

Varying the value passed to this argument affects the performance and the memory utilization.
Default value is 1000.

- *num_rows*: Specifies the maximum number of sample rows to use from the teradataml DataFrame to apply the user defined function to when *exec_mode* is 'LOCAL'.
- *data_partition_column*: Specifies the Partition By columns for the teradataml DataFrame.

Values to this argument can be provided as a list, if multiple columns are used for partition.

- *data_hash_column*: Specifies the column to be used for hashing.

The rows in the teradataml DataFrame are redistributed to AMPs based on the hash value of the column specified. The *user_function* then runs once on each AMP.

If there is no *data_partition_column*, then the entire result set, delivered by the function, constitutes a single group or partition.

- *returns*: Specifies the output column definition corresponding to the output of *user_function*.
When not specified, the function assumes that the names and types of the output columns are the same as those of the input.

Note:

Teradata reserved keywords should not be provided as column names unless the column names of output dataframe are an exact match of input dataframe. User can find the list or check if the string is a reserved keyword or not using the [list_td_reserved_keywords\(\)](#) function.

- *delimiter*: Specifies a delimiter to use when reading columns from a row and writing result columns. The default value is '\t'.

Note:

- This argument cannot be the same as *quotechar* argument.
- This argument cannot be newline character '\n'.

- *quotechar*: Specifies a character that forces all input and output of the user function to be quoted using this specified character.

Using this argument enables the Analytics Database to distinguish between NULL fields and empty strings. A string with length zero is quoted, while NULL fields are not.

If this character is found in the data, it will be escaped by a second quote character.

Note:

- This argument cannot be the same as *delimiter* argument.
- This argument cannot be newline character '\n'.

- *auth*: Specifies an authorization to use when running the *user_function*.
- *charset*: Specifies the character encoding for data.

Permitted values are 'utf-16' and 'latin'.

- *data_order_column*: Specifies the Order By columns for the teradataml DataFrame.

Values to this argument can be provided as a list, if multiple columns are used for ordering.

This argument is used in both cases: "*is_local_order* = True" and "*is_local_order* = False".

Note:

is_local_order must be set to 'True' when *data_order_column* is used with *data_hash_column*.

- *is_local_order*: Specifies a boolean value to determine whether the input data is to be ordered locally or not. with

When this argument is set to 'False' (default), *data_order_columns* specifies the order in which the values in a group, or partition, are sorted.

When this argument is set to 'True', qualified rows on each AMP are ordered in preparation to be input to a table function.

This argument is ignored, if *data_order_column* is None.

Note:

- This argument cannot be specified along with *data_partition_column*.
 - When this argument is set to True, *data_order_column* should be specified, and the columns specified in *data_order_column* are used for local ordering.
-

- *nulls_first*: Specifies a boolean value to determine whether NULLS are listed first or last during ordering.

NULLS are listed first when this argument is set to 'True', and last when set to 'False'.

This argument is ignored, if *data_order_column* is None.

- *sort_ascending*: Specifies a boolean value to determine if the result set is to be sorted on the *data_order_column* column in ascending or descending order.

The sorting is ascending when this argument is set to 'True', and descending when set to 'False'.

This argument is ignored, if *data_order_column* is None.

Note:

- *data_partition_column* cannot be specified along with *data_hash_column*.
 - *data_partition_column* cannot be specified when *is_local_order* is set to 'True'.
 - *is_local_order* must be set to 'True' when *data_order_column* is used with *data_hash_column*.
-

This function returns:

- teradataml DataFrame if *exec_mode* is "IN-DB"
- Pandas DataFrame if *exec_mode* is "LOCAL".

The method also accepts the same arguments that Script accepts, except that with *returns* is optional and the method does not accept *data*, and accepts exactly one of *data_hash_column* and *data_partition_column*. When *returns* is not provided, the method assumes that the function's output has the columns with the same names and types as the input teradataml DataFrame.

Example Prerequisite

The examples use the 'admissions_train' dataset, calculates the average 'gpa' per partition based on the value in 'admitted' column.

- Load the example data.

```
>>> load_example_data("dataframe", "admissions_train")
```

- Create a DataFrame.

```
>>> df = DataFrame('admissions_train')

>>> print(df)
   masters  gpa  stats programming  admitted
id
5         no  3.44   Novice      Novice         0
34        yes  3.85  Advanced  Beginner         0
13        no  4.00  Advanced   Novice         1
40        yes  3.95   Novice  Beginner         0
22        yes  3.46   Novice  Beginner         0
19        yes  1.98  Advanced  Advanced         0
36        no  3.00  Advanced   Novice         0
15        yes  4.00  Advanced  Advanced         1
7         yes  2.33   Novice   Novice         1
17        no  3.83  Advanced  Advanced         1
```

Example 1: Create a user defined function to calculate the average 'gpa', by reading data in chunks

In this example, the function accepts a TextFileReader object to iterate on data in chunks. The return type of the function is a numpy ndarray.

1. Load the module.

```
>>> from numpy import asarray
```

2. Create a user defined function.

```
>>> def grouped_gpa_avg_iter(rows):
    admitted = None
    row_count = 0
    gpa = 0

    for chunk in rows:
        for _, row in chunk.iterrows():
            row_count += 1
            gpa += row['gpa']
            if admitted is None:
                admitted = row['admitted']

    if row_count > 0:
        return asarray([admitted, gpa/row_count])
```

3. Apply the user defined function to the DataFrame.

```
>>> from teradatasqlalchemy.types import INTEGER, FLOAT

>>> avg_gpa_by_admitted = df.map_partition(grouped_gpa_avg_iter,
                                           returns =
OrderedDict([('admitted', INTEGER()),
('avg_gpa', FLOAT())]),
                                           data_partition_column
= 'admitted')
```

4. Print the result.

```
>>> print(avg_gpa_by_admitted)
      avg_gpa
admitted
1      3.533462
0      3.557143
```

Example 2: Create a user defined function to calculate the average 'gpa', by reading data into a pandas DataFrame

In this example, the data is read at once into a Pandas DataFrame. The function accepts a TextFileReader object to iterate on data in chunks. The return type of the function is a Pandas Series.

1. Create a user defined function.

```
>>> def grouped_gpa_avg(rows):
    pdf = rows.read()
    if pdf.shape[0] > 0:
        return pdf[['admitted', 'gpa']].mean()
```

2. Apply the user defined function to the DataFrame.

```
>>> avg_gpa_pdf = df.map_partition(grouped_gpa_avg,
                                   returns = OrderedDict([('admitted',
INTEGER()),('avg_gpa', FLOAT())]),
                                   data_partition_column = 'admitted')
```

3. Print the result.

```
>>> print(avg_gpa_pdf)
      avg_gpa
admitted
1      3.533462
0      3.557143
```

Example 3: Use a lambda function to achieve the same result

In this example, the function accepts an accept an iterator (TextFileReader object) and returns the result which is of type Pandas Series.

1. Apply the user defined function with a lambda notation.

```
>>> avg_gpa_pdf_lambda = df.map_partition(lambda
rows: grouped_gpa_avg(rows),
returns =
OrderedDict([('admitted', INTEGER()),
('avg_gpa', FLOAT())]),
data_partition_column = 'admitted')
```

2. Print the result.

```
>>> print(avg_gpa_pdf_lambda)
      avg_gpa
admitted
0          3.557143
1          3.533462
```

Example 4: Use a function that returns the input data

In this example, the function accepts an iterator (TextFileReader object) and returns the result which is of type Pandas DataFrame.

1. Create a user defined function.

```
>>> def echo(rows):
    pdf = rows.read()
    if pdf is not None:
        return pdf
```

2. Apply the user defined function.

```
>>> echo_out = df.map_partition(echo, data_partition_column = 'admitted')
```

3. Print the result.

```
>>> print(echo_out)
  masters  gpa  stats programming  admitted
id
15    yes  4.00  Advanced    Advanced         1
7     yes  2.33   Novice     Novice         1
22    yes  3.46   Novice    Beginner         0
17     no  3.83  Advanced    Advanced         1
```

| | | | | | |
|----|-----|------|----------|----------|---|
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 40 | yes | 3.95 | Novice | Beginner | 0 |

apply Method

Use the `apply()` method to apply a user defined function (UDF) to each row in a `teradataml` DataFrame, leveraging APPLY table operator of Open Analytics Framework.

Note:

- The function requires `dill` package with the same version in both remote environment and local environment.
 - Teradata recommends to use the same Python version in both remote and local environment.
-

Required arguments:

- *user_function*: Specifies the user defined function to apply to each row in the `teradataml` DataFrame.

Note:

- The function can be either a lambda function, a regular Python function, or an object of `functools.partial`.
- The first argument (positional) of the UDF must be a row in a pandas DataFrame corresponding to the teradataml DataFrame to which it is to be applied.
- A non-lambda function can be passed only when the UDF does not accept any arguments other than the mandatory input, which is the input row.

A user can also use `functools.partial` and lambda functions for the same, which are especially handy in the following scenarios:

- There is a need to pass positional and/or keyword arguments (lambda)
- There is a need to pass keyword arguments only (`functool.partial`).
- The return type of the UDF must be one of the following:
 - `numpy ndarray`
When one-dimensional, having the same number of values as output columns.
When two-dimensional, every array contained in the outer array having the same number of values as output columns.
 - `pandas Series`
 - `pandas DataFrame`
- To use the `apply` method on the UDF, packages `dill` and `pandas` must be installed in remote user environment using `install_lib` function of `UserEnv` class.

- *env_name*: Specifies the name of the remote user environment or an object of class `UserEnv`.

Optional arguments:

- *exec_mode*: Specifies the mode of execution for the UDF.
Default value is 'REMOTE'.
- *chunk_size*: Specifies the number of rows to be read in a chunk in each iteration using an iterator to apply the UDF to each row in the chunk.
The value passed to this argument affects the performance and the memory utilized by the function.
Default value is 1000.
- *returns*: Specifies output column definition corresponding to the output of *user_function*.
When not specified, the function assumes that the names and types of the output columns are the same as those of the input.
- *delimiter*: Specifies a delimiter to use when reading columns from a row and writing result columns.
The default value is comma (',').

Note:

- This argument cannot be the same as *quotechar* argument.
- This argument cannot be a newline character ('\n').

- *quotechar*: Specifies a character that forces all input and output of the user function to be quoted using this specified character.

Using this argument enables the Analytics Database to distinguish between NULL fields and empty strings. That is, a string with length zero is quoted, while NULL fields are not.

If this character is found in the data, it will be escaped by a second quote character.

Note:

- This argument cannot be the same as *delimiter* argument.
- This argument cannot be a newline character ('\n').

- *data_partition_column*: Specifies Partition By columns for data.

Values to this argument can be provided as a list, if multiple columns are used for partition.

If there is no *data_partition_column*, then the entire result set delivered by the function, constitutes a single group or partition.

Note:

- *data_partition_column* can not be specified along with *data_hash_column*.
- *data_partition_column* can not be specified along with "*is_local_order* = True".

- *data_hash_column*: Specifies the column to be used for hashing.

The rows in the input data are redistributed to AMPs based on the hash value of the column specified.

If there is no *data_hash_column*, then the entire result set, delivered by the function, constitutes a single group or partition.

Note:

- *data_hash_column* can not be specified along with *data_partition_column*.
- *data_hash_column* can not be specified along with "*is_local_order*=False" and *data_order_column*.

- *data_order_column*: Specifies the Order By column for *data*. Values to this argument can be provided as a list, if multiple columns are used for ordering.

This argument can be used no matter *is_local_order* is set to 'True' or 'False'.

Note:

data_order_column can not be specified along with *data_hash_column*.

- *is_local_order*: Specifies a boolean value to determine whether the input data is to be ordered locally or not.

data_order_column with *is_local_order* set to 'False' specifies the order in which the values in a group, or partition, are sorted.

Default value is 'False'. When set to 'True', qualified rows on each AMP are ordered in preparation to be input to a table function.

Note:

This argument is ignored, if *data_order_column* is None, .

Note:

When *is_local_order* is set to 'True', *data_order_column* should be specified, and the columns specified in *data_order_column* are used for local ordering.

- *sort_ascending*: Specifies a boolean value to determine if the result set is to be sorted on the *data_order_column* column in ascending or descending order.
 - When set to the default value 'True', data is sorted in ascending order.
 - When set to 'False', data is sorted in descending order.

Note:

This argument is ignored, if *data_order_column* is None.

- *nulls_first*: Specifies a boolean value to determine whether NULLS are listed first or last during ordering.

NULLS are listed first when this argument is set to default value 'True', and listed last when set to 'False'.

Note:

This argument is ignored, if *data_order_column* is None.

- *style*: Specifies how input is passed to and output is generated by the *apply_command* respectively. Default value is 'csv', and this is the only value supported for this argument.

Examples Setup

- Create a Python 3.7.9 environment.

```
>>> env = create_env('testenv', 'python_3.7.9', 'Test environment')
User environment testenv created.
```

- Install packages dill and pandas in remote user environment.

```
>>> env.install_lib(['pandas','dill'])
Request to install libraries initiated successfully in the remote user
environment demo_env. Check the status using status() with the claim
id 'ef255030-1be2-4d4a-9d47-12cd4365a003'.
```

- Check the status on the installation.

```
>>> env.status('ef255030-1be2-4d4a-9d47-12cd4365a003')
              Claim Id      File/Libs  Method Name
Stage          Timestamp Additional Details
0  ef255030-1be2-4d4a-9d47-12cd4365a003  pandas, dill  install_lib
Started  2022-08-04T04:27:56Z
1  ef255030-1be2-4d4a-9d47-12cd4365a003  pandas, dill  install_lib
Finished 2022-08-04T04:29:12Z
```

- Load example dataset 'admission_train'.

```
>>> load_example_data("dataframe", "admissions_train")
```

```
>>> df = DataFrame('admissions_train')
>>> print(df)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 36 | no | 3.00 | Advanced | Novice | 0 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

The following examples uses the 'admissions_train' dataset, to increase the 'gpa' by a given percentage.

Example 1: Create a user defined function to increase the 'gpa' by a given percentage**Note:**

The input to and the output # from the function is a pandas Series object.

- Create a user defined function.

```
>>> def increase_gpa(row, p=20):
    row['gpa'] = row['gpa'] + row['gpa'] * p/100
    return row
```

- Apply the user defined function to the DataFrame.

Note:

Since the output of the user defined function expects the same columns with the same types, the 'returns' argument is skipped.

```
>>> increase_gpa_20 = df.apply(increase_gpa, env_name='testenv')
```

- Print the result.

```
>>> print(increase_gpa_20)
   masters  gpa  stats programming  admitted
id
22    yes  4.152  Novice    Beginner         0
36     no  3.600  Advanced     Novice         0
15    yes  4.800  Advanced     Advanced        1
38    yes  3.180  Advanced     Beginner        1
5     no  4.128  Novice     Novice         0
17     no  4.596  Advanced     Advanced        1
34    yes  4.620  Advanced     Beginner        0
13     no  4.800  Advanced     Novice         1
26    yes  4.284  Advanced     Advanced        1
19    yes  2.376  Advanced     Advanced        0
```

Example 2: Use the same user defined function with a lambda notation to pass the percentage, 'p = 40'

- Apply the user defined function to the DataFrame with a lambda notation.

```
>>> increase_gpa_40 = df.apply(lambda row: increase_gpa(row, p =
40),
                               env_name='testenv')
```

- Print the result.

```
>>> print(increase_gpa_40)
   masters    gpa    stats programming  admitted
id
22      yes  4.844   Novice    Beginner         0
36      no  4.200  Advanced     Novice         0
15      yes  5.600  Advanced  Advanced         1
38      yes  3.710  Advanced  Beginner         1
5       no  4.816   Novice    Novice         0
17      no  5.362  Advanced  Advanced         1
34      yes  5.390  Advanced  Beginner         0
13      no  5.600  Advanced     Novice         1
26      yes  4.998  Advanced  Advanced         1
19      yes  2.772  Advanced  Advanced         0
```

Example 3: Use the same user defined function with `functools.partial` to pass the percentage, 'p = 50'

- Load required library.

```
>>> from functools import partial
```

- Apply the user defined function to the DataFrame with `functools.partial`.

```
>>> increase_gpa_50 = df.apply(partial(increase_gpa, p = 50),
                               env_name='testenv')
```

- Print the result.

```
>>> print(increase_gpa_50)
   masters    gpa    stats programming  admitted
id
13      no  6.000  Advanced     Novice         1
26      yes  5.355  Advanced  Advanced         1
5       no  5.160   Novice    Novice         0
19      yes  2.970  Advanced  Advanced         0
15      yes  6.000  Advanced  Advanced         1
40      yes  5.925   Novice  Beginner         0
7       yes  3.495   Novice    Novice         1
22      yes  5.190   Novice  Beginner         0
36      no  4.500  Advanced     Novice         0
38      yes  3.975  Advanced  Beginner         1
```

Example 4: Use a lambda function to double 'gpa', and return numpy ndarray

- Load required library.

```
>>> from numpy import asarray
```

- Create a lambda function.

```
>>> inc_gpa_lambda = lambda row, p=20: asarray([row['id'],
                                                row['masters'],
                                                row['gpa'] + row['gpa'] * p/100,
                                                row['stats'],
                                                row['programming'],
                                                row['admitted']])
```

- Apply the lambda function to the DataFrame.

```
>>> increase_gpa_100 = df.apply(lambda row: inc_gpa_lambda(row, p=100),
                                env_name='testenv')
```

- Print the result.

```
>>> print(increase_gpa_100)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 13 | no | 8.00 | Advanced | Novice | 1 |
| 26 | yes | 7.14 | Advanced | Advanced | 1 |
| 5 | no | 6.88 | Novice | Novice | 0 |
| 19 | yes | 3.96 | Advanced | Advanced | 0 |
| 15 | yes | 8.00 | Advanced | Advanced | 1 |
| 40 | yes | 7.90 | Novice | Beginner | 0 |
| 7 | yes | 4.66 | Novice | Novice | 1 |
| 22 | yes | 6.92 | Novice | Beginner | 0 |
| 36 | no | 6.00 | Advanced | Novice | 0 |
| 38 | yes | 5.30 | Advanced | Beginner | 1 |

drop_duplicate()

Use the the drop_duplicate() function to drop duplicate rows from teradataml DataFrame to return distinct values from the DataFrame.

Optional Argument:

- *column_names*: Specifies the names of the columns to drop the duplicate values of, to get the distinct values.

If not specified, all columns in the DataFrame are considered for the operation.

Example Setup

In this example, "admission_train" dataset is used.

```
>>> from teradataml import *

>>> load_example_data("dataframe", "admissions_train")

>>> df = DataFrame("admissions_train")

# Print dataframe.
>>> df
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 40 | yes | 3.95 | Novice | Beginner | 0 |
| 7 | yes | 2.33 | Novice | Novice | 1 |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 36 | no | 3.00 | Advanced | Novice | 0 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |

Example 1: Get the distinct rows of values for the column 'programming'

```
>>> df.drop_duplicate("programming")
```

| | programming |
|---|-------------|
| 0 | Novice |
| 1 | Beginner |
| 2 | Advanced |

Example 2: Get the distinct rows of values for the columns 'programming' and 'admitted'

```
>>> df.drop_duplicate(["programming", "admitted"])
```

| | programming | admitted |
|---|-------------|----------|
| 0 | Beginner | 0 |
| 1 | Advanced | 1 |
| 2 | Beginner | 1 |
| 3 | Advanced | 0 |
| 4 | Novice | 1 |
| 5 | Novice | 0 |

DataFrame Metadata

You can retrieve the metadata of a DataFrame with the following methods and properties.

DataFrame Methods

| DataFrame Methods | Description |
|-------------------------------|--|
| info() Method | Returns a summary of the DataFrame. |
| keys() Method | Returns a list of column names of the DataFrame. |

DataFrame Properties

| DataFrame Property | Description |
|----------------------------------|---|
| columns Property | Returns a list containing column names. |
| dtypes Property | Returns column names and their corresponding types. |
| index Property | Returns the index of the DataFrame, which corresponds to the primary index of the underlying Table or View. |
| shape Property | Returns a tuple representing the dimensionality of the DataFrame. |
| size Property | Returns a value representing the number of elements in the DataFrame. |

Note:

These DataFrame methods and properties do not return a teradataml DataFrame.

info() Method

The `info()` function prints a summary of the DataFrame.

Optional arguments:

- `verbose` specifies whether to print full summary or short summary.
 - If set to `True`, prints full summary.
 - If set to `False`, prints short summary.
- `buf` specifies the writable buffer to send the output to.
By default, the output is sent to `sys.stdout`.
- `max_cols` specifies the maximum number of columns allowed for printing the full summary.
- `null_counts` specifies whether to show the non-null counts. Display the counts if `null_counts` is `True`, otherwise do not display the non-null counts.

Example 1: Print information on DataFrame

This example prints information for DataFrame "sales".


```
>>> df = DataFrame("sales")

>>> df.info()
<class 'teradataml.dataframe.dataframe.DataFrame'>
Data columns (total 6 columns):
accounts          str
Feb               float
Jan              int
Mar              int
Apr              int
datetime         datetime.date
dtypes: str(1), float(1), int(3), datetime.date(1)
```

Example 2: Print a count of non-null values

This example set `null_counts` to `True` for a count of non-null values.

```
>>> df.info(null_counts=True)
<class 'teradataml.dataframe.dataframe.DataFrame'>
Data columns (total 6 columns):
accounts    6 non-null str
Feb         6 non-null float
Jan         4 non-null int
Mar         4 non-null int
Apr         4 non-null int
datetime    6 non-null datetime.date
dtypes: str(1), float(1), int(3), datetime.date(1)
```

Example 3: Print a short summary

This example set `verbose` to `False` for a short summary.

```
>>> df.info(verbose=False)
<class 'teradataml.dataframe.dataframe.DataFrame'>
Data columns (total 6 columns):
dtypes: str(1), float(1), int(3), datetime.date(1)
```

keys() Method

Use the `keys()` method to get a list of column names.

Note:

The `keys()` method is identical to the [columns Property](#).

Example

```
>>> df = DataFrame("sales")

>>> df.keys()
['accounts', 'Feb', 'Jan', 'Mar', 'Apr', 'datetime']
```

DataFrame Properties

columns Property

Use the columns property to get a list of column names.

Note:

The columns property is identical to the [keys\(\) Method](#).

Example

```
>>> df = DataFrame("sales")

>>> df.columns
['accounts', 'Feb', 'Jan', 'Mar', 'Apr', 'datetime']
```

dtypes Property

Use the dtypes property to print the column names and column types of a DataFrame.

Example

```
>>> df = DataFrame("sales")

>>> df.dtypes
accounts      str
Feb           float
Jan           int
Mar           int
Apr           int
datetime    datetime.date
```

index Property

Use the index property to retrieve the index of the teradataml DataFrame, which corresponds to the primary index of the underlying Table or View.

In case the index is explicitly set using the DataFrame.set_index() method or by passing the index_label parameter while creating the teradataml DataFrame, the property will return the value set by the user.

Example Prerequisite

Load the admissions_train dataset and create a teradataml DataFrame out of it.

```
>>> load_example_data("dataframe","admissions_train")
```

```
>>> df = DataFrame("admissions_train")
```

```
>>> df
   masters  gpa  stats programming  admitted
id
5        no  3.44   Novice      Novice         0
3        no  3.70   Novice   Beginner         1
1       yes  3.95  Beginner   Beginner         0
20      yes  3.90  Advanced  Advanced         1
8        no  3.60  Beginner  Advanced         1
25      no  3.96  Advanced  Advanced         1
18      yes  3.81  Advanced  Advanced         1
24      no  1.87  Advanced    Novice         1
26      yes  3.57  Advanced  Advanced         1
38      yes  2.65  Advanced  Beginner         1
```

Example 1: Retrieve the index

```
>>> # Get the index_label
>>> df.index
['id']
```

Example 2: Set the index using the set_index() method and then retrieve the index

Set the index for the teradataml DataFrame using the set_index() method.

```
>>> # Set new index_label
>>> df = df.set_index(['id', 'masters'])
>>> df
           gpa  stats programming  admitted
```

```
id masters
5 no 3.44 Novice Novice 0
3 no 3.70 Novice Beginner 1
1 yes 3.95 Beginner Beginner 0
17 no 3.83 Advanced Advanced 1
13 no 4.00 Advanced Novice 1
32 yes 3.46 Advanced Beginner 0
11 no 3.13 Advanced Advanced 1
9 no 3.82 Advanced Advanced 1
34 yes 3.85 Advanced Beginner 0
24 no 1.87 Advanced Novice 1
```

Retrieve the index.

```
>>> # Get the index_label
>>> df.index
['id', 'masters']
```

shape Property

Use the shape property to retrieve the dimensionality of a teradataml DataFrame.

Example

```
>>> df = DataFrame('sales')
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |

```
>>> df.shape
(6, 6)
```

size Property

Use the size property to retrieve the number of elements in a teradataml DataFrame.

Example

```
>>> df = DataFrame('sales')
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |

```
>>> df.size
36
```

tdtypes Property

Use the `tdtypes` property to print the column names and corresponding teradata sqlalchemy types of a DataFrame.

Example

```
>>> df = DataFrame("sales")

>>> df.tdtypes
accounts    VARCHAR(length=20, charset='LATIN')
Feb                                FLOAT()
Jan                                BIGINT()
Mar                                BIGINT()
Apr                                BIGINT()
datetime                                DATE()
```

Data Rotation**pivot()**

Use the `pivot()` function to rotate data from rows into columns to create easy-to-read DataFrames. The function is useful for reporting purposes, as it allows to aggregate and rotate the data.

Required Arguments:

- *columns*: Specifies the columns or dictionary of columns with distinct column values used for pivoting the data.

This argument is required when keyword arguments are not specified. Otherwise, it is optional.

- When specified as a column, this function automatically extracts the distinct values for the column.

For example: `columns = df.qtr #`, or `columns = [df.qtr, dr.yr]`

Note:

Distinct values 'Q1', 'Q2', 'Q3', and so on, are automatically extracted from column 'qtr' for pivoting.

- When specified as dictionary:
 - Key is a column expression representing a column.
 - Value is a literal or list of literals, that is, values in a column or a teradataml DataFrame with only one column.

Note:

When value is a teradataml DataFrame, the order of output pivoted columns is not guaranteed. In this case, Teradata recommends not specifying *returns* argument, so column names are generated properly according to the order of records.

When dictionary value contains literals,

- If *limit_combinations* is set to False, pivot is done based on each combination of the values specified for each key.
 - Otherwise, combinations are restricted based on the index values of the list.
-

Note:

All values in dictionary should have the same length when *limit_combinations* is set to True.

Example 1: `columns={df.qtr: ['Q1', 'Q2'], df.year: [2001, 2002]}`

In this case, pivot is based on all the combinations of values as specified in the following, for columns 'qtr' and 'yr':

- If *limit_combinations* is set to False: (Q1, 2001), (Q1, 2002), (Q2, 2001), and (Q2, 2002).
- If *limit_combinations* is set to True: (Q1, 2001) and (Q2, 2002).

Example 2: `columns={df.qtr: quarter_df, df.year: [2001, 2002]}`, where value passed to `df.qtr` is a teradataml DataFrame with one column.

In this cases, function extracts the distinct values from the DataFrame and pivot is based on all the combinations of values or limited combination of values based on argument *limit_combinations*.

Note:

String type values in dictionary are not case sensitive. See the following example 2 for more details.

- *aggfuncs*: Specifies the column aggregate functions to be used for pivoting the data.

For example:

To pivot total of 'sales' column and average of 'cogs' + 'sales' columns in the DataFrame "df", specify this argument as:

```
aggfuncs=[df.sales.sum(), (df.cogs+df.sales).avg()]
```

Optional Arguments:

- *limit_combinations*: Specifies whether to limit the number of combinations when *columns* argument is passed as a dictionary.

Default value is False.

When set to True, function limits the combinations one-to-one based on index of the values in list, hence all dictionary values should be a list of same length or a single literal.

For example: `df.pivot(columns={df.qtr: ['Q1', 'Q2'], df.year: [2001, 2002]}, limit_combinations=True, ...)`

Pivot will be based on columns 'qtr' and 'yr' for values (Q1, 2001) and (Q2, 2002) only.

Note:

This argument is ignored when *columns* is a ColumnExpression or a list of ColumnExpressions.

- *margins*: Specifies the aggregate operation to be performed on output columns.

Aggregate operation to be performed should be passed as dictionary where:

- Key is a string specifying aggregate operation to be performed.
- Value is a tuple or list of tuples specifying the output column names as string. Columns specified in the tuple are used in aggregate operation performed.

For example: If for the year 2001, following three aggregate columns are needed in the output:

- Sum of sales in Q1 and Q2
- Sum of sales in Q2 and Q3
- Average of cogs for first three quarters

margins can be specified as:

```
margins={"SUM": [("Q12001_sales", "Q22001_sales"), ("Q22001_sales", "Q32001_sales")],
"AVG": ("Q12001_cogs", "Q22001_cogs", "Q32001_cogs")}
```

Note:

- Supported aggregate functions are SUM, AVG, MIN and MAX.
- If *returns* is specified, column names for margins are considered from returns clause.

- *returns*: Specifies the custom column names to be returned in the output DataFrame.

If this argument is not specified, function internally generates the output column names.

The number of column names should be same as columns generated by pivot. For example:

- If `columns={df.qtr: ['Q1', 'Q2'], df.year: [2001, 2002]}`, `aggfuncs=[df.sales.sum(), df.cogs.sum()]` and `limit_combination=False`, then number of new columns in output will be: $\text{len(aggfuncs)} * \text{len(columns[df.year])} * \text{len(columns[df.qtr])}$ or $2 * 2 * 2 = 8$.

Hence, *returns* should be list of string with 8 elements.

- If in the previous example, `limit_combination=True`, then number of new columns in output will be: $\text{len(aggfuncs)} * (\text{len(columns[df.year]) OR len(columns[df.qtr])})$ or $2 * 2 = 4$.

Hence, *returns* should be list of string with 4 elements.

Note:

All columns including columns in DataFrame which do not participate in pivot must be specified if *all_columns* is set to True.

- *all_columns*: Specifies whether *returns* argument should include the names of only pivot columns or all columns.
 - When set to True, all columns including columns in DataFrame which do not participate in pivot must be specified.
 - When set to False, only output columns excluding columns in DataFrame which do not participate in pivot must be specified.

This is the default value.

- *kwargs*: Specifies the keyword argument to accept column name and column values as named arguments.

For example: `col1=df.year, col1_value=2001, col2=df.qtr, col2_value=['Q1', 'Q2', 'Q3']`

Note:

- Either use *columns* argument or keyword arguments.
- Format for column name arguments should be `col1, col2,... colN` with values of type `ColumnExpression`.
- Format for column value argument should be `col1_value, col2_value,... colN_value`.

Example Setup

```
# Create a teradataml DataFrame.
>>> load_example_data("teradataml", "star1")
>>> df = DataFrame("star1")
```

Example 1: Pivot data using all values in column 'qtr' and aggregate using sum of column 'sales'

```
>>> pivot_df = df.pivot(columns=df.qtr,
                          aggfuncs=df.sales.sum())
```

```
>>> print(pivot_df)
```

| | country | state | yr | cogs | sum_sales_q2 | sum_sales_q1 | sum_sales_q3 |
|---|---------|-------|------|------|--------------|--------------|--------------|
| 0 | USA | NY | 2001 | 25.0 | NaN | 45.0 | NaN |
| 1 | CANADA | ON | 2001 | 0.0 | 10.0 | NaN | NaN |
| 2 | CANADA | BC | 2001 | 0.0 | NaN | NaN | 10.0 |
| 3 | USA | CA | 2002 | 20.0 | 50.0 | NaN | NaN |
| 4 | USA | CA | 2002 | 15.0 | NaN | 30.0 | NaN |

Example 2: Pivot data using columns 'yr' and 'qtr', aggregate using sum of 'sales' and median of 'cogs' column

Limit combination of 'q1' with '2001' and 'q2' with '2002'.

```
>>> pivot_df = df.pivot(columns={df.yr: [2001, 2002], df.qtr: ['Q1', 'Q2']},
                          aggfuncs=[df.sales.sum(), df.cogs.median()],
                          limit_combinations=True)
```

```
>>> print(pivot_df)
```

| | country | state | sum_sales_2001_q1 | median_cogs_2001_q1 | sum_sales_2002_q2 | median_cogs_2002_q2 |
|---|---------|-------|-------------------|---------------------|-------------------|---------------------|
| 0 | CANADA | ON | NaN | NaN | NaN | NaN |
| 1 | USA | CA | NaN | NaN | NaN | NaN |
| 2 | USA | NY | 45.0 | 25.0 | NaN | NaN |
| 3 | CANADA | BC | NaN | NaN | NaN | NaN |

Example 3: Get all possible column combinations

Get all possible column combinations when pivoting using 'yr' and 'qtr', aggregation using 'sales' and 'cogs' column.

```
>>> pivot_df = df.pivot(columns={df.yr: [2001, 2002], df.qtr: ['q1', 'q2']},
                          aggfuncs=[df.sales.sum(), df.cogs.max()])
```

```
>>> print(pivot_df)
   country state  sum_sales_2001_q1  max_cogs_2001_q1
sum_sales_2001_q2  max_cogs_2001_q2  sum_sales_2002_q1  max_cogs_2002_q1
sum_sales_2002_q2  max_cogs_2002_q2
0  CANADA    BC          NaN          NaN          NaN
NaN          NaN          NaN          NaN          NaN
NaN          NaN          NaN          NaN          NaN
1  CANADA    ON          NaN          NaN          NaN
10.0          0.0          NaN          NaN          NaN
NaN          NaN          NaN          NaN          NaN
2    USA    NY          45.0          25.0          NaN
NaN          NaN          NaN          NaN          NaN
NaN          NaN          NaN          NaN          NaN
3    USA    CA          NaN          NaN          NaN
NaN          NaN          30.0          15.0          NaN
50.0          20.0          NaN          NaN          NaN
```

Example 4: Customize the name of returned columns using *returns* argument

```
>>> pivot_df = df.pivot(columns={df.yr:2001, df.qtr:['Q1', 'Q2']},
                          aggfuncs=[df.sales.sum(), (2*df.sales-1).max()],
                          returns=["Q1_2001_total_sales", "Q1_2001_total_cogs",
                                   "Q2_2001_total_sales", "Q2_2001_total_cogs"])
```

```
>>> print(pivot_df)
   country state  cogs  Q1_2001_total_sales  Q1_2001_total_cogs
Q2_2001_total_sales  Q2_2001_total_cogs
0    USA    NY  25.0          45.0          89.0
NaN          NaN          NaN          NaN          NaN
1  CANADA    ON   0.0          NaN          NaN
10.0          19.0          NaN          NaN          NaN
2  CANADA    BC   0.0          NaN          NaN
NaN          NaN          NaN          NaN          NaN
3    USA    CA  20.0          NaN          NaN
NaN          NaN          NaN          NaN          NaN
```

| | | | | | |
|-----|-----|----|------|-----|-----|
| 4 | USA | CA | 15.0 | NaN | NaN |
| NaN | | | NaN | | |

Example 5: Customize the name of all output columns using *returns* and *all_columns* arguments

```
>>> pivot_df = df.pivot(columns={df.yr:2001, df.qtr:['Q1', 'Q2']},
                        aggfuncs=[df.sales.avg(), df.cogs.median()],
                        returns=["con", "st",
                                "Q1_2001_total_sales", "Q1_2001_total_cogs",
                                "Q2_2001_total_sales", "Q2_2001_total_cogs"],
                        all_columns=True)
```

```
>>> print(pivot_df)
      con st Q1_2001_total_sales Q1_2001_total_cogs
Q2_2001_total_sales Q2_2001_total_cogs
0  CANADA  ON                NaN                NaN
10.0                0.0
1     USA   CA                NaN                NaN
NaN                NaN
2     USA   NY                45.0                25.0
NaN                NaN
3  CANADA  BC                NaN                NaN
NaN                NaN
```

Example 6: Use keyword arguments to specify columns and values instead of *columns* argument

Note:

Since *returns* is not specified, column names are generated by the function.

```
>>> pivot_df = df.pivot(aggfuncs=[df.sales.avg(), ((2*df.sales)+
(df.cogs/2)).min()],
                        col1=df.yr,
                        col1_values=2001,
                        col2=df.qtr,
                        col2_values=['Q1', 'Q2'])
```

```
>>> print(pivot_df)
country state avg_sales_2001_q1 min_sales_2_cogs_2_2001_q1
avg_sales_2001_q2 min_sales_2_cogs_2_2001_q2
0  CANADA   BC                NaN                NaN
NaN                NaN
```

| | | | | |
|------|--------|----|------|-------|
| 1 | CANADA | ON | NaN | NaN |
| 10.0 | | | 20.0 | |
| 2 | USA | NY | 45.0 | 102.5 |
| NaN | | | NaN | |
| 3 | USA | CA | NaN | NaN |
| NaN | | | NaN | |

Example 7: Find the median sales and mean cogs for first three quarters of year 2001 using *margins*

```
>>> pivot_df = df.pivot(columns={df.qtr: ['q1', 'q2', 'q3'], df.yr: [2001]},
                          aggfuncs=df.sales.median(),
                          margins={"SUM":
[("median_sales_q1_2001", "median_sales_q2_2001"),
("median_sales_q3_2001", "median_sales_q2_2001")],
                                "AVG":
[("median_sales_q1_2001", "median_sales_q2_2001"),
("median_sales_q2_2001", "median_sales_q1_2001")]}])
```

```
>>> print(pivot_df)
country state cogs median_sales_q1_2001 median_sales_q2_2001
median_sales_q3_2001 sum_median_sales_q1_2001 median_sales_q2_2001
sum_median_sales_q3_2001 median_sales_q2_2001
avg_median_sales_q1_2001 median_sales_q2_2001
avg_median_sales_q2_2001 median_sales_q1_2001
0 CANADA BC 0.0 NaN NaN
10.0 NaN
10.0 NaN
NaN
1 USA NY 25.0 45.0 NaN
NaN 45.0
NaN 45.0
45.0
2 CANADA ON 0.0 NaN 10.0
NaN 10.0
10.0 10.0
10.0
3 USA CA 20.0 NaN NaN
NaN NaN
NaN NaN
NaN
4 USA CA 15.0 NaN NaN
```

| | |
|-----|-----|
| NaN | NaN |
| NaN | NaN |
| NaN | |

Example 8: Find the min of sales and average of cogs for quarter 'Q1', 'Q2' of year '2001'

Name the aggregated column as 'margins'.

```
>>> pivot_df = df.pivot(columns={df.qtr: ['q1', 'q2'], df.yr: [2001]},
                        aggfuncs=[df.cogs.min(), df.sales.avg()],
                        margins={"MIN":
("cogs_min_q12001", "sales_var_samp_q22001")},
                        all_columns=True,
                        returns=["con",
                                "st",
                                "cogs_min_q12001",
                                "sales_var_samp_q12001",
                                "cogs_min_q22001",
                                "sales_var_samp_q22001",
                                "margins"])
```

```
>>> print(pivot_df)
   con st cogs_min_q12001 sales_var_samp_q12001 cogs_min_q22001
sales_var_samp_q22001 margins
0  CANADA BC           NaN                    NaN
NaN                    NaN                    NaN
1  CANADA ON           NaN                    NaN
0.0                    10.0                    10.0
2    USA  NY           25.0                    45.0
NaN                    NaN                    25.0
3    USA  CA           NaN                    NaN
NaN                    NaN                    NaN
```

Example 9: Specify a teradataml DataFrame with one column for values of 'qtr' column, and then pivot the data

Note:

This allows user to implicitly use all the distinct values for the pivot operation and not specify those explicitly.

```
>>> quarters_df = df.drop(columns=['country', 'state', 'yr', 'sales', 'cogs'])
```

```
>>> print(quarters_df)
   qtr
0  Q1
1  Q2
2  Q3
3  Q2
4  Q1

>>> pivot_df = df.pivot(columns={df.qtr: quarters_df},
                          aggfuncs=[df.sales.sum(), df.cogs.avg()])

>>> print(pivot_df)
   country state   yr  sum_sales_q2  avg_cogs_q2  sum_sales_q1  avg_cogs_q1
sum_sales_q3  avg_cogs_q3
0  CANADA    ON  2001         10.0          0.0          NaN
NaN          NaN          NaN
1  CANADA    BC  2001          NaN          NaN          NaN
NaN          10.0          0.0
2    USA    NY  2001          NaN          NaN          45.0
25.0          NaN          NaN
3    USA    CA  2002          50.0          20.0          30.0
15.0          NaN          NaN
```

unpivot()

Use the `unpivot()` function to rotate data from columns into rows to create easy-to-read DataFrames.

Required Arguments:

- **columns:** Specifies the dictionary of columns with distinct column values used for unpivoting the data.

This argument is required when keyword arguments are not specified. Otherwise, it is optional.

- Key is a column expression or tuple of column expressions.
- Value is a literal to be taken by column when transposed into *transpose_column*. If not specified, value is generated by the function based on column names.
- Number of columns specified in each key should be equal.

For example:

```
columns={(df.Q101Sales, df.Q101Cogs): "2001_Q1",
         (df.Q201Sales, df.Q201Cogs): None,
         (df.Q301Sales, df.Q301Cogs): "2001_Q3"},
transpose_column="yr_qtr",
```

This example does the following:

- Transposes columns 'Q101Sales' and 'Q101Cogs' into a row where 'yr_qtr' column value is '2001_Q1';
- Transposes columns 'Q301Sales' and 'Q301Cogs' into a row with '2001_Q3' value in 'yr_qtr' column;
- For columns 'Q201Sales' and 'Q201Cogs', value of 'yr_qtr' column is function generated.
- *transpose_column*: Specifies the name of the column in the output DataFrame, which holds the data for columns specified in the keys of *columns* argument.
- *measure_columns*: Specifies the names of output columns to unpivot the data in the columns specified in the *columns* argument.

Note:

- The number of columns specified in *measure_columns* should be equal to the number of columns specified in each key of *columns*.
- One exception for this requirement is when all the columns is to be unpivoted into a single column. In such case, *measure_columns* should be specified as a string or list of string with one value.

Optional Arguments:

- *exclude_nulls*: Specifies whether to exclude NULL (None) values while unpivoting the data. When set to True, excludes NULL (None), otherwise includes NULL.

Default value is True.

- *returns*: Specifies the custom column names to be returned in the output DataFrame.

Note:

- The number of column names should be equal to the value in the *measure_column* argument plus one.
- All columns including columns in DataFrame which do not participate in unpivot must be specified if *all_columns* is set to True.

- *all_columns*: Specifies whether *returns* argument should include the names of only unpivot columns or all columns.
 - When set to True, all columns including columns in DataFrame which do not participate in unpivot must be specified.
 - When set to False, which is the default value, only output columns excluding columns in DataFrame which do not participate in unpivot must be specified.
- *kwargs*: Specifies the keyword argument to accept column name and column values as named arguments.

For example:

```
col1=df.year, col1_value=2001, col2=df.qtr, col2_value='Q1'
```

Or

```
col1=(df.Q101Sales, df.Q101Cogs), col1_value="2001_Q1",
col2=(df.Q201Sales, df.Q201Cogs), col2_value=None,
col3=(df.Q301Sales, df.Q301Cogs), col3_value="2001_Q3"
```

Note:

- Either use *columns* argument or keyword arguments.
- Format for column name arguments should be col1, col2,... colN with values of type ColumnExpression.
- Format for column value argument should be col1_value, col2_value,... colN_value.

Example Setup

```
# Create a teradataml DataFrame.
>>> load_example_data("teradataml", "star1")
>>> df = DataFrame("star1")

>>> df = df.pivot(columns={df.qtr: ["Q1", "Q2", "Q3"], df.yr: ["2001"]},
                    aggfuncs=[df.sales.sum(), df.cogs.sum()],
                    returns=["Q101Sales", "Q201Sales", "Q301Sales",
                           "Q101Cogs", "Q201Cogs", "Q301Cogs"])
```

```
>>> print(df)
```

| | country | state | Q101Sales | Q201Sales | Q301Sales | Q101Cogs | Q201Cogs | Q301Cogs |
|---|---------|-------|-----------|-----------|-----------|----------|----------|----------|
| 0 | CANADA | BC | NaN | NaN | NaN | NaN | 10.0 | 0.0 |
| 1 | CANADA | ON | NaN | NaN | 10.0 | 0.0 | NaN | NaN |
| 2 | USA | NY | 45.0 | 25.0 | NaN | NaN | NaN | NaN |
| 3 | USA | CA | NaN | NaN | NaN | NaN | NaN | NaN |

Example 1: Unpivot quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns

This example unpivots quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns using *columns* argument.

```
>>> unpivot_df = df.unpivot(columns={(df.Q101Sales, df.Q101Cogs): "2001_Q1",
                                     (df.Q201Sales, df.Q201Cogs): None,
                                     (df.Q301Sales, df.Q301Cogs): "2001_Q3"},
```



```
transpose_column="yr_qtr",
measure_columns=["sales", "cogs"])
```

```
>>> print(unpivot_df)
  country state      yr_qtr  sales  cogs
0  CANADA   BC  Q201Sales_Q201Cogs   NaN  10.0
1  CANADA   ON      2001_Q1   NaN   0.0
2  CANADA   ON      2001_Q3   10.0   NaN
3  CANADA   BC      2001_Q3   NaN   0.0
4    USA    NY  Q201Sales_Q201Cogs  25.0   NaN
5    USA    NY      2001_Q1  45.0   NaN
```

Example 2: Unpivot 'sales' and 'cogs' into a single column 'sales_cogs'

```
>>> unpivot_df = df.unpivot(columns={(df.Q101Sales, df.Q101Cogs,
                                     df.Q201Sales, df.Q201Cogs,
                                     df.Q301Sales, df.Q301Cogs): None},
                             transpose_column="yr_qtr",
                             measure_columns="sales_cogs")
```

```
>>> print(unpivot_df)
  country state      yr_qtr  sales_cogs
0  CANADA   BC   Q201Cogs      10.0
1  CANADA   ON   Q101Cogs       0.0
2  CANADA   ON  Q301Sales      10.0
3  CANADA   BC   Q301Cogs       0.0
4    USA    NY  Q201Sales      25.0
5    USA    NY  Q101Sales      45.0
```

Example 3: Unpivot quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns

This example unpivots quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns using *keyword* arguments.

```
>>> unpivot_df = df.unpivot(transpose_column="yr_qtr",
                             measure_columns=["sales", "cogs"],
                             col1 = (df.Q101Sales, df.Q101Cogs),
                             col1_value = "Q101",
                             col2 = (df.Q201Sales, df.Q201Cogs),
                             col2_value = None,
                             col3 = (df.Q301Sales, df.Q301Cogs),
                             col3_value = "Q103")
```

```
>>> print(unpivot_df)
  country state      yr_qtr  sales  cogs
0  CANADA   BC  Q201Sales_Q201Cogs   NaN  10.0
1  CANADA   ON           Q101   NaN   0.0
2  CANADA   ON           Q103  10.0   NaN
3  CANADA   BC           Q103   NaN   0.0
4    USA    NY  Q201Sales_Q201Cogs  25.0   NaN
5    USA    NY           Q101  45.0   NaN
```

Example 4: Unpivot quarterly sales and quarterly cogs data and rename column of the output using one argument

This example unpivots quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns, and rename column of the output using the *returns* argument.

```
>>> unpivot_df = df.unpivot(columns={(df.Q101Sales, df.Q101Cogs): "Q101",
                                     (df.Q201Sales, df.Q201Cogs): None,
                                     (df.Q301Sales, df.Q301Cogs): "Q301"},
                           transpose_column="yr_qtr",
                           measure_columns=["sales", "cogs"],
                           returns=["year_quarter", "sales_data", "cogs_data"])
```

```
>>> print(unpivot_df)
  country state      year_quarter  sales_data  cogs_data
0  CANADA   BC  Q201Sales_Q201Cogs   NaN        10.0
1  CANADA   ON           Q101   NaN          0.0
2  CANADA   ON           Q301  10.0         NaN
3  CANADA   BC           Q301   NaN          0.0
4    USA    NY  Q201Sales_Q201Cogs  25.0         NaN
5    USA    NY           Q101  45.0         NaN
```

Example 5: Unpivot quarterly sales and quarterly cogs data and rename each column of the output using two arguments

This example unpivots quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns, and rename each column of the output using *returns* and *all_columns* arguments.

```
>>> unpivot_df = df.unpivot(columns={(df.Q101Sales, df.Q101Cogs): "Q101",
                                     (df.Q201Sales, df.Q201Cogs): None,
                                     (df.Q301Sales, df.Q301Cogs): "Q301"},
                           transpose_column="yr_qtr",
                           measure_columns=["sales", "cogs"],
                           returns=["con", "st", "year_quarter",
```

```
"sales_data", "cogs_data"],
all_columns=True)
```

```
>>> print(unpivot_df)
   con st   year_quarter sales_data cogs_data
0  CANADA BC  Q201Sales_Q201Cogs      NaN      10.0
1  CANADA ON              Q101      NaN      0.0
2  CANADA ON              Q301      10.0      NaN
3  CANADA BC              Q301      NaN      0.0
4    USA NY  Q201Sales_Q201Cogs      25.0      NaN
5    USA NY              Q101      45.0      NaN
```

Example 6: Unpivot quarterly sales data to 'sales' column and quarterly cogs data to 'cogs' columns, including NULLs

```
>>> unpivot_df = df.unpivot(columns={(df.Q101Sales, df.Q101Cogs): "2001_Q1",
                                     (df.Q201Sales, df.Q201Cogs): None,
                                     (df.Q301Sales, df.Q301Cogs): "2001_Q3"},
                             transpose_column="yr_qtr",
                             measure_columns=["sales", "cogs"],
                             exclude_nulls=False)
```

```
>>> print(unpivot_df)
   country state   yr_qtr sales cogs
0    USA    CA      2001_Q3   NaN  NaN
1    USA    NY  Q201Sales_Q201Cogs  25.0  NaN
2    USA    NY      2001_Q3   NaN  NaN
3  CANADA    BC      2001_Q1   NaN  NaN
4  CANADA    BC      2001_Q3   NaN   0.0
5  CANADA    ON      2001_Q1   NaN   0.0
6  CANADA    BC  Q201Sales_Q201Cogs   NaN  10.0
7    USA    NY      2001_Q1  45.0  NaN
8    USA    CA  Q201Sales_Q201Cogs   NaN  NaN
9    USA    CA      2001_Q1   NaN  NaN
```

Saving teradataml DataFrames to Vantage

to_sql() DataFrame Method

Use the `to_sql()` DataFrame method to create a table in Vantage based on a teradataml DataFrame.

The function takes a table name as an argument, and generates DDL and DML queries that creates a table in Vantage. You can also specify the name of the schema in the database to write the table to. If no schema name is specified, the default database schema is used.

Required argument:

- *table_name*: Specifies the name of the table to be created in Vantage.

Optional arguments:

- *schema_name*: Specifies the name of the SQL schema in Vantage to write to. The default value is None, which means the default Vantage schema.

Note:

schema_name is ignored when *temporary* is True.

- *if_exists*: Specifies the action to take when table already exists in the database. The possible values are:
 - 'fail': If table exists, do nothing;
 - 'replace': If table exists, drop it, recreate it, and insert data;
 - 'append': If table exists, insert data. Create if does not exist. This is the default value.

Note:

Replacing a table with the content of a teradataml DataFrame based on the same underlying table is not supported.

- *primary_index*: Creates Vantage tables with Primary index column. The value can be a single column name or a list of columns names. when *primary_index* is not specified, "No Primary Index Vantage tables are created.
- *temporary*: Specifies whether to create a permanent table or volatile table. Volatile tables only exist for the lifetime of the session. Volatile tables automatically go away when the session ends. The possible values are:
 - True: Creates a volatile table;
 - False: Creates a permanent table. This is the default value.
- *types*: Specifies required data types for requested columns to be saved in Vantage.
- *primary_time_index_name*: Specifies a name for the Primary Time Index (PTI) when the table to be created must be a PTI table.

Note:

This argument is not required or used when the table to be created is not a PTI table. It is ignored if specified without the *timecode_column*.

- *timecode_column*: Specifies the column in the DataFrame that reflects the form of the timestamp data in the time series.

Note:

This argument is required when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. If this argument is specified, *primary_index* is ignored.

- *timezero_date*: Specifies the earliest time series data that the PTI will accept; a date that precedes the earliest date in the time series data. Default value is DATE '1970-01-01'.

Note:

This argument is used when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. It is ignored if specified without the *timecode_column*.

- *timebucket_duration*: Specifies a duration that serves to break up the time continuum in the time series data into discrete groups or buckets.

Note:

This argument is required if *columns_list* is not specified or is empty. It is used when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. It is ignored if specified without the *timecode_column*.

- *column_list*: Specifies a list of one or more PTI table column names.

Note:

This argument is required if *timebucket_duration* not specified. It is used when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. It is ignored if specified without the *timecode_column*.

- *sequence_column*: Specifies the column of type Integer containing the unique identifier for time series data reading when they are not unique in time.
 - When specified, implies SEQUENCED, meaning more than one reading from the same sensor may have the same timestamp. This column will be the TD_SEQNO column in the table created.
 - When not specified, implies NONSEQUENCED, meaning there is only one sensor reading per timestamp. This is the default.

Note:

This argument is used when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. It will be ignored if specified without the *timecode_column*.

- `seq_max`: Specifies the maximum number of sensor data rows that can have the same timestamp. Can be used when 'sequenced' is True.

Note:

This argument is used when the DataFrame must be saved as a PTI table. This argument is not required or used when the table to be created is not a PTI table. It will be ignored if specified without the `timecode_column`.

- `set_table`: Specifies a flag to determine whether to create a SET or a MULTiset table.
 - When True, a SET table is created.
 - When False, a MULTiset table is created. This is the default value.

Note:

- Specifying `set_table=True` also requires specifying `primary_index` or `timecode_column`.
- Creating SET table (`set_table=True`) may result in loss of duplicate rows.
- This argument has no effect if the table already exists and `if_exists='append'`.

Example 1: Create a NOPI table with selected columns from existing table

This example uses the `to_sql()` function to create a new table "sales_df1" based on the existing table "sales" in Teradata Vantage. The new table "sales_df1" includes only the columns "accounts", "Jan", and "Feb".

- Create a teradataml DataFrame "df" from the existing table "sales".

```
df = DataFrame("sales")
```

```
>>> df
      Feb   Jan   Mar   Apr  datetime
accounts
Blue Inc   90.0  50.0  95.0  101.0  04/01/2017
Alpha Co  210.0 200.0 215.0  250.0  04/01/2017
Jones LLC  200.0 150.0 140.0  180.0  04/01/2017
Yellow Inc   90.0   NaN   NaN   NaN  04/01/2017
Orange Inc  210.0   NaN   NaN  250.0  04/01/2017
Red Inc    200.0 150.0 140.0   NaN  04/01/2017
```

- Create a new dataframe with only the columns "accounts", "Jan", "Feb".

```
>>> df = df.select(["accounts", "Jan", "Feb"])
```

```
>>> df
      accounts   Jan   Feb
```

```

0   Jones LLC   150.0  200.0
1    Blue Inc    50.0   90.0
2  Yellow Inc    NaN   90.0
3  Orange Inc    NaN  210.0
4   Alpha Co   200.0  210.0
5    Red Inc   150.0  200.0

```

- Creates a new table "sales_df1" in Vantage based on the new DataFrame. The table "sales_df1" is a NOPI table because the primary index was not specified.

```
>>> df.to_sql("sales_df1")
```

- Verify that the new table "sales_df1" exists in Vantage using the DataFrame() function which creates a DataFrame based on an existing table.

```
>>> df1 = DataFrame("sales_df1")
```

```

>>> df1
   accounts  Jan  Feb
0   Blue Inc   50  90.0
1  Orange Inc  None 210.0
2    Red Inc   150 200.0
3  Yellow Inc  None  90.0
4   Jones LLC   150 200.0
5   Alpha Co   200 210.0

```

Example 2: Create a table using the optional parameter primary_index

This example uses the to_sql() function with the optional parameter "primary_index" to create a new table "sales_df2" in Vantage with the primary index "accounts". Use the optional parameter "if_exists" to replace the existing table "sales_df2" in Vantage

- Create a teradataml DataFrame "df" from the existing table "sales".

```
df = DataFrame("sales")
```

- Create a new dataframe with only the columns "accounts", "datetime".

```
>>> df = df.select(["accounts", "datetime"])
```

- Create a new table "sales_df2" with the primary index "accounts".

```
>>> df.to_sql("sales_df2", if_exists="replace", primary_index="accounts")
```

- Verify the new table by creating a new DataFrame from the new table.

```
>>> df2 = DataFrame("sales_df2")
```

```
>>> df2
      datetime
accounts
Blue Inc    17/01/04
Orange Inc  17/01/04
Red Inc     17/01/04
Yellow Inc  17/01/04
Jones LLC   17/01/04
Alpha Co    17/01/04
```

Example 3: Use optional parameter "set_table" to create a SET table "sales_subset_set"

- Create a teradataml DataFrame "df" from the existing table "sales".

```
df = DataFrame("sales")
```

- Create a new dataframe with the columns "Apr", "datetime".

```
>>> df_for_set = df.select(['Apr', 'datetime'])
```

```
>>> df_for_set
   Apr    datetime
0  101  04/01/2017
1  250  04/01/2017
2  180  04/01/2017
3  None  04/01/2017
4  250  04/01/2017
5  None  04/01/2017
```

- Create a SET table "sales_subset_set" using the optional parameter "set_table".

```
>>> df_for_set.to_sql('sales_subset_set',
primary_index='Apr', set_table=True)
```

- Verify the new table by creating a new DataFrame from the new table.

```
>>> set_df = DataFrame('sales_subset_set')
```

```
>>> set_df
      datetime
Apr
180  17/01/04
101  17/01/04
```



```
250 17/01/04
NaN 17/01/04
```

Example 4: Create a new Primary Time Index table "sales_pti".

- Create a teradataml DataFrame "df" from the existing table "sales".

```
df = DataFrame("sales")
```

- Create a new Primary Time Index table "sales_pti".

```
>>> df.to_sql('sales_pti',
timecode_column='datetime', columns_list='accounts')
```

- Verify the new table by creating a new DataFrame from the new table.

```
>>> sales_pti= DataFrame('sales_pti')
```

```
>>> sales_pti
      TD_TIMECODE  Feb  Jan  Mar  Apr
accounts
Blue Inc      17/01/04  90.0   50   95  101
Orange Inc    17/01/04  210.0  None  None  250
Red Inc       17/01/04  200.0  150  140  None
Yellow Inc    17/01/04   90.0  None  None  None
Jones LLC     17/01/04  200.0  150  140  180
Alpha Co      17/01/04  210.0  200  215  250
```

Note:

See also [copy_to_sql\(\)](#).

Exporting DataFrame to External Entities

to_pandas() Method

Use the `to_pandas()` function to creates pandas DataFrame from a teradataml DataFrame.

Note:

Column types of the resulting Pandas DataFrame depends on `pandas.read_sql_query()`.

Optional arguments:

- `index_column` specifies column name or List of column names representing columns to be used as Pandas index.

Note:

When the optional parameter *index_column* is provided, the specified column is used as the Pandas index. Otherwise, the index (if exists) of the teradataml DataFrame or the primary index of the database table is used as the Pandas index. The default integer index is used if none of these indexes exists.

- *all_rows* specifies whether all rows from the teradataml DataFrame is retrieved while creating a pandas DataFrame. The default is False.
- *num_rows* specifies the number of rows to retrieve randomly from the teradataml DataFrame while creating pandas DataFrame. The default is 99999.

Note:

This argument is ignored if *all_rows* is set to True.

- *fastexport* specifies whether FastExport protocol should be used while converting a teradataml DataFrame to a Pandas DataFrame.

Possible values for this argument:

- The default value is False. Which means, by default, FastExport protocol is not used while converting teradataml DataFrame to a Pandas DataFrame.
- If the argument is set to True, FastExport protocol is used internally for data transfer.
- If the argument is set to None, the approach is decided based on the number of rows requested by the user for extraction:
 - If requested number of rows is greater than or equal to 100,000, then FastExport is used.
 - If requested number of rows is less than 100,000, regular mode is used for data extraction.

Note:

- Teradata recommends using FastExport when number of rows in teradataml DataFrame is at least 100,000.

To extract lesser number of rows, ignore this option and go with regular approach. Because FastExport opens multiple data transfer connections to the database, which is time consuming.

- FastExport does not support all data types in Vantage.
For example, tables with BLOB and CLOB type columns cannot be extracted.
- FastExport cannot be used to extract data from a volatile or temporary table.
- For best efficiency, do not use `DataFrame.groupby()` and `DataFrame.sort()` with FastExport.

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about FastExport protocol through teradatasql driver.

- *catch_errors_warnings* specifies whether to catch errors and warnings, if any, raised by FastExport protocol while converting a teradataml DataFrame to a Pandas DataFrame.

Note:

This argument is ignored if *fastexport* argument is set to FALSE.

When FastExport is used and this argument is set to True, *to_pandas()* returns a tuple containing:

- Pandas DataFrame
- Errors, if any, in a list thrown by FastExport
- Warnings, if any, in a list thrown by FastExport

When FastExport is used and this argument is set to False (the default value), *to_pandas()* prints the FastExport errors and warnings, if any, to the standard output.

- *kwargs* specifies keyword arguments.

Arguments *coerce_float*, *parse_dates*, and *open_sessions* can be passed as keyword arguments.

- *coerce_float* specifies a boolean to attempt to convert non-string, non-numeric objects to floating point.
- *parse_dates* specifies columns to parse as dates.
- *open_sessions* specifies the number of Teradata data transfer sessions to be opened for fastexport. This argument is only applicable in fastexport mode.

Note:

If *open_sessions* argument is not set, by default, the number of data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about number of data transfer session opened during fastexport.

See https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html for more information about the *coerce_float* and *parse_dates* arguments.

Example Setup

Assume a teradataml DataFrame "df" is created from a Vantage table "sales", using command:

```
df = DataFrame("sales")
```

Example 1: Create a pandas DataFrame without specifying index

```
>>> pandas_df = df.to_pandas()
```

```
>>> pandas_df
      Feb   Jan   Mar   Apr   datetime
accounts
Alpha Co   210   200   215   250  2017-04-01
Blue Inc    90    50    95   101  2017-04-01
Yellow Inc  90  None  None  None  2017-04-01
Jones LLC  200   150   140   180  2017-04-01
Red Inc    200   150   140  None  2017-04-01
Orange Inc 210  None  None   250  2017-04-01
```

Example 2: Create a pandas DataFrame using index_column to set the index to "Feb"

```
>>> pandas_df = df.to_pandas(index_column = 'Feb')
```

```
>>> pandas_df
      accounts   Jan   Mar   Apr   datetime
Feb
210   Alpha Co   200   215   250  2017-04-01
90    Blue Inc    50    95   101  2017-04-01
90   Yellow Inc  None  None  None  2017-04-01
200   Jones LLC  150   140   180  2017-04-01
200    Red Inc   150   140  None  2017-04-01
210  Orange Inc  None  None   250  2017-04-01
```

Example 3: Create a pandas DataFrame using a list of column names for a multicolumn index

```
>>> pandas_df = df.to_pandas(index_column = ['accounts', 'Feb'])
```

```
>>> pandas_df
      Jan   Mar   Apr   datetime
accounts Feb
Yellow Inc 90  None  None  None  2017-04-01
Alpha Co   210  200   215   250  2017-04-01
Jones LLC  200   150   140   180  2017-04-01
Orange Inc 210  None  None   250  2017-04-01
Blue Inc   90    50    95   101  2017-04-01
Red Inc    200   150   140  None  2017-04-01
```

Example 4: Create a pandas DataFrame using num_rows to limit the number of rows to 3

```
>>> pandas_df = df.to_pandas(index_column = 'Feb', num_rows = 3)
```

```
>>> pandas_df
      accounts      Jan      Mar      Apr      datetime
Feb
90.0  Yellow Inc      NaN      NaN      NaN  2017-01-04
90.0   Blue Inc    50.0    95.0   101.0  2017-01-04
200.0   Red Inc   150.0   140.0      NaN  2017-01-04
```

Example 5: Create a pandas DataFrame using all_rows

```
>>> pandas_df = df.to_pandas(all_rows = True)
```

```
>>> pandas_df
      Feb      Jan      Mar      Apr      datetime
accounts
Red Inc    200.0   150.0   140.0      NaN  2017-01-04
Orange Inc  210.0      NaN      NaN   250.0  2017-01-04
Blue Inc    90.0    50.0    95.0   101.0  2017-01-04
Alpha Co   210.0   200.0   215.0   250.0  2017-01-04
Yellow Inc  90.0      NaN      NaN      NaN  2017-01-04
Jones LLC   200.0   150.0   140.0   180.0  2017-01-04
```

Example 6: Convert teradataml DataFrame to pandas DataFrame using fastexport, printing errors, if any, on screen

This example prints errors and warnings, if any, on to the screen, as *catch_errors_warnings* argument is not set.

```
>>> pandas_df = df.to_pandas(fastexport = True)
Errors: []
Warnings: []
```

```
>>> pandas_df
      Feb      Jan      Mar      Apr      datetime
accounts
Red Inc    200.0   150.0   140.0      NaN  2017-01-04
Orange Inc  210.0      NaN      NaN   250.0  2017-01-04
Blue Inc    90.0    50.0    95.0   101.0  2017-01-04
Yellow Inc  90.0      NaN      NaN      NaN  2017-01-04
Alpha Co   210.0   200.0   215.0   250.0  2017-01-04
Jones LLC   200.0   150.0   140.0   180.0  2017-01-04
```

Example 7: Convert teradataml DataFrame to pandas DataFrame using fastexport, catching errors, if any

This example catches errors and warnings, if any, raised by fastexport, and returns a tuple.

```
>>> pandas_df, err, warn = df.to_pandas(fastexport = True,
catch_errors_warnings = True)
```

```
# Print pandas df.
```

```
>>> pandas_df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|-------|-------|-------|------------|
| accounts | | | | | |
| Red Inc | 200.0 | 150.0 | 140.0 | NaN | 2017-01-04 |
| Orange Inc | 210.0 | NaN | NaN | 250.0 | 2017-01-04 |
| Blue Inc | 90.0 | 50.0 | 95.0 | 101.0 | 2017-01-04 |
| Yellow Inc | 90.0 | NaN | NaN | NaN | 2017-01-04 |
| Alpha Co | 210.0 | 200.0 | 215.0 | 250.0 | 2017-01-04 |
| Jones LLC | 200.0 | 150.0 | 140.0 | 180.0 | 2017-01-04 |

```
# Print errors list.
```

```
>>> err
[]
```

```
# Print warnings list.
```

```
>>> warn
[]
```

Example 8: Convert teradataml DataFrame to pandas DataFrame using fastexport by opening specific number of sessions

```
>>> pandas_df = df.to_pandas(fastexport = True, open_sessions = 2)
Errors: []
Warnings: []
```

```
# Print pandas df.
```

```
>>> pandas_df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|-------|-------|-------|------------|
| accounts | | | | | |
| Red Inc | 200.0 | 150.0 | 140.0 | NaN | 2017-01-04 |
| Orange Inc | 210.0 | NaN | NaN | 250.0 | 2017-01-04 |
| Blue Inc | 90.0 | 50.0 | 95.0 | 101.0 | 2017-01-04 |
| Yellow Inc | 90.0 | NaN | NaN | NaN | 2017-01-04 |
| Alpha Co | 210.0 | 200.0 | 215.0 | 250.0 | 2017-01-04 |
| Jones LLC | 200.0 | 150.0 | 140.0 | 180.0 | 2017-01-04 |

to_csv() Method

Use the `to_csv()` function to export data from a teradataml DataFrame to CSV file.

Required argument:

- `csv_file`: Specifies the name of CSV file to export the data into.

Optional arguments:

- `num_rows`: Specifies the number of rows to retrieve from DataFrame.

The default is 99999.

Note:

This argument is ignored if `all_rows` is set to True.

- `all_rows`: Specifies whether all rows from the teradataml DataFrame to be exported to the CSV file or not.

The default is False.

- `fastexport`: Specifies whether FastExport protocol should be used while exporting data.

Possible values for this argument:

- The default value is False. Which means, by default, FastExport protocol is not used.
 - If the argument is set to True, data is exported using FastExport protocol.
 - If the argument is set to None, the approach is decided based on the number of rows to be exported:
 - If requested number of rows is greater than or equal to 100,000, then FastExport is used.
 - If requested number of rows is less than 100,000, regular mode is used.
-

Note:

- Teradata recommends using FastExport when number of rows in teradataml DataFrame is at least 100,000.

To extract lesser number of rows, ignore this option and go with regular approach. Because FastExport opens multiple data transfer connections to the database, which is time consuming.

- FastExport does not support all data types in Vantage.

For example, tables with BLOB and CLOB type columns cannot be extracted.

- FastExport cannot be used to extract data from a volatile or temporary table.
 - For best efficiency, do not use `DataFrame.groupby()` and `DataFrame.sort()` with FastExport.
-

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about FastExport protocol through teradatasql driver.

- *sep*: Specifies a single character string used to separate fields in a CSV file, with default value ' , '.
- *quotechar*: Specifies a single character string used to quote fields in a CSV file, with default value ' " ' (double quote).

Note:

- *sep* and *quotechar* cannot be line feed ('\n') or carriage return ('\r').
 - *sep* and *quotechar* should not be the same.
 - Length of *sep* and *quotechar* should be 1.
-

- *catch_errors_warnings*: Specifies whether to catch errors and warnings, if any, raised by FastExport protocol while exporting data.
-

Note:

This argument is ignored if *fastexport* argument is set to FALSE.

When FastExport is used and this argument is set to True, *to_csv()* returns a tuple containing:

- Errors, if any, in a list thrown by FastExport
- Warnings, if any, in a list thrown by FastExport
- *kwargs*: Specifies keyword arguments.

Argument *open_sessions* can be passed as keyword arguments.

open_sessions specifies the number of data transfer sessions to be opened for fastexport. This argument is only applicable in fastexport mode.

Note:

If *open_sessions* argument is not set, by default, the number of data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about number of data transfer session opened during fastexport.

Example Setup

```
# Create a teradataml DataFrame.
>>> load_example_data("dataframe", "admissions_train")
>>> df = DataFrame("admissions_train")
```


Example 1: Export data from teradataml DataFrame into CSV with only required argument

```
>>> df.to_csv("export_to_csv_1.csv")
Data is successfully exported into export_to_csv_1.csv
```

Example 2: Export all rows from teradataml DataFrame into CSV using FastExport protocol

```
>>> df.to_csv("export_to_csv_2.csv", all_rows=True, fastexport=True)
Data is successfully exported into export_to_csv_2.csv
```

Example 3: Export 20 rows from teradataml DataFrame into CSV

```
>>> df.to_csv("export_to_csv_3.csv", num_rows=20)
Data is successfully exported into export_to_csv_3.csv
```

Example 4: Export data from teradataml DataFrame into CSV with additional settings

This example exports data from teradataml DataFrame into CSV using FastExport protocol by opening one Teradata data transfer session and saves errors and warnings thrown by fastexport.

```
>>> err, warn = df.to_csv("export_to_csv_4.csv", fastexport=True,
catch_errors_warnings=True, open_sessions=1 )
Data is successfully exported into export_to_csv_4.csv
```

```
>>>err
[]
```

```
>>>warn
[]
```

Example 5: Export data from teradataml DataFrame into CSV with specified separator and quote character

This example exports data from teradataml DataFrame into CSV with '|' as field separator and single quote(') as field quote character.

```
>>>df.to_csv("export_to_csv_5.csv", sep="|", quotechar="'" )
```

teradataml DataFrame Column

Access teradataml DataFrame Column

In order to use teradataml DataFrame Column, also known as ColumnExpression, in various ways in filter, assign or join, you must access the Column.

There are two ways to access column:

- Access column as DataFrame attribute:

```
<dataframe_object>.column_name
```

- Access column like dictionary:

```
<dataframe_object>["column_name"]
```

Note:

If column name contains whitespace or special character, Teradata recommends accessing ColumnExpression like dictionary.

Example Prerequisites

```
>>> from teradataml.dataframe.sql_functions import case
>>> load_example_data("GLM", ["admissions_train"])
```

```
>>> df = DataFrame("admissions_train")
>>> print(df)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 3 | no | 3.70 | Novice | Beginner | 1 |
| 1 | yes | 3.95 | Beginner | Beginner | 0 |
| 20 | yes | 3.90 | Advanced | Advanced | 1 |
| 8 | no | 3.60 | Beginner | Advanced | 1 |
| 25 | no | 3.96 | Advanced | Advanced | 1 |
| 18 | yes | 3.81 | Advanced | Advanced | 1 |
| 24 | no | 1.87 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |

Example: Access ColumnExpression as attribute and use the same as predicate for filter

```
>>> gpa = df.gpa

>>> good_df = df[case([(gpa > 3.0, 'good'),
                       (gpa > 2.0, 'average')],
                      else_='bad') == 'good']

>>> print(good_df)
   masters  gpa  stats programming  admitted
id
13      no  4.00  Advanced      Novice         1
11      no  3.13  Advanced    Advanced         1
9       no  3.82  Advanced    Advanced         1
26     yes  3.57  Advanced    Advanced         1
3       no  3.70   Novice    Beginner         1
1      yes  3.95  Beginner    Beginner         0
20     yes  3.90  Advanced    Advanced         1
18     yes  3.81  Advanced    Advanced         1
5       no  3.44   Novice    Novice           0
32     yes  3.46  Advanced    Beginner         0
```

```
>>> print(good_df.shape)
(35, 6)
```

Example: Access ColumnExpression like dictionary and use the same to create a new DataFrame

This example accesses ColumnExpression like dictionary and uses the same to create a new DataFrame with an additional 'rating' column using assign operation, with the same case construct used in the previous example.

```
>>> gpa = df['gpa']

>>> whens_df = df.assign(rating = case([(gpa > 3.0, 'good'),
                                         (gpa > 2.0, 'average')],
                                       else_='bad'))

>>> print(whens_df)
   masters  gpa  stats programming  admitted  rating
id
5       no  3.44   Novice    Novice         0    good
3       no  3.70   Novice    Beginner        1    good
```

| | | | | | | |
|----|-----|------|----------|----------|---|---------|
| 1 | yes | 3.95 | Beginner | Beginner | 0 | good |
| 20 | yes | 3.90 | Advanced | Advanced | 1 | good |
| 8 | no | 3.60 | Beginner | Advanced | 1 | good |
| 25 | no | 3.96 | Advanced | Advanced | 1 | good |
| 18 | yes | 3.81 | Advanced | Advanced | 1 | good |
| 24 | no | 1.87 | Advanced | Novice | 1 | bad |
| 26 | yes | 3.57 | Advanced | Advanced | 1 | good |
| 38 | yes | 2.65 | Advanced | Beginner | 1 | average |

```
>>> print(whens_df.shape)
(40, 7)
```

teradataml DataFrame Column Manipulation

isin()

Use the `isin()` function to check for the presence of values in a column.

Example Prerequisites

```
>>> load_example_data("dataframe", "admissions_train")
```

```
>>> df = DataFrame('admissions_train')
>>> df
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 7 | yes | 2.33 | Novice | Novice | 1 |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 40 | yes | 3.95 | Novice | Beginner | 0 |

Example: Filter results where gpa values are in any of the specified values

```
>>> df[df.gpa.isin([4.0, 3.0, 2.0, 1.0, 3.5, 2.5, 1.5])]
   masters  gpa  stats programming  admitted
id
31    yes  3.5  Advanced    Beginner        1
```

| | | | | | |
|----|-----|-----|----------|----------|---|
| 6 | yes | 3.5 | Beginner | Advanced | 1 |
| 13 | no | 4.0 | Advanced | Novice | 1 |
| 4 | yes | 3.5 | Beginner | Novice | 1 |
| 29 | yes | 4.0 | Novice | Beginner | 0 |
| 15 | yes | 4.0 | Advanced | Advanced | 1 |
| 36 | no | 3.0 | Advanced | Novice | 0 |

Example: Filter results where the 'stats' values are neither 'Novice' nor 'Advanced'

```
>>> df[~df.stats.isin(['Novice', 'Advanced'])]
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 1 | yes | 3.95 | Beginner | Beginner | 0 |
| 2 | yes | 3.76 | Beginner | Beginner | 0 |
| 8 | no | 3.60 | Beginner | Advanced | 1 |
| 4 | yes | 3.50 | Beginner | Novice | 1 |
| 6 | yes | 3.50 | Beginner | Advanced | 1 |

NA Checking: isna()/notna() and isnull()/notnull() Methods

Missing values in data can be handled by using the `isna()` or `notna()` methods. Currently, the only NA value supported is `None`. `'isnull'` and `'notnull'` are aliases of `'isna'` and `'notna'` respectively. Other possible NA values, `+Inf`, `-Inf`, and `NaN` (typically seen in floating point calculations) are not supported. See the [Teradata Package for Python Limitations and Considerations](#) section for more information.

Example

```
>>> df = DataFrame('iris')
>>> df
```

| | PetalWidth | SepalLength | SepalWidth | PetalLength | Name |
|---|------------|-------------|------------|-------------|---------------------|
| 0 | | 5.800 | 2.700 | 5.100 | 1.90 Iris-virginica |
| 1 | | 6.500 | 3.000 | 5.800 | 2.20 Iris-virginica |
| 2 | | 1.012 | 1.202 | 3.232 | 4.23 None |
| 3 | | 5.400 | 3.700 | 1.500 | 0.20 Iris-setosa |
| 4 | | 6.700 | 2.500 | 5.800 | 1.80 Iris-virginica |
| 5 | | 7.200 | 3.600 | 6.100 | 2.50 Iris-virginica |

```
>>> df.assign(drop_columns = True, Name = df.Name, NullName = df.Name.isna())
```

| | Name | NullName |
|---|----------------|----------|
| 0 | None | 1 |
| 1 | Iris-virginica | 0 |
| 2 | Iris-virginica | 0 |
| 3 | Iris-virginica | 0 |

```
4 Iris-virginica      0
5 Iris-virginica      0
```

Type Casting: cast()

A DataFrame column (Column Expression) can be cast to a specified terdatasqlalchemy type using the cast method, and can be used with the DataFrame methods assign and filter.

Example Prerequisites

```
>>> load_example_data("dataframe","admissions_train")
```

```
>>> df = DataFrame('admissions_train')
>>> df
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 40 | yes | 3.95 | Novice | Beginner | 0 |
| 7 | yes | 2.33 | Novice | Novice | 1 |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 36 | no | 3.00 | Advanced | Novice | 0 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |

```
>>> df.dtypes
id                int
masters           str
gpa              float
stats            str
programming      str
admitted         int
```

Example: Use assign to create a new DataFrame with a new column, casted to VARCHAR(5) type

Use assign() to create a new DataFrame with a new column 'char_id', which is the 'id' column (of type INTEGER) cast to VARCHAR(5) (object of terdatasqlalchemy type corresponding to SQL Type VARCHAR(5)).

```
>>> from terdatasqlalchemy import VARCHAR
```

```
>>> new_df = df.assign(char_id = df.id.cast(type_=VARCHAR(5)))
>>> new_df
```

| | masters | gpa | stats | programming | admitted | char_id |
|----|---------|------|----------|-------------|----------|---------|
| id | | | | | | |
| 5 | no | 3.44 | Novice | Novice | 0 | 5 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 | 34 |
| 13 | no | 4.00 | Advanced | Novice | 1 | 13 |
| 40 | yes | 3.95 | Novice | Beginner | 0 | 40 |
| 22 | yes | 3.46 | Novice | Beginner | 0 | 22 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 19 |
| 36 | no | 3.00 | Advanced | Novice | 0 | 36 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 | 15 |
| 7 | yes | 2.33 | Novice | Novice | 1 | 7 |
| 17 | no | 3.83 | Advanced | Advanced | 1 | 17 |

```
>>> new_df.dtypes
id                int
masters           str
gpa              float
stats            str
programming       str
admitted         int
char_id          str
```

Example: Use assign to create a new DataFrame with a new column, casted to VARCHAR type

Use assign() to create a new DataFrame with a new column 'char_id', which is the 'id' column (of type INTEGER) cast to VARCHAR (teradatasqlalchemy type corresponding to SQL Type VARCHAR).

```
>>> new_df_2 = df.assign(char_id = df.id.cast(type_=VARCHAR))
>>> new_df_2
```

| | masters | gpa | stats | programming | admitted | char_id |
|----|---------|------|----------|-------------|----------|---------|
| id | | | | | | |
| 5 | no | 3.44 | Novice | Novice | 0 | 5 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 | 34 |
| 13 | no | 4.00 | Advanced | Novice | 1 | 13 |
| 40 | yes | 3.95 | Novice | Beginner | 0 | 40 |
| 22 | yes | 3.46 | Novice | Beginner | 0 | 22 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 | 19 |
| 36 | no | 3.00 | Advanced | Novice | 0 | 36 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 | 15 |
| 7 | yes | 2.33 | Novice | Novice | 1 | 7 |

| | | | | | | |
|----|----|------|----------|----------|---|----|
| 17 | no | 3.83 | Advanced | Advanced | 1 | 17 |
|----|----|------|----------|----------|---|----|

```
>>> new_df_2.dtypes
id                int
masters           str
gpa              float
stats            str
programming       str
admitted         int
char_id          str
```

Example: Filter data with match on a column cast to another type

```
>>> df[df.id.cast(VARCHAR(5)) == '1']
  masters  gpa  stats programming  admitted
id
1    yes  3.95  Beginner    Beginner         0
```

```
>>> df[df.id.cast(VARCHAR) == '1']
  masters  gpa  stats programming  admitted
id
1    yes  3.95  Beginner    Beginner         0
```

SQL Functions

to_numeric

The `to_numeric` function converts a string-like representation of a number to a numeric type. It can be used with the string columns of the DataFrame in the `DataFrame.assign` method.

Example Prerequisite

A DataFrame with all string type columns.

```
>>> df = DataFrame('numeric_strings')

   hex decimal commas numbers
0   19FF   00.77   08,8       1
1   abcd   0.77   0,88       1
```


| | | | | |
|---|------------|-------|------|-----|
| 2 | ABCDEFABCD | 0.7.7 | ,088 | 999 |
| 3 | 2018 | .077 | 088, | 0 |

```
>>> df.dtypes
```

```
hex      str
decimal  str
commas   str
numbers  str
```

Example: Convert to Numeric Type

To use the DataFrame in a numerical calculation, they first need to be converted to numeric type.

```
>>> from teradataml import to_numeric
```

```
>>> df = df.assign(drop_columns = True,
                  numbers = df.numbers,
                  numeric = to_numeric(df.numbers))
```

| | numbers | numeric |
|---|---------|---------|
| 0 | 1 | 1 |
| 1 | 1 | 1 |
| 2 | 999 | 999 |
| 3 | 0 | 0 |

```
>>> df.dtypes
```

```
numbers      str
numeric    decimal.Decimal
```

Example: Use Optional *format_* Keyword when Converting

The `to_numeric` function may not be able to parse the string into a numeric value if the string has an unrecognizable format. It returns `None` in this case.

```
# converting decimal-like strings to numeric
>>> df.assign(drop_columns = True,
              decimal = df.decimal,
              numeric_dec = to_numeric(df.decimal))
```

| | decimal | numeric_dec |
|---|---------|-------------|
| 0 | 00.77 | .77 |
| 1 | 0.77 | .77 |

```
2  0.7.7      None
3   .077      .077
```

You can control which strings are recognizable by passing a format string into the optional `format_` keyword.

```
# converting comma (group separated) strings to numeric
>>> df.assign(drop_columns = True,
              commas = df.commas,
              numeric_commas = to_numeric(df.commas, format_ = '9G99'))
```

| | commas | numeric_commas |
|---|--------|----------------|
| 0 | 08,8 | None |
| 1 | 0,88 | 88 |
| 2 | ,088 | None |
| 3 | 088, | None |

```
# converting hex strings to numeric
>>> df.assign(drop_columns = True,
              hex = df.hex,
              numeric_hex = to_numeric(df.hex, format_ = 'XXXXXXXXXX'))
```

| | hex | numeric_hex |
|---|------------|--------------|
| 0 | 19FF | 6655 |
| 1 | abcd | 43981 |
| 2 | ABCDEFABCD | 737894443981 |
| 3 | 2018 | 8216 |

The format string follows the syntax of the `to_number` function in the Analytics Database.

Note:

For more information, see the Data Type Conversion Functions section in the *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145, Release 16.20.

Example: Use String Literals as Arguments

The `to_numeric` function can take DataFrame columns or string literals as arguments.

```
# converting literals to numeric
>>> df.assign(drop_columns = True,
              a = to_numeric('123,456', format_ = '999,999'),
              b = to_numeric('1,333.555', format_ = '9,999D999'),
              c = to_numeric('2,333,2', format_ = '9G999G9'),
              d = to_numeric('3E20'),
```

```
e = to_numeric('$41.99', format_ = 'L99.99'),
f = to_numeric('$.12', format_ = 'L.99'),
g = to_numeric('dollar123,456.00',
               format_ = 'L999G999D99',
               nls = {'param': 'currency',
                     'value': 'dollar'})).head(1)
```

| | a | b | c | d | e | f | g |
|---|--------|----------|-------|--------------------------|-------|-----|--------|
| 0 | 123456 | 1333.555 | 23332 | 300000000000000000000000 | 41.99 | .12 | 123456 |

case

The `case()` function can be used to add SQL CASE logic based on DataFrame column expressions. It can be used with DataFrame assign method and filter method.

Example Prerequisites

```
>>> from teradataml.dataframe.sql_functions import case
```

```
>>> load_example_data("GLM", ["admissions_train"])
```

```
>>> df = DataFrame("admissions_train")
```

```
>>> print(df)
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 3 | no | 3.70 | Novice | Beginner | 1 |
| 1 | yes | 3.95 | Beginner | Beginner | 0 |
| 20 | yes | 3.90 | Advanced | Advanced | 1 |
| 8 | no | 3.60 | Beginner | Advanced | 1 |
| 25 | no | 3.96 | Advanced | Advanced | 1 |
| 18 | yes | 3.81 | Advanced | Advanced | 1 |
| 24 | no | 1.87 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |

```
>>> print(df.shape)
(40, 6)
```

Example: Run case() with 'whens' passing a 2-tuple

This example shows the 'whens' argument passes a 2-tuple and filters only the 'good' records based on the rating:

- - gpa > 3.0 : 'good'
- - 2.0 < gpa <=3.0 : 'average'
- - gpa <= 2.0 : 'bad'

```
>>> good_df = df[case([(df.gpa > 3.0, 'good'),
                      (df.gpa > 2.0, 'average')],
                      else_='bad') == 'good']
```

```
>>> print(good_df)
   masters  gpa  stats programming  admitted
id
13      no  4.00  Advanced      Novice         1
11      no  3.13  Advanced    Advanced         1
9       no  3.82  Advanced    Advanced         1
26     yes  3.57  Advanced    Advanced         1
3       no  3.70   Novice    Beginner         1
1      yes  3.95  Beginner    Beginner         0
20     yes  3.90  Advanced    Advanced         1
18     yes  3.81  Advanced    Advanced         1
5       no  3.44   Novice    Novice           0
32     yes  3.46  Advanced    Beginner         0

>>> print(good_df.shape)
(35, 6)
```

Example: With same case construct, create a new DataFrame with an additional 'rating' column using assign operation

```
>>> whens_df = df.assign(rating = case([(df.gpa > 3.0, 'good'),
                                       (df.gpa > 2.0, 'average')],
                                       else_='bad'))
```

```
>>> print(whens_df)
   masters  gpa  stats programming  admitted  rating
id
5       no  3.44   Novice    Novice         0    good
3       no  3.70   Novice    Beginner        1    good
1      yes  3.95  Beginner    Beginner        0    good
20     yes  3.90  Advanced    Advanced        1    good
```

| | | | | | | |
|----|-----|------|----------|----------|---|---------|
| 8 | no | 3.60 | Beginner | Advanced | 1 | good |
| 25 | no | 3.96 | Advanced | Advanced | 1 | good |
| 18 | yes | 3.81 | Advanced | Advanced | 1 | good |
| 24 | no | 1.87 | Advanced | Novice | 1 | bad |
| 26 | yes | 3.57 | Advanced | Advanced | 1 | good |
| 38 | yes | 2.65 | Advanced | Beginner | 1 | average |

```
>>> print(whens_df.shape)
(40, 7)
```

Example: Run case() without 'else_' argument

This example shows running the case() function without specifying the 'else_' argument, this results NULLs when no condition in the 'whens' argument is met.

```
>>> no_else = df.assign(rating = case([(df.gpa > 3.0, 'good')]))
```

```
>>> print(no_else)
  masters  gpa  stats programming  admitted  rating
id
5      no  3.44   Novice      Novice         0    good
3      no  3.70   Novice   Beginner         1    good
1     yes  3.95  Beginner   Beginner         0    good
20    yes  3.90  Advanced  Advanced         1    good
8      no  3.60  Beginner  Advanced         1    good
25    no  3.96  Advanced  Advanced         1    good
18    yes  3.81  Advanced  Advanced         1    good
24    no  1.87  Advanced   Novice         1   None
26    yes  3.57  Advanced  Advanced         1    good
38    yes  2.65  Advanced  Beginner         1   None
```

```
>>> print(no_else.shape)
(40, 7)
```

Example: Run case() with 'whens' passing dictionary along with argument 'value'

This example shows the 'whens' argument passing dictionary along with the argument 'value'. The result is 'admitted' when the argument df.admitted == 1, 'not admitted' when df.admitted == 0, and 'don't know' otherwise.

```
>>> whens_value_df = df.assign(admitted_text = case({ 1 : "admitted", 0 :
"not admitted"}),
```

```
value=df.admitted,
else_="don't know"))
```

```
>>> print(whens_value_df)
   masters  gpa  stats programming  admitted  admitted_text
id
13      no  4.00  Advanced      Novice         1      admitted
11      no  3.13  Advanced    Advanced         1      admitted
9       no  3.82  Advanced    Advanced         1      admitted
28      no  3.93  Advanced    Advanced         1      admitted
33      no  3.55   Novice     Novice          1      admitted
10      no  3.71  Advanced    Advanced         1      admitted
16      no  3.70  Advanced    Advanced         1      admitted
32     yes  3.46  Advanced  Beginner          0  not admitted
34     yes  3.85  Advanced  Beginner          0  not admitted
17      no  3.83  Advanced    Advanced         1      admitted

>>> print(whens_value_df.shape)
(40, 7)
```

Example: Run case() using literal column name as SQL expressions

This example shows using literal column name as SQL expressions and choosing to emit values from two different columns as a result of the same case expression.

In the example, you project values from column 'average_rating' if $2.0 < \text{gpa} \leq 3.0$, and values from column 'good_rating' when $\text{gpa} > 3.0$, naming the column 'ga_rating'.

```
>>> from sqlalchemy.sql import literal_column
```

Create a new DataFrame with a new column 'good_rating' which is set to 'good' only when $\text{gpa} > 3.0$, else NULL.

```
>>> whens_new_df = df.assign(good_rating = case([(df.gpa > 3.0, 'good')]))
```

Add another column named 'avg_rating' to the DataFrame which is set to 'average' only when $2 < \text{gpa} \leq 3.0$, else NULL.

```
>>> whens_new_df = whens_new_df.assign(avg_rating = case([(whens_new_df.gpa >
2.0) & (whens_new_df.gpa <= 3.0), 'average'])))
```

Create a DataFrame with another column named 'ga_rating', which is set to the value from column 'good_rating' when $\text{gpa} > 3$, and value from column 'avg_rating' when $\text{gpa} > 2$, else NULL.

```
>>> literal_df = whens_new_df.assign(ga_rating = case([(whens_new_df.gpa
> 3.0, literal_column('good_rating')), (whens_new_df.gpa >
2.0, literal_column('avg_rating'))]))
```

```
>>> print(literal_df)
id      masters  gpa      stats programming  admitted good_rating  avg_rating ga_rating
5         no    3.44    Novice      Novice         0         good        None      good
3         no    3.70    Novice      Beginner        1         good        None      good
1         yes   3.95    Beginner    Beginner        0         good        None      good
20        yes   3.90    Advanced    Advanced        1         good        None      good
8         no    3.60    Beginner    Advanced        1         good        None      good
25        no    3.96    Advanced    Advanced        1         good        None      good
18        yes   3.81    Advanced    Advanced        1         good        None      good
24        no    1.87    Advanced    Novice         1         None        None      None
26        yes   3.57    Advanced    Advanced        1         good        None      good
38        yes   2.65    Advanced    Beginner        1         None        average  average
```

desc

The `desc()` function generates a new `ColumnExpression` which sorts the actual expression in descending order.

Note:

- This function is supported only while sorting data.
- This function is neither supported in projection nor supported in filtering data.

Example

```
>>> from teradataml import load_example_data
```

```
>>> load_example_data("dataframe","sales")
```

```
# Create teradataml dataframe.
```

```
>>> df = DataFrame.from_table('sales')
```

```
>>> df.csum(sort_columns=[df.accounts, df.Feb.desc()], drop_columns=True)
```

| | csum_Feb | csum_Jan | csum_Mar | csum_Apr |
|---|----------|----------|----------|----------|
| 0 | 500.0 | 400 | 450 | 531 |
| 1 | 910.0 | 550 | 590 | 781 |
| 2 | 1000.0 | 550 | 590 | 781 |
| 3 | 710.0 | 400 | 450 | 781 |
| 4 | 300.0 | 250 | 310 | 351 |
| 5 | 210.0 | 200 | 215 | 250 |

asc

The `asc()` function generates a new `ColumnExpression` which sorts the actual expression in ascending order.

Note:

- This function is supported only while sorting data.
- This function is neither supported in projection nor supported in filtering data.

Example

```
>>> from teradataml import load_example_data

>>> load_example_data("dataframe","sales")

# Create teradataml dataframe.
>>> df = DataFrame.from_table('sales')

>>> df.csum(sort_columns=[df.accounts, df.Feb.asc()], drop_columns=True)
   csum_Feb  csum_Jan  csum_Mar  csum_Apr
0    380.0    200.0    235.0    281.0
1    790.0    350.0    375.0    531.0
2   1000.0    550.0    590.0    781.0
3    580.0    350.0    375.0    281.0
4    180.0     50.0     95.0    101.0
5     90.0     NaN     NaN     NaN
```

distinct

The `distinct()` function generates a new `ColumnExpression` which removes duplicate rows while processing the function.

Note:

- This function is supported only in projection.
- This function is neither supported in sorting data nor supported in filtering data.

Example

```
>>> from teradataml import *

>>> load_example_data("dataframe","sales")

>>> df = DataFrame.from_table('sales')
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|-------|-------|-------|------------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50.0 | 95.0 | 101.0 | 04/01/2017 |
| Alpha Co | 210.0 | 200.0 | 215.0 | 250.0 | 04/01/2017 |
| Jones LLC | 200.0 | 150.0 | 140.0 | 180.0 | 04/01/2017 |
| Yellow Inc | 90.0 | NaN | NaN | NaN | 04/01/2017 |
| Orange Inc | 210.0 | NaN | NaN | 250.0 | 04/01/2017 |
| Red Inc | 200.0 | 150.0 | 140.0 | NaN | 04/01/2017 |

```
>>> df.assign(drop_columns=True, distinct_feb=df.Feb.distinct())
      distinct_feb
0             210.0
1              90.0
2             200.0
```

String Manipulation

DataFrame columns have methods that can be used to manipulate string-like data. These methods are available by invoking the 'str' property. The 'str' property is only valid for string-like columns.

contains() Method

The contains() method tests if the given regular expression pattern matches string values in the column.

Example

```
>>> df = DataFrame('sales')

>>> accounts = df['accounts']

>>> df.assign(drop_columns = True, Accounts = accounts, has_llc
= accounts.str.contains('LLC'))
```

| | Accounts | has_llc |
|---|----------|---------|
| 0 | Alpha Co | 0 |
| 1 | Blue Inc | 0 |

```

2 Yellow Inc      0
3   Jones LLC      1
4     Red Inc      0
5 Orange Inc      0

```

Examples using the case parameter

The `contains()` method has a `case` parameter to toggle case-sensitive matching on or off. The default value is on.

```
>>> df = DataFrame('sales')
```

```
>>> accounts = df['accounts']
```

```

>>> df.assign(drop_columns = True, Accounts = accounts, has_llc =
accounts.str.contains('llc', case=True))
      Accounts has_llc
0   Blue Inc         0
1   Alpha Co         0
2   Jones LLC         0
3 Yellow Inc         0
4 Orange Inc         0
5   Red Inc         0

```

```

>>> df.assign(drop_columns = True, Accounts = accounts, has_llc =
accounts.str.contains('llc', case=False))
      Accounts has_llc
0   Blue Inc         0
1   Alpha Co         0
2   Jones LLC         1
3 Yellow Inc         0
4 Orange Inc         0
5   Red Inc         0

```

Example using the na parameter

Use the `na` parameter to specify an optional fill value for columns that have a NULL value. You can pass numeric, string, or bool literals.

```
>>> df = DataFrame('employee_info')
```

```

>>> df
      first_name marks    dob joined_date
employee_no

```

| | | | | |
|-----|-------|------|------|----------|
| 112 | None | None | None | 18/12/05 |
| 101 | abcde | None | None | 02/12/05 |
| 100 | abcd | None | None | None |

```
>>> df.assign(has_name = df.first_name.str.contains('abcd', na = 'FNU'))
      first_name marks  dob joined_date has_name
employee_no
112          None  None  None    18/12/05      FNU
101        abcde  None  None    02/12/05        1
100         abcd  None  None         None        1
```

lower() Method

The lower() method maps character values to lowercase.

Example Prerequisite

```
>>> df = DataFrame('sales')
```

```
>>> df
      Feb  Jan  Mar  Apr  datetime
accounts
Alpha Co  210.0  200  215  250  04/01/2017
Red Inc   200.0  150  140  None  04/01/2017
Orange Inc 210.0  None  None  250  04/01/2017
Jones LLC  200.0  150  140  180  04/01/2017
Yellow Inc  90.0  None  None  None  04/01/2017
Blue Inc   90.0   50   95  101  04/01/2017
```

Example

```
>>> df.assign(drop_columns = True, lower = df.accounts.str.lower())
      lower
0  alpha co
1  red inc
2  orange inc
3  jones llc
4  yellow inc
5  blue inc
```

Example

```
>>> df[df.accounts.str.lower() == 'orange inc']
           Feb  Jan  Mar  Apr  datetime
accounts
Orange Inc  210.0  None  None  250  04/01/2017
```

strip() Method

The strip() method in teradataml package is similar to the strip method of strings in Python. It removes leading and trailing whitespace from string values in a column.

Example Prerequisite

```
>>> df = DataFrame('employee_info')
```

```
>>> df
           first_name marks  dob joined_date
employee_no
101             abcde  None  None    02/12/05
112              None  None  None    18/12/05
100             abcd  None  None         None
```

```
>>> fn = df.first_name
```

```
# create a column with some whitespace
```

```
>>> wdf = df.assign(drop_columns = True, fn = fn, w_spaces = '\n ' + fn +
'\v\f \t')
```

Example: Display without the Strip Method

```
>>> >>> wdf
           fn          w_spaces
0  None          None
1  abcde  \n abcde

           \t
2  abcd  \n abcd

           \t
```

Example: Display with the Strip Method

```
>>> wdf.assign(drop_columns = True, wo_wspaces = wdf.w_spaces.str.strip())
   wo_wspaces
0         None
1        abcde
2         abcd
```

Regular Aggregate Functions Supported by DataFrame Column

teradataml DataFrameColumn supports the following set of regular aggregate functions which can be used with and without DataFrame.groupby().

Note:

- You must use DataFrame.assign() when using the aggregate functions on ColumnExpression, also known as, teradataml DataFrameColumn.
- You should always use "drop_columns=True" in DataFrame.assign() while running the aggregate operation on teradataml DataFrame.
- `drop_columns` argument in DataFrame.assign() is ignored, when aggregate function is operated on DataFrame.groupby().

See the [DataFrameColumn Aggregate Functions](https://docs.teradata.com/) section of *Teradata Package for Python Function Reference*, B700-4008) at <https://docs.teradata.com/> for detailed description and usage examples of these functions.

Regular Aggregate Functions supported by DataFrame Column

| Sr. No. | Function Name | Description |
|---------|---------------|---|
| 1 | corr() | Returns the Sample Pearson product moment correlation coefficient of its arguments for all non-null data point pairs. |
| 2 | count() | Returns column-wise count of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 3 | covar_pop() | Returns the column-wise population covariance of its arguments for all non-null data point pairs. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |
| 4 | covar_samp() | Returns the column-wise sample covariance of its arguments for all non-null data point pairs. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |

| Sr. No. | Function Name | Description |
|---------|------------------|--|
| 5 | kurtosis() | <p>Returns column-wise kurtosis value of the ColumnExpression, also known as, teradataml DataFrameColumn.</p> <p>Kurtosis is the fourth moment of the distribution of the standardized (z) values. It is a measure of the outlier (rare, extreme observation) character of the distribution as compared with the normal (or Gaussian) distribution.</p> <ul style="list-style-type: none"> • The normal distribution has a kurtosis of 0. • Positive kurtosis indicates that the distribution is more outlier-prone than the normal distribution. • Negative kurtosis indicates that the distribution is less outlier-prone than the normal distribution. |
| 6 | max() | Returns column-wise maximum value of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 7 | mean() | Returns column-wise mean value of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 8 | median() | Returns column-wise median value of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 9 | min() | Returns column-wise minimum value of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 10 | percentile() | Return the value which represents the desired percentile for the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 11 | regr_avgx() | Returns the column-wise mean of the independent variable for all non-null data pairs of the dependent and an independent variable arguments. |
| 12 | regr_avgy() | Returns the column-wise mean of the dependent variable for all non-null data pairs of the dependent and independent variable arguments. |
| 13 | regr_count() | Returns the column-wise count of all non-null data pairs of the dependent and independent variable arguments. |
| 14 | regr_intercept() | <p>Returns the column-wise intercept of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments.</p> <p>The intercept is the point at which the regression line through the non-null data pairs in the sample intersects the ordinate, or y-axis, of the graph.</p> |
| 15 | regr_r2() | Returns the column-wise coefficient of determination for all non-null data pairs of the dependent and independent variable arguments. |
| 16 | regr_slope() | Returns the column-wise coefficient slope of the univariate linear regression line through all non-null data pairs of the dependent and an independent variable arguments. |
| 17 | regr_sxx() | Returns the column-wise sum of the squares of the independent variable expression for all non-null data pairs of dependent and an independent variable arguments. |

| Sr. No. | Function Name | Description |
|---|-------------------------|--|
| 18 | <code>regr_sxy()</code> | Returns the column-wise sum of the products of the independent variable and the dependent variable for all non-null data pairs of the dependent and independent variable arguments. |
| 19 | <code>regr_syy()</code> | Returns the column-wise sum of the squares of the dependent variable expression for all non-null data pairs of dependent and an independent variable arguments. |
| 20 | <code>skew()</code> | <p>Returns column-wise skewness of the distribution of the ColumnExpression, also known as, teradataml DataFrameColumn.</p> <p>Skewness is the third moment of a distribution. It is a measure of the asymmetry of the distribution about its mean compared with the normal (or Gaussian) distribution.</p> <ul style="list-style-type: none"> • The normal distribution has a skewness of 0. • Positive skewness indicates a distribution having an asymmetric tail extending toward more positive values. • Negative skewness indicates an asymmetric tail extending toward more negative values. |
| 21 | <code>std()</code> | <p>Returns column-wise sample or population standard deviation value of the ColumnExpression, also known as, teradataml DataFrameColumn. The standard deviation is the second moment of a distribution.</p> <ul style="list-style-type: none"> • For a sample, it is a measure of dispersion from the mean of that sample. • For a population, it is a measure of dispersion from the mean of that population. <p>The computation is more conservative for the population standard deviation to minimize the effect of outliers on the computed value.</p> |
| 22 | <code>sum()</code> | Returns column-wise sum value of the ColumnExpression, also known as, teradataml DataFrameColumn. |
| 23 | <code>var()</code> | <p>Returns column-wise sample or population variance of the columns of the ColumnExpression, also known as, teradataml DataFrameColumn.</p> <ul style="list-style-type: none"> • The variance of a population is a measure of dispersion from the mean of that population. • The variance of a sample is a measure of dispersion from the mean of that sample. It is the square of the sample standard deviation. |
| teradataml special aggregate functions | | |
| 24 | <code>csum()</code> | Returns cumulative sum value for rows in the partition of the column. |
| 25 | <code>msum()</code> | Computes the moving sum for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |
| 26 | <code>mavg()</code> | Computes the moving average for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |
| 27 | <code>mdiff()</code> | Computes the moving difference for the current row and the preceding "width" rows in a partition, by sorting the rows according to "sort_columns". |

| Sr. No. | Function Name | Description |
|---------|------------------------|---|
| 28 | <code>mmlnreg()</code> | Computes the moving linear regression for the current row and the preceding "width"-1 rows in a partition, by sorting the rows according to "sort_columns". |

teradataml Window Aggregates

Window on DataFrame

Use the `window()` function to run window aggregate functions on the teradataml DataFrame.

The function allows user to specify the following window types for computations:

- Cumulative
- Group
- Moving
- Remaining

By default, window with Unbounded Preceding and Unbounded Following is considered for calculation.

Note:

If both *partition_columns* and *order_columns* are 'None' and the original DataFrame has BLOB and CLOB type of columns, then

- Window aggregate operation on CLOB and BLOB type of columns is omitted;
 - Resultant DataFrame does not contain the BLOB and CLOB type of columns from the original DataFrame.
-

Example Setup

```
>>> from teradataml import *

>>> load_example_data("dataframe", "sales")

>>> df = DataFrame.from_table('sales')
```

Example 1: Create a window on a teradataml DataFrame

```
>>> window = df.window()
```

Example 2: Create a cumulative (expanding) window

This example creates a cumulative (expanding) window with rows between unbounded preceding and 3 preceding, with *partition_columns* and *order_columns* arguments, and with default sorting.

```
>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
```

```

window_start_point=None,
window_end_point=-3)

```

Example 3: Create a moving (rolling) window

This example creates a moving (rolling) window with rows between current row and 3 following, with sorting done on 'Feb', 'datetime' columns in descending order, and with *partition_columns* argument.

```

>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        window_start_point=0,
                        window_end_point=3)

```

Example 4: Create a remaining (contracting) window

This example creates a remaining (contracting) window with rows between current row and unbounded following, with sorting done on 'Feb', 'datetime' columns in ascending order, and NULL values in 'Feb', 'datetime' columns appear at last.

```

>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        nulls_first=False,
                        window_start_point=0,
                        window_end_point=None)

```

Example 5: Create a grouping window

This example creates a grouping window, with sorting done on 'Feb', 'datetime' columns in ascending order, and NULL values in 'Feb', 'datetime' columns appear at last.

```

>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        nulls_first=False,
                        window_start_point=None,
                        window_end_point=None)

```

Example 6: Create a window, ignoring all parameters

```

>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,

```

```

nulls_first=False,
ignore_window=True)

```

Example 7: Perform sum on every valid column in DataFrame

```

>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        nulls_first=False,
                        ignore_window=True)

```

```

>>> window.sum()
      Feb      Jan      Mar      Apr      datetime      Apr_sum      Feb_sum      Jan_sum      Mar_sum
accounts
Jones LLC    200.0    150.0    140.0    180.0    04/01/2017         781      1000.0         550         590
Red Inc      200.0    150.0    140.0      NaN    04/01/2017         781      1000.0         550         590
Yellow Inc    90.0      NaN      NaN      NaN    04/01/2017         781      1000.0         550         590
Orange Inc   210.0      NaN      NaN    250.0    04/01/2017         781      1000.0         550         590
Blue Inc      90.0     50.0     95.0    101.0    04/01/2017         781      1000.0         550         590
Alpha Co     210.0    200.0    215.0    250.0    04/01/2017         781      1000.0         550         590

```

Example 8: Perform count on every valid column in DataFrame

```

>>> window = df.window()

```

```

>>> window.count()
      Feb      Jan      Mar      Apr      datetime      Apr_count      Feb_count      Jan_count      Mar_count
accounts_count      datetime_count
accounts
Jones LLC    200.0    150.0    140.0    180.0    04/01/2017         4         6         4         4
6
Red Inc      200.0    150.0    140.0      NaN    04/01/2017         4         6         4         4
6
Yellow Inc    90.0      NaN      NaN      NaN    04/01/2017         4         6         4         4
6
Orange Inc   210.0      NaN      NaN    250.0    04/01/2017         4         6         4         4
6
Blue Inc      90.0     50.0     95.0    101.0    04/01/2017         4         6         4         4
6
Alpha Co     210.0    200.0    215.0    250.0    04/01/2017         4         6         4         4
6

```

Example 9: Perform count of all valid columns in DataFrame, which is grouped by 'accounts'

```

>>> window = df.groupby("accounts").window()

```

```

>>> window.count()
      accounts      accounts_count
0    Jones LLC         6
1      Red Inc         6
2    Yellow Inc         6
3    Orange Inc         6
4      Blue Inc         6
5     Alpha Co         6

```

Window on DataFrame Column

Use the `window()` function to run window aggregate functions on the teradataml DataFrame columns.

The function allows user to specify the following window types for computations:

- Cumulative
- Group
- Moving
- Remaining

By default, window with Unbounded Preceding and Unbounded Following is considered for calculation.

Note:

If both *partition_columns* and *order_columns* are 'None', then window cannot be created on CLOB and BLOB type of columns.

Example Setup

```
>>> from teradataml import *

>>> load_example_data("dataframe", "sales")

>>> df = DataFrame.from_table('sales')
```

Example 1: Create a window on a teradataml DataFrame Column

```
>>> window = df.Feb.window()
```

Example 2: Create a cumulative (expanding) window

This example creates a cumulative (expanding) window with rows between unbounded preceding and 3 preceding, with *partition_columns* and *order_columns* arguments, and with default sorting.

```
>>> window = df.window(partition_columns="Feb",
...                     order_columns=["Feb", "datetime"],
...                     window_start_point=None,
...                     window_end_point=-3)
```

Example 3: Create a moving (rolling) window

This example creates a moving (rolling) window with rows between current row and 3 following, with sorting done on 'Feb', 'datetime' columns in descending order, and with *partition_columns* argument.

```
>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        window_start_point=0,
                        window_end_point=3)
```

Example 4: Create a remaining (contracting) window

This example creates a remaining (contracting) window with rows between current row and unbounded following, with sorting done on 'Feb', 'datetime' columns in ascending order, and NULL values in 'Feb', 'datetime' columns appear at last.

```
>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        nulls_first=False,
                        window_start_point=0,
                        window_end_point=None)
```

Example 5: Create a grouping window

This example creates a grouping window, with sorting done on 'Feb', 'datetime' columns in ascending order, and NULL values in 'Feb', 'datetime' columns appear at last.

```
>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        nulls_first=False,
                        window_start_point=None,
                        window_end_point=None)
```

Example 6: Create a window, ignoring all parameters

```
>>> window = df.window(partition_columns="Feb",
                        order_columns=["Feb", "datetime"],
                        sort_ascending=False,
                        nulls_first=False,
                        ignore_window=True)
```

Example 7: Perform sum of 'Feb' and attach new column to the DataFrame

```
>>> window = df.Feb.window()

>>> df.assign(feb_sum=window.sum())
      Feb      Jan      Mar      Apr      datetime      feb_sum
```

```
accounts
Jones LLC    200.0  150.0  140.0  180.0  04/01/2017  1000.0
Red Inc      200.0  150.0  140.0    NaN  04/01/2017  1000.0
Yellow Inc   90.0    NaN    NaN    NaN  04/01/2017  1000.0
Orange Inc  210.0    NaN    NaN  250.0  04/01/2017  1000.0
Blue Inc     90.0   50.0   95.0  101.0  04/01/2017  1000.0
Alpha Co    210.0  200.0  215.0  250.0  04/01/2017  1000.0
```

Example 8: Perform min and max operations on column 'Apr' and attach both columns to the DataFrame

```
>>> window = df.Apr.window()

>>> df.assign(apr_min=window.min(), apr_max=window.max())
      Feb    Jan    Mar    Apr  datetime  apr_max  apr_min
accounts
Jones LLC  200.0  150.0  140.0  180.0  04/01/2017    250    101
Red Inc    200.0  150.0  140.0    NaN  04/01/2017    250    101
Yellow Inc  90.0    NaN    NaN    NaN  04/01/2017    250    101
Orange Inc 210.0    NaN    NaN  250.0  04/01/2017    250    101
Blue Inc   90.0   50.0   95.0  101.0  04/01/2017    250    101
Alpha Co   210.0  200.0  215.0  250.0  04/01/2017    250    101
```

Example 9: Perform count and max operations on column 'accounts' grouped by 'accounts' and attach column to DataFrame

```
>>> df = df.groupby("accounts")

>>> window = df.accounts.window()

>>> df.assign(accounts_max=window.max(), accounts_count=window.count())
   accounts  accounts_count  accounts_max
0  Jones LLC              6  Yellow Inc
1    Red Inc              6  Yellow Inc
2  Yellow Inc              6  Yellow Inc
3  Orange Inc              6  Yellow Inc
4   Blue Inc              6  Yellow Inc
5   Alpha Co              6  Yellow Inc
```

Window Aggregate Functions

teradataml supports following window aggregate functions that can be executed on top of teradataml Window object created using `DataFrame.window()` and `DataFrameColumn.window()`.

See the [teradataml: Window Aggregates](#) section of *Teradata Package for Python Function Reference*, B700-4008) at <https://docs.teradata.com/> for detailed description and usage examples of these functions.

| Sr. No. | Function Name | Description |
|---------|-----------------------------|---|
| 1 | <code>corr()</code> | Returns the Sample Pearson product moment correlation coefficient of its arguments for all non-null data point pairs in a teradataml DataFrame or ColumnExpression over the specified window. |
| 2 | <code>count()</code> | Returns the total number of qualified rows in a teradataml DataFrame or ColumnExpression over the specified window. |
| 3 | <code>covar_pop()</code> | Returns the population covariance of its arguments for all non-null data point pairs over the specified window. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |
| 4 | <code>covar_samp()</code> | Returns the sample covariance of its arguments for all non-null data point pairs over the specified window. Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair. |
| 5 | <code>cume_dist()</code> | Returns the cumulative distribution of values in a teradataml DataFrame or ColumnExpression over the specified window. |
| 6 | <code>dense_rank()</code> | Returns the ordered ranking of all the rows in a teradataml DataFrame or ColumnExpression, according to "order_columns", over the specified window. |
| 7 | <code>first_value()</code> | Returns the first value of an ordered set of values in a teradataml DataFrame or ColumnExpression over the specified window. |
| 8 | <code>lag()</code> | The lag function accesses data from the row preceding the current row at a specified offset value over the specified window in a teradataml DataFrame or ColumnExpression. |
| 9 | <code>last_value()</code> | Returns the last value of an ordered set of values in a teradataml DataFrame or ColumnExpression over the specified window. |
| 10 | <code>lead()</code> | The lead function accesses data from the row following the current row at a specified offset value over the specified window in a teradataml DataFrame or ColumnExpression. |
| 11 | <code>max()</code> | Returns the maximum of values in teradataml DataFrame or ColumnExpression over the specified window. |
| 12 | <code>mean()</code> | Returns the arithmetic average of all values in teradataml DataFrame or ColumnExpression over the specified window. |
| 13 | <code>min()</code> | Returns the minimum of values in teradataml DataFrame or ColumnExpression over the specified window. |
| 14 | <code>percent_rank()</code> | Returns the relative rank of all the rows in a teradataml DataFrame or ColumnExpression, according to "order_columns", over the specified window. |
| 15 | <code>rank()</code> | Returns the rank (1 ... n) of all the rows in a teradataml DataFrame or ColumnExpression, according to "order_columns", over the specified window. |

| Sr. No. | Function Name | Description |
|---------|------------------|--|
| 16 | regr_avgx() | Returns the mean of the independent variable for all non-null data pairs of the dependent and an independent variable arguments over the specified window. |
| 17 | regr_avgy() | Returns the mean of the dependent variable for all non-null data pairs of the dependent and independent variable arguments over the specified window. |
| 18 | regr_count() | Returns the column-wise count of all non-null data pairs of the dependent and independent variable arguments over the specified window. |
| 19 | regr_intercept() | Returns the intercept of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments over the specified window. The intercept is the point at which the regression line through the non-null data pairs in the sample intersects the ordinate, or y-axis, of the graph. |
| 20 | regr_r2() | Returns the coefficient of determination for all non-null data pairs of the dependent and independent variable arguments over the specified window. |
| 21 | regr_slope() | <p>Returns the slope of the univariate linear regression line through all non-null data pairs of the dependent and an independent variable arguments over the specified window. When function is executed, "expression" is treated as an independent variable and dependent variable is:</p> <ul style="list-style-type: none"> • a ColumnExpression when invoked using a window created on ColumnExpression. • all columns of the teradataml DataFrame which are valid for this function, when executed on a window created on teradataml DataFrame. |
| 22 | regr_sxx() | <p>Returns the sum of the squares of the independent variable expression for all non-null data pairs of dependent and an independent variable arguments over the specified window. When function is executed, "expression" is treated as an independent variable and dependent variable is:</p> <ul style="list-style-type: none"> • a ColumnExpression when invoked using a window created on ColumnExpression. • all columns of the teradataml DataFrame which are valid for this function, when executed on a window created on teradataml DataFrame. |
| 23 | regr_sxy() | <p>Returns the sum of the products of the independent variable and the dependent variable for all non-null data pairs of the dependent and independent variable arguments over the specified window. When function is executed, "expression" is treated as an independent variable and dependent variable is:</p> <ul style="list-style-type: none"> • a ColumnExpression when invoked using a window created on ColumnExpression. • all columns of the teradataml DataFrame which are valid for this function, when executed on a window created on teradataml DataFrame. |
| 24 | regr_syy() | <p>Returns the sum of the squares of the dependent variable expression for all non-null data pairs of dependent and an independent variable arguments over the specified window. When function is executed, "expression" is treated as an independent variable and dependent variable is:</p> <ul style="list-style-type: none"> • a ColumnExpression when invoked using a window created on ColumnExpression. • all columns of the teradataml DataFrame which are valid for this function, |

| Sr. No. | Function Name | Description |
|---------|---------------|---|
| | | when executed on a window created on teradataml DataFrame. |
| 25 | row_number() | Returns the sequential row number, starting with first row as number one, for all the rows in a teradataml DataFrame or ColumnExpression, according to "order_columns", over the specified window. |
| 26 | std() | <p>Returns the standard deviation for the non-null data points in a teradataml DataFrame or ColumnExpression over the specified window.</p> <p>The standard deviation is the second moment of either a sample or population. For a population, it is a measure of dispersion from the mean of that population. For a sample, it is a measure of dispersion from the mean of that sample. The computation is more conservative for the population standard deviation to minimize the effect of outliers on the computed value.</p> |
| 27 | sum() | Returns the sum of values in a teradataml DataFrame or ColumnExpression over the specified window. |
| 28 | var() | <p>Returns the variance for the data points in a teradataml DataFrame or ColumnExpression over the specified window.</p> <p>By default calculates the variance of sample. Variance of a sample is a measure of dispersion from the mean of that sample. It is the square of the sample standard deviation. However, if parameter "population" is True, then the function calculates the variance of a population. Variance of a population is a measure of dispersion from the mean of that population.</p> <p>The computation is more conservative than that for the population standard deviation to minimize the effect of outliers on the computed value.</p> |

Context to Teradata Vantage

create_context

Use the `create_context()` function to create a connection to Vantage using the `teradatasql` and `teradatasqlalchemy` DBAPI and dialect combination.

You can pass all required arguments (*host*, *username*, *password*) to establish a connection to Vantage, or pass a sqlalchemy engine to the `tdsqlengine` parameter to override the default DBAPI and dialect combination. You can create connection to Vantage enabled with various security mechanisms.

The optional *logdata* argument specifies parameters to the LOGMECH command beyond those needed by the logon mechanism, such as user ID, password and tokens (in case of JWT) to successfully authenticate the user.

The optional *database* argument specifies the initial database to use after log on, instead of your default database.

In case you do not have permissions to create tables or views in your own database, but are allowed to create the required objects in other common database, you must use the argument *temp_database_name* to pass the name of the other database where temporary tables and views will be created. If this argument is not provided, intermediate database objects are created in your default database.

Note:

The *temp_database_name* and *database* arguments play a crucial role in determining which database is used by default to lookup for tables and views while creating teradataml DataFrame using 'DataFrame()' and 'DataFrame.from_table()' and which database is used to create all internal temporary objects.

| Arguments provided | | Where the actions are done | |
|---------------------------|-----------------|---|---|
| <i>temp_database_name</i> | <i>database</i> | Internal temporary objects are created in | Database object (table or view) lookup is done from |
| Yes | Yes | <i>temp_database_name</i> | <i>database</i> |
| No | Yes | <i>database</i> | <i>database</i> |
| Yes | No | <i>temp_database_name</i> | User's default database |
| No | No | User's default database | User's default database |

Note:

teradataml requires that the user has certain permissions on the user's default database or the initial default database specified using *database*, or the temporary database specified using *temp_database_name*.

These permissions allow the user to:

- Create tables and views to save results of teradataml analytic functions;
- Create views in the background for results of DataFrame APIs such as 'assign()', 'filter()', etc., whenever the result for these APIs are accessed using a 'print()';
- Create views in the background on the query passed to the 'DataFrame.from_query()' API.

It is expected that the user has the correct permissions to create these objects in the database to be used.

The access to the views created may also require issuing additional `GRANT SELECT ... WITH GRANT OPTION` permission depending on which database is used and which object the view being created is based on.

Note:

Passwords passed as part of either *password* or *logdata* argument can be in encrypted format using Stored Password Protection.

The following examples section demonstrates passing encrypted password to the *password* argument.

For more information on Stored Password Protection and how to generate key and encrypted password file used in examples, see the Stored Password Protection section under the teradatasql project, on <https://pypi.org/user/teradata/>.

Note:

When overwriting an existing context associated with a Vantage connection, most of the operations on any teradataml DataFrames created before will not work.

Note:

Teradata recommends calling `remove_context()` to end a session, so that intermediate views and tables created by teradataml are garbage collected.

Note:

Executing `create_context()` triggers garbage collection in teradataml.

See [Garbage Collection in teradataml](#) for more details.

Note:

teradataml expects the client environments are already setup with appropriate security mechanisms and are in working conditions.

For more information, see *Teradata Vantage™ - Analytics Database Security Administration*.

Special characters that may be used in the password are encoded by default. The encoding is done using `urllib.parse` library and can be disabled by setting the optional argument `url_encode` to `False`. For example, if the password is: "kx%jj5/g", then this password is encoded as "kx%25jj5%2Fg", where, '%25' represents the '%' character and '%2F' represents the '/' character. Details of how the special characters are replaced can be found on the [URL escape codes](#) page.

Note:

When password contains space:

- The space should not be encoded.
- Optional argument `url_encode` must be set to `False`.
- All other special characters, if present in this password, must be manually encoded, following the [URL escape codes](#).

See example 16 for a detailed demonstration.

Note:

When password contains unreserved characters like tilde ("~"), dot ("."), underscore ("_"), hyphen ("-"):

- Unreserved characters are not URL encoded by default.
- If unreserved character in the password needs to be encoded, it must be encoded manually encoded and encoded password must be passed along with optional argument `url_encode` set to `False`.
- Unreserved characters differ in different Python versions.

For example, the encoding standards for Python version 3.7 can be found in the following link: <https://docs.python.org/3/library/urllib.parse.html#url-quoting>.

Teradata recommends checking the encoding standards for your Python version before manual encoding special characters.

Example 1: Create a context using hostname, username and password

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', username='tduser', password = 'tdpassword')
```

Example 2: Create a context using already created sqlalchemy engine

```
>>> from sqlalchemy import create_engine
>>> sqlalchemy_engine = create_engine('teradatasql://' + tduser + ':' +
tdpassword + '@'+tdhost

>>> from teradataml.context.context import *
>>> create_context(tdsqlengine = sqlalchemy_engine)
```

Example 3: Create a context with default logmech 'TD2'

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', username='tduser', password =
'tdpassword', logmech='TD2')
```

Example 4: Create a context with default logmech 'TDNEGO'

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', username='tduser', password =
'tdpassword', logmech='TDNEGO')
```

Example 5: Create a context with default logmech 'LDAP'

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', username='tduser', password =
'tdpassword', logmech='LDAP')
```

Example 6: Create a context with default logmech 'KRB5'

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', logmech='KRB5')
```

Example 7: Create a context with default logmech 'JWT'**Note:**

You must use the 'logdata' argument when using 'JWT' as logon mechanism.

```
>>> from teradataml.context.context import *
>>> create_context(host = 'tdhost', logmech='JWT', logdata='token=eyJpc...h8dA')
```

Example 8: Create a context using encrypted password and key passed to the 'password' argument

Specify the password in the following format:

```
ENCRYPTED_PASSWORD(file:<PasswordEncryptionKeyFileName>,
file:<EncryptedPasswordFileName>)
```

Where:

- *PasswordEncryptionKeyFileName* specifies the name of a file that contains the password encryption key and associated information
- *EncryptedPasswordFileName* specifies the name of a file that contains the encrypted password and associated information.

Each filename must be preceded by the 'file:' prefix. The *PasswordEncryptionKeyFileName* must be separated from the *EncryptedPasswordFileName* by a single comma.

```
>>> td_context = create_context(host = 'tdhost', username='tduser', password =
"ENCRYPTED_PASSWORD(file:PassKey.properties, file:EncPass.properties)")
```

Example 9: Create a context using encrypted password in LDAP logon mechanism

```
>>> td_context = create_context(host = 'tdhost',
username='tduser', password = "ENCRYPTED_PASSWORD(file:PassKey.properties,
file:EncPass.properties)", logmech='LDAP')
```

Example 10: Create a context to connect to a different initial database, using hostname, username and password

This example creates a context using hostname, username and password, and connect to a different initial database by setting the *database* argument.

```
>>> td_context = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', database = 'database_name')
```

Example 11: Creates a context to connect to a different initial database, using already created sqlalchemy engine

This example creates a context using already created sqlalchemy engine, and connect to a different initial database by setting the *database* argument.

```
>>> from sqlalchemy import create_engine
>>> sqlalchemy_engine = create_engine('teradata:sql://'+ tduser +':'+
```

```
tdpassword + '@'+tdhost + '/?DATABASE=database_name')
>>> create_context(tdsqlengine = sqlalchemy_engine)
```

Example 12: Create a context to connect to a different initial database, using 'LDAP' logmech

This example creates a context with 'LDAP' logmech, and connect to a different initial database by setting the *database* argument.

```
>>> td_context = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', logmech='LDAP', database = 'database_name')
```

Example 13: Create a context using 'tera' mode

This example creates a context using 'tera' mode with *log* value set to 8 and *lob_support* disabled.

```
>>> td_context = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', tmode = 'tera', log = 8, lob_support = False)
```

Example 14: Create a context when password contains special characters

This example creates a context when password has special characters, and the example password "alice@pass" is encoded by default.

```
>>> td_context = create_context(host = 'tdhost', username='alice_pass', password
= 'alice@pass')
UserWarning: Warning: Password is URL encoded.
```

Example 15: Create a context when password contains space and special characters

This example creates a context with password containing space and special characters.

In this scenario, optional argument *url_encode* must be set to 'False' and special characters must be manually encoded.

For example, if the password is: "kx%jj5/ g", then this password is encoded as "kx%25jj5%2F g".

Where,

- '%25' represents the '%' character
- '%2F' represents the '/' character
- Space is not encoded

```
>>> td_context = create_context(host = 'tdhost', username='alice_pass', password
= 'kx%25jj5%2F g', url_encode=False)
```

get_context

The `get_context` function returns the Vantage engine associated with the current context.

Example

Create a context using username & password.

```
>>> from teradataml.context.context import *
>>> from teradataml.dataframe.dataframe import DataFrame
>>> create_context(host = "myhostname", username="myusername", password
= "mypassword")
```

Get the connection engine.

```
>>> get_context()
Engine(teradatasql://myusername:***@myhostname)
```

get_connection

The `get_connection` function returns the Vantage connection associated with the current context.

Example

Create a context using username and password.

```
>>> from teradataml.context.context import *
>>> from teradataml.dataframe.dataframe import DataFrame
>>> create_context(host = "myhostname", username="myusername", password
= "mypassword")
```

Get the connection object.

```
>>> get_connection()
<sqlalchemy.engine.base.Connection at 0x258d86cd470>
```

set_context

The `set_context` function specifies a Vantage sqlalchemy engine as current context.

Note:

- When overwriting an existing context associated with a Vantage connection, most of the operations on any teradataml DataFrames created before will not work.
- Running the `set_context()` triggers garbage collection in teradataml. For more details of Garbage Collection, see [Garbage Collection in teradataml](#).

Example

The following example creates two contexts using different host, then sets the context to the first one.

```
>>> from teradataml.context.context import *
>>> from teradataml.dataframe.dataframe import DataFrame
```

Create a connection:

```
>>> td_connection1 = create_context(host = "myhostname", username="myusername",
password = "mypassword")
```

Overwrite the existing connection with new connection using 'create_context()':

```
>>> td_connection2 = create_context(host = "myhostname2",
username="myusername2", password = "mypassword2")
302: UserWarning: [Teradata][teradataml](TDML_2002) Overwriting an existing
context associated with Teradata connection. Most of the operations on any
teradataml DataFrames created before this will not work.
warnings.warn(Messages.get_message(MessageCodes.OVERWRITE_CONTEXT))
```

Get the engine currently associated with the context:

```
>>> get_context()
Engine(teradatasql://myusername2:***@myhostname2)
```

Overwrite the existing context, by passing a sqlalchemy engine object as input to 'set_context()', thereby creating a new context:

```
>>> set_context(tdsqlengine = td_connection1)
302: UserWarning: [Teradata][teradataml](TDML_2002) Overwriting an existing
context associated with Teradata connection. Most of the operations on any
teradataml DataFrames created before this will not work.
warnings.warn(Messages.get_message(MessageCodes.OVERWRITE_CONTEXT))

>>> get_context()
Engine(teradatasql://myusername:***@myhostname)
```

remove_context

The `remove_context` function removes the current context associated with the Vantage connection.

`remove_context()` not only closes the connection but also garbage collects the intermediate views and tables created by `teradataml`.

Teradata recommends calling `remove_context()` to end a session, so that intermediate views and tables created by `teradataml` are garbage collected.

Note:

Executing `remove_context()` triggers garbage collection in `teradataml`. See [Garbage Collection in teradataml](#) for more details.

Example

```
>>> from teradataml.context.context import *
>>> from teradataml.dataframe.dataframe import DataFrame
>>> td_connection = create_context(host = "myhostname", username="myusername",
password = "mypassword")
```

```
>>> get_context()
Engine(teradatasql://myusername:***@myhostname)
```

```
>>> remove_context()
True
```

```
>>> get_context()
```

teradataml General Functions

Data Transfer Utility

Saving DataFrame to Vantage

`copy_to_sql()`

Use the `copy_to_sql()` function to create a table in Vantage based on a teradataml DataFrame or a pandas DataFrame.

The function takes a teradataml DataFrame or a pandas DataFrame and a table name as arguments, and generates DDL and DML commands that creates a table in Vantage. You can also specify the name of the schema in the database to write the table to. If no schema name is specified, the default database schema is used.

Required Arguments:

- *df*: Specifies the Pandas or teradataml DataFrame object to be saved.
- *table_name*: Specifies the name of the table to be created in Vantage.

Optional Arguments:

- *schema_name*: Specifies the name of the SQL schema in Vantage to write to.
- *if_exists*: Specifies the action to take when table already exists in the database.

Possible values are:

- 'fail': If table exists, do nothing;
- 'replace': If table exists, drop it, recreate it, and insert data;
- 'append': If table exists, insert data. Create if does not exist.

The default value is 'append'.

Note:

Replacing a table with the contents of a teradataml DataFrame based on the same underlying table is not supported.

-
- *index*: Specifies whether to save pandas DataFrame index as a column or not. Possible values are True or False.

The default value is False.

Note:

Only use as True when attempting to save Pandas DataFrames (and not on teradataml DataFrames).

- *index_label*: Specifies the column labels for pandas DataFrame index columns. If no value is given and *index* is True, then a default label 'index_label' is used.

The default value is None.

Note:

Only use this argument when attempting to save Pandas DataFrames (and not on teradataml DataFrames).

Note:

If this argument is not specified (defaulted to None or is empty) and argument *index* is set to True, then the *names* property of the DataFrames index is used as the label(s), and if that too is None or empty, then:

- A default label 'index_label' or 'level_0' (when 'index_label' is already taken) is used when index is standard.
- Default labels 'level_0', 'level_1', etc. are used when index is multi-level index.

- *primary_index*: Specifies the column(s) to use as primary index while creating tables in Vantage. When None (default value), No Primary Index (NOPI) tables are created.

For example:

- `primary_index = 'my_primary_index'`
- `primary_index = ['my_primary_index1', 'my_primary_index2', 'my_primary_index3']`

- *temporary*: Specifies whether to create Vantage tables as permanent or volatile.

Possible values are:

- True: Creates volatile tables, and *schema_name* is ignored.
- False: Creates permanent tables.

The default value is False.

- *types*: Specifies required data types for requested columns to be saved in Vantage.

The default value is None.

This argument accepts a dictionary of column names and their required teradatasqlalchemy types as key-value pairs, allowing to specify a subset of the columns of a specific type.

Note:

- When only a subset of all columns are provided, the rest are defaulted to appropriate types.
- When *types* argument is not provided, all column types are appropriately assigned and defaulted. The column types are assigned as listed in the following table.

| pandas/NumPy Type | teradata sqlalchemy Type |
|----------------------------|---|
| int32 | INTEGER |
| int64 | BIGINT |
| bool | BYTEINT |
| float32/float64 | FLOAT |
| datetime64/datatime64[ns] | TIMESTAMP |
| datetime64[ns,<time_zone>] | TIMESTAMP(timezone=True) |
| Any other data type | VARCHAR(configure.default_varchar_size) |

- *primary_time_index_name*: Specifies a name for the Primary Time Index (PTI) when the table to be created must be a PTI table.

Note:

This argument is not required or used when the table to be created is not a PTI table. It will be ignored if specified without the argument *timecode_column*.

- *timecode_column*: Specifies the column in the DataFrame that reflects the form of the timestamp data in the time series.

Note:

This argument is required when the DataFrame must be saved as a PTI table.

This column will be the TD_TIMECODE column in the table created. It should be of the SQL type `TIMESTAMP(n)`, `TIMESTAMP(n) WITH TIME ZONE`, or `DATE`, corresponding to the Python types `datetime.datetime` or `datetime.date`, or Pandas dtype `datetime64[ns]`.

Note:

This argument is not required when the table to be created is not a PTI table. When specifying this argument, an attempt to create a PTI table will be made. If this argument is specified, *primary_index* will be ignored.

- *timezero_date*: Specifies the earliest time series data that the PTI will accept; a date that precedes the earliest date in the time series data. Value specified must be of the following format: DATE 'YYYY-MM-DD'.

The default value is: DATE '1970-01-01'.

Note:

- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It will be ignored if specified without the *timecode_column* argument.

-
- *timebucket_duration*: Specifies a duration that serves to break up the time continuum in the time series data into discrete groups or buckets.

Specified using the formal form *time_unit*(n), where n is a positive integer, and *time_unit* can be any of the following: CAL_YEARS, CAL_MONTHS, CAL_DAYS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS, OR MICROSECONDS.

Note:

- This argument is required if *columns_list* is not specified or is empty.
- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It will be ignored if specified without the *timecode_column* argument.

-
- *columns_list*: Specifies a list of one or more PTI table column names.

Note:

- This argument is required if *timebucket_duration* is not specified or is empty.
- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It will be ignored if specified without the *timecode_column* argument.

-
- *sequence_column*: Specifies the column of type Integer containing the unique identifier for time series data reading when they are not unique in time.
 - When specified, implies SEQUENCED, meaning more than one reading from the same sensor may have the same timestamp. This column will be the TD_SEQNO column in the table created.
 - When not specified, implies NONSEQUENCED, meaning there is only one sensor reading per timestamp. This is the default.

Note:

- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It will be ignored if specified without the *timecode_column* argument.

- *seq_max*: Specifies the maximum number of sensor data rows that can have the same timestamp. Can be used when 'sequenced' is True.

Accepted values range from 1 to 2147483647, with default value 20000.

Note:

- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It will be ignored if specified without the *timecode_column* argument.

- *set_table*: Specifies a flag to determine whether to create a SET or a MULTiset table.
 - When True, a SET table is created.
 - When False, a MULTiset table is created. This is the default value.

Note:

- Specifying *set_table*=True also requires specifying *primary_index* or *timecode_column*.
- Creating SET table (*set_table*=True) may result in
 - an error if the source is a Pandas DataFrame having duplicate rows;
 - loss of duplicate rows if the source is a teradataml DataFrame.
- This argument has no effect if the table already exists and *if_exists*='append'.

- *match_column_order*: Specifies whether the order of the columns in existing table matches the order of the columns in the "df" or not.

When set to False, the DataFrame to be loaded can have any order and number of columns.

The default value is True.

- *chunksize*: Specifies the number of rows to be loaded in a batch, if DataFrame is pandas DataFrame. The default value is 16383.

Example 1: Create a new table in Vantage based on a pandas DataFrame

You must import pandas and *copy_to_sql* first.

```
>>>import pandas as pd
>>>from teradataml.dataframe.copy_to import copy_to_sql
```

Assume you create the "sales" table as follows:

```
>>>sales = [{'accounts': 'Jones LLC', 'Jan': 150, 'Feb': 200.0, 'Mar': 140, 'datetime':'2017-04-01'},
             {'accounts': 'Alpha Co', 'Jan': 200, 'Feb': 210.0, 'Mar': 215, 'datetime': '2017-04-01'},
             {'accounts': 'Blue Inc', 'Jan': 50, 'Feb': 90.0, 'Mar': 95, 'datetime': '2017-04-01' }]
```

And a pandas DataFrame "pdf" is created from the table "sales", using command:

```
pdf = pd.DataFrame(sales)
```

Enter pdf to display the pandas DataFrame:

```
>>>pdf
      Feb  Jan  Mar  accounts  datetime
0  200.0  150  140  Jones LLC  2017-04-01
1  210.0  200  215   Alpha Co  2017-04-01
2   90.0   50   95   Blue Inc  2017-04-01
```

Use the copy_to_sql() function to create a new Vantage table "pandas_sales" based on the pandas DataFrame "pdf" that is created in the prerequisite.

```
>>>copy_to_sql(df = pdf, table_name = "pandas_sales",
primary_index="accounts", if_exists="replace")
```

Verify that the new table "pandas_sales" exists in Vantage using the DataFrame() function which creates a DataFrame based on an existing table.

```
>>>df = DataFrame("pandas_sales")
```

```
>>> df
      Feb  Jan  Mar  accounts  datetime
accounts
Jones LLC  200.0  150  140  2017-04-01 00:00:00.000000
Alpha Co   210.0  200  215  2017-04-01 00:00:00.000000
Blue Inc    90.0   50   95  2017-04-01 00:00:00.000000
```


Example 2: Use the optional index and index_label parameters, and use the index as Primary Index.

```
>>> copy_to_sql(df = pdf, table_name = "pandas_sales", index=True,
index_label="idx", primary_index="idx", if_exists="replace")
```

```
>>> df = DataFrame("pandas_sales")
>>> df
```

| | Feb | Jan | Mar | accounts | datetime |
|-----|-------|-----|-----|-----------|----------------------------|
| idx | | | | | |
| 0 | 200.0 | 150 | 140 | Jones LLC | 2017-04-01 00:00:00.000000 |
| 2 | 90.0 | 50 | 95 | Blue Inc | 2017-04-01 00:00:00.000000 |
| 1 | 210.0 | 200 | 215 | Alpha Co | 2017-04-01 00:00:00.000000 |

Example 3: Create a new table in Vantage based on a teradataml DataFrame

```
>>> df = DataFrame("sales")
>>> df
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|------------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50 | 95 | 101 | 04/01/2017 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 04/01/2017 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 04/01/2017 |
| Yellow Inc | 90.0 | None | None | None | 04/01/2017 |
| Orange Inc | 210.0 | None | None | 250 | 04/01/2017 |
| Red Inc | 200.0 | 150 | 140 | None | 04/01/2017 |

```
>>> copy_to_sql(df = df, table_name = "sales_df1",
primary_index="accounts", if_exists="replace")
```

```
>>> df1 = DataFrame("sales_df1")
>>> df1
```

| | Feb | Jan | Mar | Apr | datetime |
|------------|-------|------|------|------|----------|
| accounts | | | | | |
| Blue Inc | 90.0 | 50 | 95 | 101 | 17/01/04 |
| Orange Inc | 210.0 | None | None | 250 | 17/01/04 |
| Red Inc | 200.0 | 150 | 140 | None | 17/01/04 |
| Yellow Inc | 90.0 | None | None | None | 17/01/04 |
| Jones LLC | 200.0 | 150 | 140 | 180 | 17/01/04 |
| Alpha Co | 210.0 | 200 | 215 | 250 | 17/01/04 |

Example 4: Create a new Primary Time Index Table in Vantage from a Pandas DataFrame

```
>>> copy_to_sql(df = pdf, table_name = "pandas_sales_pti",
timecode_column="datetime", columns_list="accounts")
```

```
>>> pandas_sales_pti = DataFrame('pandas_sales_pti')
>>> pandas_sales_pti
```

| | | TD_TIMECODE | Feb | Jan | Mar |
|-----------|------------|-----------------|-------|-----|-----|
| accounts | | | | | |
| Jones LLC | 2017-04-01 | 00:00:00.000000 | 200.0 | 150 | 140 |
| Alpha Co | 2017-04-01 | 00:00:00.000000 | 210.0 | 200 | 215 |
| Blue Inc | 2017-04-01 | 00:00:00.000000 | 90.0 | 50 | 95 |

Note:

See also [to_sql\(\) DataFrame Method](#).

fastload()

The `fastload()` function writes records from a Pandas DataFrame to Vantage using Fastload, and can be used to quickly load large amounts of data in an empty table on Vantage.

FastLoad opens multiple data transfer connections to the database. The number of data transfer sessions can be set using argument `open_sessions`. If this argument is not set, by default, data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

Teradata recommends using `fastload()` function when number of rows in the Pandas DataFrame is greater than 100,000 for better performance. To insert lesser rows, you can use the [copy_to_sql\(\)](#) function for optimized performance. The data is loaded in batches.

Note:

- FastLoad API cannot load duplicate rows in the DataFrame if the table is a MULTISSET table with Primary Index.
- FastLoad API does not support all Analytics Database data types.
For example, target table having BLOB and CLOB data type columns cannot be loaded.
- If there are any incorrect rows due to constraint violations, data type conversion errors, etc., FastLoad protocol ignores those rows and inserts all valid rows.
- Rows in the DataFrame that failed to get inserted are categorized into errors and warnings by FastLoad protocol and these errors and warnings are stored into respective error and warning tables by FastLoad API.
- If 'save_errors' argument is set to True, the names of error and warning tables are shown once the fastload operation is complete. These tables will be persisted using copy_to_sql.

See the FastLoad section of <https://pypi.org/project/teradatasql/> for more information about FastLoad protocol through teradatasql driver.

Required Arguments:

- *df*: Specifies the pandas or teradataml DataFrame object to be saved in Vantage.
- *table_name*: Specifies the name of the table to be created in Vantage.

Optional Arguments:

- *schema_name*: Specifies the name of the SQL schema in Vantage to write to.
The default value is None, which means use the default Vantage schema.
- *if_exists*: Specifies the action to take when table already exists in the database.

Possible values are:

- 'fail': If table exists, do nothing;
- 'replace': If table exists, drop it, recreate it, and insert data;
- 'append': If table exists, insert data. Create if does not exist.

The default value is 'replace'.

- *index*: Specifies whether to save pandas DataFrame index as a column or not. Possible values are True or False.

The default value is False.

- *index_label*: Specifies the column labels for pandas DataFrame index columns.

The default value is None.

- *primary_index*: Specifies the column(s) to use as primary index while creating tables in Vantage. When None (default value), No Primary Index (NOPI) tables are created.

For example:

- `primary_index = 'my_primary_index'`
- `primary_index = ['my_primary_index1', 'my_primary_index2', 'my_primary_index3']`
- **types:** Specifies required data types for requested columns to be saved in Vantage.

The default value is None.

This argument accepts a dictionary of column names and their required terdatasqlalchemy types as key-value pairs (`{column_name1: type_value1, ... column_nameN: type_valueN}`), allowing to specify a subset of the columns of a specific type.

Note:

- When only a subset of all columns are provided, the rest are defaulted to appropriate types.
- When **types** argument is not provided, all column types are appropriately assigned and defaulted. The column types are assigned as listed in the following table.

| pandas/NumPy Type | terdatasqlalchemy Type |
|----------------------------|---|
| int32 | INTEGER |
| int64 | BIGINT |
| bool | BYTEINT |
| float32/float64 | FLOAT |
| datetime64[ns] | TIMESTAMP |
| datetime64[ns,<time_zone>] | TIMESTAMP(timezone=True) |
| Any other data type | VARCHAR(configure.default_varchar_size) |

- This argument does not have any effect when the table specified using *table_name* and *schema_name* exists and `if_exists = 'append'`.

-
- **batch_size:** Specifies the number of rows to be loaded in a batch. The value of *batch_size* must be a positive integer.

For better performance, Teradata recommends batch size to be at least 100,000.

If this argument is None, which is the default value, there are two cases based on the number of rows, say N, in the dataframe 'df':

- If N is greater than 100,000, the rows are divided into batches of equal size with each batch having at least 100,000 rows (except the last batch which might have more rows).
- If N is less than 100,000, the rows are inserted in one batch after notifying the user that insertion happens with degradation of performance.

If this argument is not None, the rows are inserted in batches of size given in the argument, irrespective of the recommended batch size.

The last batch will have rows less than the batch size specified, if the number of rows is not an integral multiples of the argument *batch_size*.

- *save_errors*: Specifies whether to persist the error and warning information in Vantage or not.
 - If set to False, which is the default value, error and warnings are not persisted as tables.
 - If set to True, the error and warnings information are persisted and names of error and warning tables are returned. Otherwise, the function returns None for the names of the tables.
- *open_sessions*: Specifies the number of Teradata data transfer sessions to be opened for fastload operation.

If this argument is not provided, the default value is the smaller of 8 or the number of AMPs available.

See the FastLoad section of <https://pypi.org/project/teradatasql/> for additional information about number of Teradata data transfer sessions opened during fastload.

Returns

FastLoad returns a dict containing the following attributes:

- *errors_dataframe*: It is a Pandas DataFrame containing error messages thrown by fastload. DataFrame is empty if there are no errors.
- *warnings_dataframe*: It is a Pandas DataFrame containing warning messages thrown by fastload. DataFrame is empty if there are no warnings.
- *errors_table*: Name of the table containing errors. It is None, if argument *save_errors* is set to 'False'.
- *warnings_table*: Name of the table containing warnings. It is None, if argument *save_errors* is set to 'False'.

Minimum version requirements for fastload()

teradatasql version 16.20.00.48 or later is required for fastload() function to work properly. If you have a lower version installed, then teradatasql raises OperationalError and fastload() call ends with the following error:

```
[Teradata Database] [Error 3706] Syntax error: expected something between the
beginning of the request and the word 'teradata_require_fastloadINSERT'.
```

pandas version 0.24 or later is required for fastload() API to work properly. If you have a lower version, fastload() API will fail with following error:

```
AttributeError: 'Index' object has no attribute 'to_list'
```

Install pandas >= 0.24 to solve this issue.

Example Setup

```
>>> from teradataml.dataframe.fastload import fastload
>>> from teradatasqlalchemy.types import *
>>> import pandas as pd
```

```
>>> df = {'emp_name': ['A1', 'A2', 'A3', 'A4'],
          'emp_sage': [100, 200, 300, 400],
          'emp_id': [133, 144, 155, 177],
          'marks': [99.99, 97.32, 94.67, 91.00]
          }
```

```
>>> pandas_df = pd.DataFrame(df)
```

Example 1: Save a Pandas DataFrame with default signature

```
>>> fastload(df = pandas_df, table_name = 'my_table')
```

Example 2: Save a Pandas DataFrame with primary_index

```
>>> pandas_df = pandas_df.set_index(['emp_id'])
```

```
>>> fastload(df = pandas_df, table_name = 'my_table_1', primary_index='emp_id')
```

Example 3: Save a Pandas DataFrame with index and primary_index

```
>>> fastload(df = pandas_df, table_name = 'my_table_2',
index=True, primary_index='index_label')
```

Example 4: Save a Pandas DataFrame with types, appending to an existing table

```
>>> fastload(df = pandas_df, table_name = 'my_table_3', schema_name = 'alice',
index = True, index_label = 'my_index_label', primary_index = ['emp_id'],
if_exists = 'append', types = {'emp_name': VARCHAR, 'emp_sage': INTEGER,
'emp_id': BIGINT, 'marks': DECIMAL})
```

Example 5: Save a Pandas DataFrame using levels in index of type MultiIndex, replacing an existing table

```
>>> pandas_df = pandas_df.set_index(['emp_id', 'emp_name'])
```

```
>>> fastload(df = pandas_df, table_name = 'my_table_4', schema_name = 'alice',
index = True, index_label = ['index1', 'index2'], primary_index = ['index1'],
if_exists = 'replace')
```

Example 6: Save a Pandas DataFrame by opening given number of Teradata data transfer sessions

This example saves a Pandas DataFrame by opening two Teradata data transfer sessions.

```
>>> fastload(df = pandas_df, table_name = 'my_table_5', open_sessions = 2)
```

fastexport()

The `fastexport()` function exports teradataml DataFrame to pandas DataFrame or CSV file using the FastExport data transfer protocol.

Teradata recommends using `fastexport()` function when number of rows in the teradataml DataFrame is at least 100,000. To extract lesser rows, you can ignore this function and use regular [to_pandas\(\)](#) or [to_csv\(\)](#) functions.

FastExport opens multiple data transfer connections to the database. The number of data transfer sessions can be set using the keyword argument `open_sessions`. If `open_sessions` argument is not set, by default, data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

Note:

- FastExport does not support all database data types.
For example, tables with BLOB and CLOB type columns cannot be extracted.
- FastExport cannot be used to extract data from a volatile or temporary table.
- For best efficiency, do not use `DataFrame.groupby()` and `DataFrame.sort()` with FastExport.

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about FastExport protocol through teradatasql driver.

Required Arguments:

- `df`: Specifies the pandas DataFrame object to be saved in Vantage.

Optional Arguments:

- `export_to`: Specifies a value that notifies where to export the data.

Permitted values are:

- "pandas": Export data to a pandas DataFrame.
- "csv": Export data to a given CSV file.

The default value is 'panda'.

- *index_column*: Specifies column(s) to be used as index column for the converted object.

Default values is None.

Note:

This argument is applicable only when *export_to* is set to "pandas".

- *catch_errors_warnings*: Specifies whether to catch errors and warnings (if any) raised by FastExport protocol while converting teradataml DataFrame.

Default values is False.

- When *export_to* is set to "pandas" and *catch_errors_warnings* is set to True, *fastexport()* returns a tuple containing:
 - a. Pandas DataFrame.
 - b. Errors(if any) in a list thrown by *fastexport*.
 - c. Warnings(if any) in a list thrown by *fastexport*.

When set to False, prints the *fastexport* errors and warnings to the standard output, if there are any.

- When *export_to* is set to "csv" and *catch_errors_warnings* is set to True, *fastexport()* returns a tuple containing:
 - a. Errors (if any) in a list thrown by *fastexport*.
 - b. Warnings(if any) in a list thrown by *fastexport*.
- *csv_file*: Specifies the name of CSV file to which data is to be exported.

Note:

This argument is required when *export_to* is set to "csv".

- *kwargs* specifies keyword arguments.
 - *sep*: Specifies a single character string used to separate fields in a CSV file, with default value ','.
 - *quotechar*: Specifies a single character string used to quote fields in a CSV file, with default value '"' (double quote).
 - *coerce_float*: Specifies whether to convert non-string, non-numeric objects to floating point.
 - *parse_dates*: Specifies columns to parse as dates.
 - *open_sessions*: Specifies the number of Teradata data transfer sessions to be opened for *fastexport*. This argument is only applicable in *fastexport* mode.

Note:

If *open_sessions* argument is not set, by default, the number of data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

Note:

- *sep* and *quotechar* cannot be line feed ('\n') or carriage return ('\r').
- *sep* and *quotechar* should not be the same.
- Length of *sep* and *quotechar* should be 1.

See the FastExport section of <https://pypi.org/project/teradatasql/> for more information about number of data transfer session opened during fastexport.

See https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html for more information about the *coerce_float* and *parse_dates* arguments.

Returns

The `fastexport()` function returns:

- When *export_to* is set to 'pandas' and *catch_errors_warnings* is set to 'True', the `fastexport()` function returns a tuple containing:
 - pandas DataFrame;
 - Errors, if any, in a list of strings thrown by fastexport;
 - Warnings, if any, in a list of strings thrown by fastexport.
- When *export_to* is set to 'pandas' and *catch_errors_warnings* is set to 'False', the `fastexport()` function returns a pandas DataFrame, and prints the fastexport errors and warnings to the standard output, if there is any.
- When *export_to* is set to 'csv' and *catch_errors_warnings* is set to 'True', `fastexport()` function returns a CSV file with name specified by argument *csv_file* and a tuple containing:
 - Errors, if any, in a list of strings thrown by fastexport;
 - Warnings, if any, in a list of strings thrown by fastexport.

Example Setup

```
>>> from teradataml import fastexport

>>> load_example_data("dataframe", "admissions_train")

>>> df = DataFrame("admissions_train")
```

Example 1: Export teradataml DataFrame to pandas DataFrame

```
>>> fastexport(df)
```

Example 2: Export teradataml DataFrame to pandas DataFrame with settings

This example exports the teradataml DataFrame 'df' to pandas DataFrame, setting index column with argument *index_column*, converting non-string, non-numeric objects to floating point using argument *coerce_float*, and catching errors and warnings thrown by fastexport.

```
>>> pandas_df, err, warn = fastexport(df, index_column="gpa", coerce_float=True)
```

```
# Print pandas DataFrame.
```

```
>>> pandas_df
```

```
# Print errors list.
```

```
>>> err
```

```
# Print warnings list.
```

```
>>> warn
```

Example 3: Export teradataml DataFrame to pandas DataFrame by opening specific number of sessions

This example exports the teradataml DataFrame 'df' to pandas DataFrame using two sessions.

```
>>> fastexport(df, open_sessions=2)
```

Example 4: Export teradataml DataFrame to a given CSV file

```
>>> fastexport(df, export_to="csv", csv_file="Test.csv")
```

Example 5: Export teradataml DataFrame to a given CSV file by opening specific number of sessions

```
>>> fastexport(df, export_to="csv", csv_file="Test_1.csv", open_sessions=2)
```

Example 6: Export teradataml DataFrame to a given CSV file and catch errors and warnings

```
>>> err, warn = fastexport(df, export_to="csv",
catch_errors_warnings=True, csv_file="Test_3.csv")
```

```
# Print errors list.
```

```
>>> err
```

```
# Print warnings list.
>>> warn
```

Example 7: Export teradataml DataFrame to CSV file with field separator and field quote character

This example exports the teradataml DataFrame 'df' to a CSV file with (|) as field separator and single quote (') as field quote character.

```
>>> fastexport(df, export_to="csv", csv_file="Test_4.csv", sep =
"|", quotechar="'")
```

Loading Data from CSV to Vantage

read_csv

The read_csv() function loads data from CSV file into Teradata Vantage. This function is used to quickly load large amounts of data in a table on Vantage using FastLoadCSV protocol.

When load data from CSV file, optional arguments can be used to identify fields in a CSV file.

- *sep* specifies a single character string used to separate fields in a CSV file, with default value ' , '.
- *quotechar* specifies a single character string used to quote fields in a CSV file, with default value ' ' (double quote).

Considerations when using a CSV file:

- Each record is on a separate line of the CSV file. Records are delimited by line breaks (CRLF).
- The last record in the file may or may not have an ending line break.
- The first line in the CSV must be header line. The header line lists the column names separated by the field separator (e.g. col1,col2,col3).

Limitations when using a CSV file with FastLoad:

- read_csv function cannot load duplicate rows in a DataFrame if the table is a MULTiset table having primary index.
- read_csv function does not support all Teradata Analytics Database data types.

For example, target table having BLOB and CLOB data type columns cannot be loaded.

- If there are some incorrect rows due to constraint violations, data type conversion errors, and so on, FastLoad protocol ignores those rows and inserts all valid rows.
- Rows in the DataFrame that failed to get inserted are categorized into errors and warnings by FastLoad protocol and these errors and warnings are stored into respective error and warning tables by FastLoad API.

Teradata recommends using FastLoad protocol when number of rows to be loaded is at least 100,000. FastLoad opens multiple data transfer connections to the database, when the argument *use_fastload* is set to 'True'. The number of data transfer sessions can be set using argument *open_sessions*. If this argument is not set, by default, data transfer sessions opened by teradataml is the smaller of 8 and the number of available AMPs in Vantage.

See the CSV BATCH INSERTS section and FastLoad section of <https://pypi.org/project/teradatasql/> for more information.

Required Arguments:

- *filepath*: Specifies the CSV filepath including name of the file to load the data from.
- *table_name*: Specifies the table name to load the data into.
- *types*: Specifies the data types for requested columns to be saved in Vantage.

The default value is None.

Note:

This argument is optional when *if_exists=append* and non-PTI table already exists; and is required otherwise.

This argument accepts a dictionary of column names and their required teradatasqlalchemy types as key-value pairs.

Note:

This should be `OrderedDict`, if CSV file does not contain header.

Optional Arguments:

- *sep*: Specifies a single character string used to separate fields in a CSV file, with default value ' , '.
- *quotechar*: Specifies a single character string used to quote fields in a CSV file, with default value ' \" ' (double quote).

Note:

- *sep* and *quotechar* cannot be line feed ("\\n") or carriage return ("\\r").
 - *sep* and *quotechar* should not be the same.
 - Length of *sep* and *quotechar* should be 1.
-

- *schema_name*: Specifies the name of the SQL schema in Vantage to write to.

The default value is None, which means use the default Vantage schema.

- *if_exists*: Specifies the action to take when table already exists in the database.

Permitted values are:

- 'fail': If table exists, raise `TeradataMIException`;

- 'replace': If table exists, drop it, recreate it, and insert data;
- 'append': If table exists, append the existing table.

The default value is 'replace'.

- *primary_index*: Specifies the column(s) to use as primary index while creating tables in Vantage. When None (default value), No Primary Index (NOPI) tables are created.

The default value is None.

For example:

- `primary_index = 'my_primary_index'`
- `primary_index = ['my_primary_index1', 'my_primary_index2', 'my_primary_index3']`
- *set_table*: Specifies a flag to determine whether to create a SET or a MULTiset table.
 - When True, a SET table is created.
 - When False, a MULTiset table is created. This is the default value.

Note:

- Specifying *set_table*=True also requires specifying *primary_index*.
 - Creating SET table (*set_table*=True) may result in loss of duplicate rows, if CSV contains duplicate rows.
 - This argument has no effect if the table already exists and *if_exists*='append'.
-

- *temporary*: Specifies whether to create a table as volatile.

The default value is False.

When set to True:

- FastloadCSV protocol is not used for loading the data.
 - *schema_name* is ignored.
 - *primary_time_index_name*: Specifies a name for the Primary Time Index (PTI) when the table is to be created as PTI table.
-

Note:

This argument is not required or used when the table to be created is not a PTI table. It is ignored if specified without the *timecode_column*.

- *timecode_column*: Specifies the column in the DataFrame that reflects the form of the timestamp data in the time series.

This column will be the TD_TIMECODE column in the table created. It should be of SQL type `TIMESTAMP(n)`, `TIMESTAMP(n) WITH TIMEZONE`, or `DATE`, corresponding to Python types `datetime.datetime` or `datetime.date`.

Note:

- This argument is required when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- When this argument is specified, an attempt to create a PTI table will be made.
- If this argument is specified, *primary_index* is ignored.

- *timezero_date*: Specifies the earliest time series data that the PTI will accept; a date that precedes the earliest date in the time series data.

Value specified must be of the following format: DATE 'YYYY-MM-DD'. Default value is DATE '1970-01-01'.

Note:

- This argument is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It is ignored if specified without the *timecode_column*.

- *timebucket_duration*: Specifies a duration that serves to break up the time continuum in the time series data into discrete groups or buckets.

Value specified must use the formal form *time_unit*(n), where n is a positive integer, and *time_unit* can be any of the following: CAL_YEARS, CAL_MONTHS, CAL_DAYS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS, or MICROSECONDS.

Note:

- This argument is required if *columns_list* is not specified or is None.
- It is used when the DataFrame must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It is ignored if specified without the *timecode_column*.

- *column_list*: Specifies a list of one or more PTI table column names.

Note:

- This argument is required if *timebucket_duration* not specified.
- It is used when the CSV data must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It is ignored if specified without the *timecode_column*.

- *sequence_column*: Specifies the column of type Integer containing the unique identifier for time series data reading when they are not unique in time.
 - When specified, implies SEQUENCED, meaning more than one reading from the same sensor may have the same timestamp. This column will be the TD_SEQNO column in the table created.
 - When not specified, implies NONSEQUENCED, meaning there is only one sensor reading per timestamp. This is the default.

Note:

- This argument is used when the CSV data must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It is ignored if specified without the *timecode_column*.

- *seq_max*: Specifies the maximum number of data rows that can have the same timestamp. Can be used when 'sequenced' is True.

Permitted values range from 1 to 2147483647, with default value 20000.

Note:

- This argument is used when the CSV data must be saved as a PTI table.
- This argument is not required or used when the table to be created is not a PTI table.
- It is ignored if specified without the *timecode_column* argument.

- *save_errors*: Specifies whether to persist the error and warning information in Vantage or not.
 - If set to False, which is the default value,
 - Errors or warnings (if any) are not persisted into tables;
 - Errors table generated by FastloadCSV are not persisted.
 - If set to True,
 - The errors or warnings information is persisted and names of error and warning tables are returned. Otherwise, the function returns None for the names of the tables.
 - The errors tables generated by FastloadCSV are persisted and name of error tables are returned. Otherwise, the function returns None for the names of the tables.
- *catch_errors_warnings*: Specifies whether to catch errors and warnings (if any) raised by FastLoad protocol while loading data into the Vantage table.

When set to False, which is the default value, function does not catch any errors and warnings, otherwise catches errors and warnings, if any, and returns as a dictionary along with teradataml DataFrame.

See the following Returns section for more details.

- *use_fastload*: Specifies whether to use Fastload CSV protocol or not. Default value is True.

Teradata recommends to use Fastload when number of rows to be loaded are at least 100,000. To load lesser rows set this argument to 'False'. Fastload opens multiple data transfer connections to the database.

Note:

- When this argument is set to True, you can load the data into table using FastloadCSV protocol:
 - Set table
 - Multiset table
 - When this argument is set to False, you can load the data in following types of tables:
 - Set table
 - Multiset table
 - PTI table
 - Volatile table
-

- *open_sessions*: Specifies the number of Teradata data transfer sessions to be opened for fastload operation.

If this argument is not provided, the default value is the smaller of 8 or the number of AMPs available.

See the FastLoad section of <https://pypi.org/project/teradatasql/> for additional information about number of Teradata data transfer sessions opened during fastload.

Returns

Based on the input, the `read_csv` function returns different output:

- If *use_fastload* is set to 'False', it returns teradataml DataFrame.
- If *use_fastload* is set to 'True' and *catch_errors_warnings* is set to 'False', it returns only teradataml DataFrame.
- If *use_fastload* is set to 'True' and *catch_errors_warnings* is set to 'True', it returns a tuple containing teradataml DataFrame and a dict containing the following attributes:
 - *errors_dataframe*: It is a Pandas DataFrame containing error messages thrown by fastload. DataFrame is empty if there are no errors.
 - *warnings_dataframe*: It is a Pandas DataFrame containing warning messages thrown by fastload. DataFrame is empty if there are no warnings.
 - *errors_table*: Name of the table containing errors. It is None, if argument *save_errors* is set to 'False'.
 - *warnings_table*: Name of the table containing warnings.

It is None, if argument `save_errors` is set to 'False'.

- `fastloadcsv_error_tables`: Name of the tables containing errors generated by FastLoadCSV.

It is empty list, if argument `save_errors` is set to 'False'.

Example Setup

```
>>> import csv
>>> from collections import OrderedDict
>>> from teradataml.dataframe.data_transfer import read_csv
>>> from teradatasqlalchemy.types import *

>>> csv_filename = "test_file.csv"
>>> table_name = "read_csv_table"
>>> record = [
    'id,fname,lname,marks',
    '101,abc,def,200',
    '102,lmn,pqr,105',
    '103,def,xyz,492',
    '101,abc,def,300'
]

>>> # Function to write list data into csv file.
>>> def write_data_into_csv(filename, record):
    with open(filename, 'w', newline='') as csvFile:
        csvWriter = csv.writer(csvFile, delimiter='\n')
        csvWriter.writerow(record)
>>>
>>> write_data_into_csv(csv_filename, record)
```

Example 1: Default execution with types passed as OrderedDict

This example loads the data from CSV file into a table, with argument `types` passed as `OrderedDict`.

```
>>> read_csv('test_file.csv', 'my_first_table', types)>>> types =
OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT)
```

Example 2: Catch all errors and warnings, and store in the table

This example loads the data from CSV file into a table using fastload CSV protocol, and catch all errors and warnings and store those in the table.

```
>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table1',
              types=types,
              save_errors=True,
              catch_errors_warnings=True)
```

Example 3: Load data into an existing table, replace the table

This example loads the data from CSV file into a table using fastload CSV protocol. If table exists, then replace the table.

```
>>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT)

>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              types=types,
              if_exists='replace')
```

Example 4: Load data into an existing table, append the table

This example loads the data from CSV file into a table using fastload CSV protocol. If table exists in specified schema, then append the table.

```
>>> # Create new table in Vantage.
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              types=types,
              if_exists='fail')

>>> # Write different data into csv file.
>>> record = [
    'id,fname,lname,marks',
    '501,orl,mal,740'
]

>>> write_data_into_csv(csv_filename, record)
>>>
>>> # Append the existing table.
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              if_exists='append')
```

Example 5: Load data into a SET table, catch all errors and warnings

This example loads the data from CSV file into a SET table using fastload CSV protocol. Catch all errors and warnings as well as store those in the table.

```
>>> # Write duplicate rows in csv file.
>>> record = [
    'id,fname,lname,marks',
    '101,abc,def,200',
    '102,lmn,pqr,105',
    '103,def,xyz,492',
    '101,abc,def,200'
]
>>> write_data_into_csv(csv_filename, record)
```

```
>>> # Create SET table.
>>> read_csv(filepath='test_file.csv',
             table_name='my_first_table',
             types=types,
             if_exists='replace',
             set_table=True,
             primary_index='id',
             save_errors=True,
             catch_errors_warnings=True)
```

Example 6: Load data into a table without FastLoad protocol

This example loads the data from CSV file with DATE and TIMESTAMP columns into a table without FastLoad protocol. If table exists in specified schema, then append to the table.

```
>>> # Write date and timestamp in csv file.
>>> record = [
    'id,fname,lname,marks,admission_date,admission_time',
    '101,abc,def,200,2009-07-29,2009-07-29 20:17:59',
    '102,lmn,pqr,105,2009-07-29,2009-07-29 20:17:59',
    '103,def,xyz,492,2009-07-29,2009-07-29 20:17:59',
    '101,abc,def,200,2009-07-29,2009-07-29 20:17:59'
]
>>> write_data_into_csv(csv_filename, record)

>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT,
                        admission_date=DATE, admission_time=TIMESTAMP)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              types=types,
              if_exists='replace',
              use_fastload=False)
```

```
>>> # Append the existing table.
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              if_exists='append',
              use_fastload=False)
```

Example 7: Load data into a temporary table without FastLoad protocol

This example loads the data from CSV file into a temporary table without FastLoad protocol. If table exists in specified schema, then append to the table.

```
>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT,
                        admission_date=DATE, admission_time=TIMESTAMP)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              types=types,
              if_exists='replace',
              temporary=True)
```

```
>>> # Append the existing table.
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table',
              if_exists='append',
              temporary=True)
```

Example 8: Load data into a PTI table

This example loads data from CSV file with TIMESTAMP columns into a PTI table. If specified table exists, then append to the table; otherwise, creates new table.

```
>>> types = OrderedDict(partition_id=INTEGER, adid=INTEGER, productid=INTEGER,
                        event=VARCHAR, clicktime=TIMESTAMP)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_read_csv_pti_table',
              types=types,
              if_exists='append',
```

```
timecode_column='clicktime',
columns_list='event',
use_fastload=False)
```

Example 9: Load data into a SET PTI table

This example loads data from CSV file with TIMESTAMP columns into a SET PTI table. If specified table exists, then append to the table; otherwise, creates new table.

```
>>> types = OrderedDict(partition_id=INTEGER, adid=INTEGER, productid=INTEGER,
event=VARCHAR, clicktime=TIMESTAMP)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_read_csv_pti_table',
              types=types,
              if_exists='append',
              timecode_column='clicktime',
              columns_list='event',
              set_table=True)
```

Example 10: Load data into a temporary PTI table

This example loads data from CSV file with TIMESTAMP columns into a temporary PTI table. If specified table exists, then append to the table; otherwise, creates new table.

```
>>> types = OrderedDict(partition_id=INTEGER, adid=INTEGER, productid=INTEGER,
event=VARCHAR, clicktime=TIMESTAMP)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_read_csv_pti_table',
              types=types,
              if_exists='append',
              timecode_column='clicktime',
              columns_list='event',
              temporary=True)
```

Example 11: Load data into a table by opening given number of Teradata data transfer sessions

This example loads the data from CSV file into a Vantage table by opening two Teradata data transfer sessions.

```
>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT)
```

```
>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table_with_open_sessions',
              types=types,
              open_sessions=2)
```

Example 12: Load data into table and set primary index

This example loads data from CSV file into a Vantage table and set primary index provided through the *primary_index* argument.

```
>>> types = OrderedDict(id=BIGINT, fname=VARCHAR, lname=VARCHAR, marks=FLOAT)

>>> read_csv(filepath='test_file.csv',
              table_name='my_first_table_with_primary_index',
              types=types,
              primary_index = ['fname'])
```

Data Transfer between Vantage and Cloud Storage

Python users with data in various S3 compatible object store can transfer the data to or from the Analytics Database using the Native Object Store feature in teradataml. S3 compatible object store include AWS S3, Google Cloud Storage and Azure Data Lake Storage Gen2, and so on.

The Native Object Store feature in teradataml provides ability for Python users to:

- Search and query CSV, JSON, and Parquet format datasets located on external S3-compatible object storage platforms.
- Write data stored on Vantage to external S3-compatible object storage platforms.

Following APIs in teradataml enable reading and writing datasets to external storage platforms:

- [ReadNOS](#)
- [WriteNOS](#)

Note:

- You should import these functions only after establishing the connection to Vantage. Importing these functions before context creation raises import error.
 - You can check whether these functions are available or not by running `display_analytic_functions()`.
 - If these functions are available for the corresponding Vantage version, then `display_analytic_functions()` displays the functions.
 - Otherwise, these functions are not displayed in `display_analytic_functions()` output.
-

Prerequisites

Before proceeding to examples, verify with your database administrator that you have the correct privileges for the following:

- [Create Authorization Object](#)
- [Create Function Mapping](#)

Create Authorization Object

An authorization object is used to control who can access an external object store. Before creating an authorization object, user must have permission from the external object store to access the data. The credentials are configured on the object store that you want to access.

For example, to access an Amazon S3 bucket, an Access Key and ID or an AWS Identity and Access Management (IAM) user credential is required, depending on your external object store.

Once your external storage allows user to access it, set up an authorization object with the credentials for your external object store.

Note:

Public buckets or containers in external object stores do not require credentials for access. To access a public bucket or container, put an empty string for USER and PASSWORD.

The following example demonstrates creation of authorization object within teradataml environment.

```
from teradataml import create_context

td_context = create_context("Your-host", "Your-user", "Your-password")

auth_obj_cmd = """CREATE AUTHORIZATION authorization_object
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';"""

td_context.execute(auth_obj_cmd)
```

Create Function Mapping

Function mapping is a new database object you can use to specify a simple name to execute a function.

An authorized database can be used to create function mappings for READ_NOS and WRITE_NOS functions with name user desires. User can also store argument value which the user believes to stay constant, for example, "authorization".

The following example demonstrates creation of function mapping for both functions.

```
# Create definer trusted authorization object.
def_trustes_auth_cmd = """CREATE AUTHORIZATION DefAuth
AS DEFINER TRUSTED
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';"""
```

```
fm_name = "<CUSTOM READ_NOS FUNCTION MAPPING NAME>"
location = "<EXTERNAL STORE LOCATION>"
ReadNOS_fm_create_cmd = '''CREATE FUNCTION MAPPING {fm_name}
FOR READ_NOS
EXTERNAL SECURITY DEFINER TRUSTED DefAuth
USING
ANY in TABLE,
LOCATION('{location}'),
BUFFERSIZE,
RETURNTYPE,
SAMPLE_PERC,
STOREDAS,
FULLSCAN,
MANIFEST,
ROWFORMAT,
HEADER;'''.format(fm_name, location)
td_context.execute(auth_obj_cmd)
```

```
fm_name = "<CUSTOM WRITE_NOS FUNCTION MAPPING NAME>"
WriteNOS_fm_create_cmd = '''CREATE FUNCTION MAPPING {fm_name}
FOR WRITE_NOS
EXTERNAL SECURITY DEFINER TRUSTED DefAuth
USING
ANY in TABLE,
LOCATION('{location}'),
STOREDAS('PARQUET'),
NAMING,
MANIFESTFILE,
MANIFESTONLY,
OVERWRITE,
INCLUDE_ORDERING,
INCLUDE_HASHBY,
MAXOBJECTSIZE,
COMPRESSION;'''.format(fm_name, location)
td_context.execute(auth_obj_cmd)
```

Set the function mappings created by setting configuration options.


```
from teradataml.options.configure import configure
```

```
configure.write_nos_function_mapping = "<CUSTOM WRITE_NOS FUNCTION  
MAPPING NAME>"
```

```
configure.read_nos_function_mapping = "<CUSTOM READ_NOS FUNCTION MAPPING NAME>"
```

Use function without specifying argument that is stored in function mapping.

```
wn_obj = WriteNOS(data=titanic_data, stored_as='PARQUET')
# Print the result DataFrame.
print(wn_obj.result)
```

```
rn_obj = ReadNOS()
# Print the result DataFrame.
print(obj.result)
```

ReadNOS

ReadNOS allows users to do the following:

- Perform an ad hoc query on all data formats with the data in-place on an external object storage;
- List all the objects and path structure of an object storage;
- List the object store;
- Discover the schema of the data;
- Read CSV, JSON, and Parquet data;
- Bypass creating a foreign table in the user.

Example 1: Read Parquet file from AWS S3 bucket

```
obj = ReadNOS(location='/S3/s3.amazonaws.com/Your-Bucket/location/',
               authorization='authorization_object',
               stored_as='PARQUET')
```

```
# Print the result DataFrame.
print(obj.result)
```

Example 2: Read CSV file with authorization using Access_Key and Access_ID

```
# Create a table to store the data.
td_context.execute("CREATE TABLE read_nos_support_tab (payload dataset storage
```

```
format csv) NO PRIMARY INDEX;")
read_nos_support_tab = DataFrame("read_nos_support_tab")

# Read the CSV data using "data" argument.
obj = ReadNOS(data=read_nos_support_tab,
               authorization='{"Access_ID": "YOUR-ID", "Access_Key": "YOUR-KEY"}',
               location='/S3/s3.amazonaws.com/Your-Bucket/csv-location/',
               stored_as='TEXTFILE')

# Print the result DataFrame.
print(obj.result)
```

WriteNOS

WriteNOS allows users to do the following:

- Extract selected or all columns from an user table or from derived results and write to an external object storage in Parquet data format;
- Write to Teradata supported external object storage, such as Amazon S3.

Example setup

```
from teradataml import DataFrame, load_example_data

# Load the example data.
load_example_data("teradataml", "titanic")

# Create teradataml DataFrame objects.
titanic_data = DataFrame.from_table("titanic")
```

Example 1: Write a DataFrame to Amazon S3 bucket

```
obj = WriteNOS(data=titanic_data,
                location='/S3/s3.amazonaws.com/Your-Bucket/location/',
                authorization='{"Access_ID": "YOUR-ID", "Access_Key": "YOUR-KEY"}',
                stored_as='PARQUET')
```

Example 2: Write a DataFrame to Amazon S3 bucket as CSV with authorization using Access_Key and Access_ID

```
obj = WriteNOS(data=titanic_data,
                location='/S3/s3.amazonaws.com/Your-Bucket/location/',
```

```
authorization='authorization_object',
stored_as='CSV')
```

Database Utility

db_drop_table()

Use the `db_drop_table()` function to drop a table from a given schema. Returns True if the operation is successful.

Required arguments:

- *table_name* specifies the table name to be dropped.

Optional arguments:

- *schema_name* specifies the database of the table to be dropped. If a database is not specified, the function drops table from the current database.

Example Prerequisites

- Import the `teradataml` module:

```
>>> from teradataml import *
```

- Load example data:

```
>>> load_example_data("dataframe", "admissions_train")
```

Example 1: Drop table "admissions_train" from current database

```
>>> db_drop_table(table_name = "admissions_train")
True
```

Example 2: Drop table "admissions_train" from schema "alice"

```
>>> db_drop_table(table_name = "admissions_train", schema_name = "alice")
True
```

db_drop_view()

Use the `db_drop_view()` function to drop a view from a given schema. Returns True if the operation is successful.

Required arguments:

- *view_name* specifies the view name to be dropped.

Optional arguments:

- *schema_name* specifies the database of the view to be dropped. If a database is not specified, the function drops view from the current database.

Example Prerequisites

Create a temporary view:

```
>>> connection_object.execute("create view temporary_view as (select 1 as
dummy_col1, 2 as dummy_col2);")
```

Example 1: Drop a view "temporary_view" from current database

```
>>> db_drop_view(view_name = 'temporary_view')
True
```

Example 2: Drop a view "temporary_view" from schema "alice"

```
>>> db_drop_view(view_name = 'temporary_view', schema_name = 'alice')
True
```

db_list_tables()

Use the `db_list_tables()` function to list tables and views from a specified schema.

Optional arguments:

- *schema_name* specifies the name of a database. If a database is not specified, the function lists tables from the current database. Default value is `None`.
- *object_name* specifies a table or view name or pattern to be used for filtering the list of objects. Pattern may contain '%' or '_' as pattern matching characters:
 - A '%' represents any string of zero or more arbitrary characters. Any string of characters is acceptable as replacement for the percent sign.
For example: '%abc' will return all table and view object names starting with any character and ending with 'abc'.
 - A '_' represents exactly one arbitrary character. Any single character is acceptable in the position in which the underscore sign appears.
For example: 'a_c' will return all table and view object names starting with 'a', ending with 'c' and has length of 3.
- *Object_type* specifies object type to apply the filter. Valid value includes:
 - `all`: list all the object types, which is the default value
 - `table`: list only tables
 - `view`: list only views

- volatile: list only volatile tables
- temp: list all teradataml temporary objects created in the specified database

Example 1: List all the objects in the default schema

```
>>> load_example_data("dataframe", "admissions_train")

>>> db_list_tables()
```

Example 2: List all the views in the default schema

```
>>> connection_object.execute("create view temporary_view as (select 1 as
dummy_col1, 2 as dummy_col2);")

>>> db_list_tables(None, None, 'view')
```

Example 3: List all the objects in the default schema with specific conditions

This example lists all the objects in the default schema whose names begin with 'abc' followed by one arbitrary character and any number of characters in the end.

```
>>> connection_object.execute("create view abcd123 as (select 1 as dummy_col1,
2 as dummy_col2);")

>>> db_list_tables(None, 'abc_%', None)
```

Example 4: List all the tables in the default schema with specific conditions

This example lists all the tables in the default schema whose names begin with 'adm' followed by one arbitrary character and any number of characters in the end.

```
>>> load_example_data("dataframe", "admissions_train")

>>> db_list_tables(None, 'adm_%', 'table')
```

Example 5: List all the views in the default schema with specific conditions

This example lists all the views in the default schema whose names begin with any character but end with 'abc'.

```
>>> connection_object.execute("create view view_abc as (select 1 as dummy_col1,
2 as dummy_col2);")

>>> db_list_tables(None, '%abc', 'view')
```

Example 6: List all the volatile tables in the default schema with specific conditions

This example lists all the volatile tables in the default schema whose names begin with 'abc' and ends with any arbitrary character and has a length of 4.

```
>>> connection_object.execute("CREATE volatile TABLE abcd(col0 int, col1 float)
NO PRIMARY INDEX;")

>>> db_list_tables(None, 'abc_', 'volatile')
```

Example 7: List all the temporary objects created by teradataml in the default schema with specific conditions

This example lists all the temporary objects created by teradataml in the default schema whose names begin and end with any number of arbitrary characters but contains 'filter' in between.

```
>>> db_list_tables(None, '%filter%', 'temp')
```

db_python_package_details()

Use the `db_python_package_details()` function to get the Python packages installed on Vantage, and their corresponding versions.

Note:

Use this function only when Python interpreter and add-on packages are installed on the Vantage node.

Optional arguments:

- *names* specifies the name(s) and pattern(s) of the Python package(s) for which version information is to be fetched from Vantage.

If this argument is not specified or None, versions of all installed Python packages are returned.

Example 1: Get the details of a Python package 'dill'

```
>>> db_python_package_details("dill")
package version
0    dill  0.2.8.2
```

Example 2: Get the details of all Python packages having string 'mpy'

```
>>> db_python_package_details(names = "mpy")
package version
0      simpy  3.0.11
```

```

1      numpy    1.16.1
2      gmpy2     2.0.8
3 msgpack-numpy 0.4.3.2
4      sympy     1.3

```

Example 3: Get the details of all Python packages having string 'numpy', string 'learn', or both

```

>>> db_python_package_details(["numpy", "learn"])
      package version
0  scikit-learn 0.20.3
1      numpy    1.16.1
2 msgpack-numpy 0.4.3.2

```

Example 4: Get the details of all Python packages installed on Vantage

```

>>> db_python_package_details()
      package version
0      packaging 18.0
1      cycpler   0.10.0
2      simpy     3.0.11
3 more-itertools 4.3.0
4      mpmath    1.0.0
5      toolz     0.9.0
6      wordcloud 1.5.0
7      mistune   0.8.4
8 singledispatch 3.4.0.3
9      attrs     18.2.0

```

display_analytic_functions()

Use the `display_analytic_functions()` function to display a list of analytic functions available to use on the Teradata Vantage system that the user is connected to.

Example Setup

```
>>> from teradataml import create_context, display_analytic_functions
```

Example 1: Display a list of available analytic functions

```
>>> create_context(host = host, username=user, password=password)
```

```

>>> display_analytic_functions()
List of available SQL analytic functions:

```

```

1: Antiselect
2: Attribution
3: DecisionForestPredict
4: DecisionTreePredict
5: GLMPredict
6: MovingAverage
7: NaiveBayesPredict
8: NaiveBayesTextClassifierPredict
9: NGrams
10: NPath
...

```

Example 2: When no analytic functions are available on the cluster

```

>>> display_analytic_functions()
No analytic functions available with connected Teradata Vantage system.

```

list_td_reserved_keywords()

Use the `list_td_reserved_keywords()` function to validate if the specified string is Teradata reserved keyword or not.

Note:

If a key is not specified in the function call, all the Teradata reserved keywords are displayed.

Optional arguments:

- *key* specifies a string to validate if it is a Teradata reserved keyword.
- *raise_error* specifies whether to raise exception or not.

When set to True, an exception is raised, if specified "key" is a Teradata reserved keyword, otherwise not.

The default value is False.

Example Setup

```

>>> from teradataml import list_td_reserved_keywords

```

Example 1: List all available Teradata reserved keyword

```

>>> list_td_reserved_keywords()
      restricted_word
0                ABS
1             ACCOUNT

```



```

2          ACOS
3          ACOSH
4      ADD_MONTHS
5          ADMIN
6          ADD
7      ACCESS_LOCK
8      ABORTSESSION
9          ABORT

```

Example 2: Validate if keyword "account" is a Teradata reserved keyword or not

```

>>> list_td_reserved_keywords("account")
True

```

Example 3: Validate and raise exception if keyword "account" is a Teradata reserved keyword

```

>>> list_td_reserved_keywords("account", raise_error=True)
TeradataMlException: [Teradata][teradataml](TDML_2121) 'account' is a Teradata
reserved keyword.

```

File Management Functions

install_file()

Use the `install_file()` function to install or replace external language script or model files in Vantage. On success, it prints a message confirming that file is installed or replaced.

Installed language script can be executed using [execute_script](#) method in [Script](#).

Required arguments:

- *file_identifier* specifies the name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database. The name should be unique within the database. It can be any valid Teradata identifier.
- *file_path* specifies the absolute or relative path of the file (including file name) to be installed. This file is identified in Vantage by *file_identifier*.

Note:

File can be on client or remote server. The file location should be specified accordingly.

Optional arguments:

- *file_on_client* specifies whether the file is present on client or at remote location on Vantage. The default value is True, which means on client. Set to False if the file to be installed is present at remote location on Vantage.
- *is_binary* specifies if the file to be installed is a binary file.
- *replace* specifies if the file is to be installed or replaced.
 - If set to True: the file is replaced based on value of *force_replace*;
 - If set to False: the file is installed.
- *force_replace* specifies if system should check for the file being used before replacing it:
 - If set to True: the file is replaced even if it is being executed.
 - If set to False: an error is thrown if it is being executed.

Note:

This argument is ignored if *replace* is set to False.

Example 1: Install a file using default text mode**Note:**

To run this example, "mapper.py" is required on client at the path specified by argument *file_path*.

- Create the "mapper.py" file:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

- Install the file:

```
>>> install_file (file_identifier='mapper', file_path='data/
scripts/mapper.py')
File mapper.py installed in Vantage
```

Example 2: Install a file using binary mode

```
>>> install_file (file_identifier='binaryfile', file_path='data/scripts/
binary_file.dms', file_on_client = True, is_binary = True)
File binary_file.dms installed in Vantage
```

Example 3: Replace a file with another one using default text mode

This example replaces the file 'mapper.py' with 'mapper_replace.py' found at the relative path using the default text mode.

Note:

To run this example, "mapper_replace.py" is required on client at the path specified by argument *file_path*.

- Create the "mapper_replace.py" file:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for newword in words:
        print ('%s,%s' % (newword, 1))
```

- Replace the existing file with the new file:

```
>>> install_file (file_identifier='mapper', file_path='data/scripts/
mapper_replace.py', file_on_client=True, is_binary= False,
replace=True, force_replace=True)
File mapper_replace.py replaced in Vantage
```

remove_file()

Use the `remove_file()` function to remove user installed files and scripts from Vantage.

Required arguments:

- *file_identifier* specifies the name associated with the user installed file. It cannot have a database name associated with it, as the file is always installed in the current database.
- *force_remove* specifies if system should check for the file being used before removing it.
 - If set to True, then the file is removed even if it is being executed.
 - If set to False, which is the default value, then an error is thrown if it is being executed.

Example 1: Remove an installed file using default text mode

- See "Example 1: Install a file using default text mode" in [install_file\(\)](#) to install the "mapper.py" file.
- Remove the installed "mapper.py" file:

```
>>> remove_file (file_identifier='mapper', force_remove=True)
File mapper removed from Vantage
```

Example 2: Remove a binary file

- See "Example 2: Install a file using binary mode" in [install_file\(\)](#) to install the "binary_file.dms" file.
- Remove the installed "binary_file.dms" file:

```
>>> remove_file (file_identifier='binaryfile', force_remove=False)
File binaryfile removed from Vantage
```

Miscellaneous Functions

print_options()

Use the `print_options()` function to display both configure and display options set in the current user's session.

Example 1: Run 'print_options()' after creating a connection object

If you run the `print_options()` function after creating a connection object, it displays values set for all the options.

```
>>> import teradataml as tdm1

>>> tdm1.print_options()
Display Options
-----
byte_encoding = base16
max_rows = 10
precision = 3
print_sqlmr_query = False
suppress_vantage_runtime_warnings = False

Configure Options
-----
column_casesensitive_handler = False
default_varchar_size = 1024
val_install_location = None
vantage_version = vantage1.3
```

Example 2: Run 'print_options()' without a working connection

If you run the `print_options()` function without a working connection, it displays values set for all the options except `vantage_verison`. A warning is thrown for `vantage_version` option.

```
>>> import teradataml as tdm1

>>> tdm1.print_options()
Display Options
-----
byte_encoding = base16
max_rows = 10
precision = 3
print_sqlmr_query = False
suppress_vantage_runtime_warnings = False

Configure Options
-----
column_casesensitive_handler = False
default_varchar_size = 1024
val_install_location = None
Option 'vantage_version' information requires a working connection object.
```

show_versions()

Use the `show_versions()` function to display client information, version information for client package dependencies and server information.

Server information is displayed only when a context is already created, otherwise only client information is displayed.

Example 1: Run 'show_versions()' without initiating connection

If you run `show_versions()` function without initiating connection, i.e., creating context, the function displays client information, version information for client package dependencies, and warning is thrown for server information.

```
>>> show_versions()
.....
UserWarning: Server information requires a working connection object.
  warnings.warn("Server information requires a working connection object.")

INSTALLED VERSIONS: Client
-----
python: 3.7.3.final.0
python-bits: 64
OS: Windows
OS-release: 10
machine: AMD64
```

```

processor: Intel64 Family 6 Model 94 Stepping 3, GenuineIntel
byteorder: little
LC_ALL: None
LANG: None
LOCALE: None.None

pandas: 0.24.2
sqlalchemy: 1.3.5
terdatasqlalchemy: 16.20.0.8
terdatasql: 16.20.0.55
teradataml: 16.20.0.4

INSTALLED VERSIONS: Server
-----

```

Example 2: Run 'show_versions()' after initiating connection

If you run the `show_versions()` function after initiating connection, i.e., creating context, the function displays client information, version information for client package dependencies and server information.

```

>>> from teradataml import show_versions
>>> ## Make sure to execute create_context() before this.

>>> show_versions()

INSTALLED VERSIONS: Client
-----
python: 3.7.3.final.0
python-bits: 64
OS: Windows
OS-release: 10
machine: AMD64
processor: Intel64 Family 6 Model 94 Stepping 3, GenuineIntel
byteorder: little
LC_ALL: None
LANG: None
LOCALE: None.None

pandas: 0.24.2
sqlalchemy: 1.3.5
terdatasqlalchemy: 16.20.0.8
terdatasql: 16.20.0.55
teradataml: 16.20.0.4

```

```

INSTALLED VERSIONS: Server
-----
BUILD_VERSION: 08.10.01.00-4180efe
RELEASE: Vantage 1.1 GA

```

view_log()

Use the `view_log()` function to view log on Vantage. Logs are pulled from 'scriptlog' file or 'byom.log' on database node.

Optional arguments:

- *log_type* specifies the log to view.
 - If set to 'script', script log will be pulled from database node;
 - If set to 'byom', byom log is pulled from database node.

Default value is 'script'.

- *num_lines* specifies the number of lines to be read and displayed from log.

The default value is 1000.

Example 1: View script log

```
view_log(log_type="script", num_lines=200)
```

Example 2: View byom log

```
view_log(log_type="byom", num_lines=200)
```

Apply SET Operations on teradataml DataFrames

teradataml offers various API's that allow users to perform set operations which internally uses Teradata SET operators, to combine similar data sets or find difference of a list of teradataml DataFrames or GeoDataFrames.

The SET operators are similar to the JOINS, the only difference is that JOIN combines the columns from different DataFrames or GeoDataFrames, whereas SET operators combine rows from different DataFrames or GeoDataFrames.

Supported SET operators include:

- [concat](#)
- [td_except](#)
- [td_intersect](#)
- [td_minus](#)

Note:

The data types of the columns which are being used in the SET operators should match.

concat

Use the `concat()` API to concatenate a list of teradataml DataFrames, GeoDataFrames, or both, along the index axis. The operation is performed by carrying out a database-style UNION or UNION ALL operation.

Note:

If the list contains both teradataml DataFrames and GeoDataFrames, that is, it contains geometry data, the function returns a GeoDataFrame. See example 6.

Example Prerequisites

```
>>> df = DataFrame("admissions_train")
>>> df
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 36 | no | 3.00 | Advanced | Novice | 0 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 38 | yes | 2.65 | Advanced | Beginner | 1 |
| 5 | no | 3.44 | Novice | Novice | 0 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 34 | yes | 3.85 | Advanced | Beginner | 0 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 26 | yes | 3.57 | Advanced | Advanced | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

```
>>> df1 = df[df.gpa == 4].select(['id', 'stats', 'masters', 'gpa'])
>>> df1
```

| | stats | masters | gpa |
|----|----------|---------|-----|
| id | | | |
| 13 | Advanced | no | 4.0 |
| 29 | Novice | yes | 4.0 |
| 15 | Advanced | yes | 4.0 |

```
>>> df2 = df[df.gpa < 2].select(['id', 'stats', 'programming', 'admitted'])
>>> df2
```

| | stats | programming | admitted |
|----|-------|-------------|----------|
| id | | | |


```

24  Advanced      Novice      1
19  Advanced      Advanced    0

```

Example 1: Run concat() with default values for optional arguments

```

>>> cdf = concat([df1,df2])
>>> cdf
      stats masters  gpa programming admitted
id
19  Advanced      None  NaN      Advanced      0
24  Advanced      None  NaN      Novice      1
13  Advanced       no  4.0        None      None
29   Novice      yes  4.0        None      None
15  Advanced      yes  4.0        None      None

```

Example 2: Run concat() with optional argument "join"

Set join = inner

```

>>> cdf = concat([df1,df2], join='inner')
>>> cdf
      stats
id
19  Advanced
24  Advanced
13  Advanced
29   Novice
15  Advanced

```

Example 3: Run concat() with optional argument "allow_duplicates"

- Set allow_duplicates = True (default)

```

>>> cdf = concat([df1,df2])
>>> cdf
      stats masters  gpa programming admitted
id
19  Advanced      None  NaN      Advanced      0
24  Advanced      None  NaN      Novice      1
13  Advanced       no  4.0        None      None
29   Novice      yes  4.0        None      None
15  Advanced      yes  4.0        None      None

```

```

>>> cdf = concat([cdf,df2])
>>> cdf

```

| | stats | masters | gpa | programming | admitted |
|----|----------|---------|-----|-------------|----------|
| id | | | | | |
| 19 | Advanced | None | NaN | Advanced | 0 |
| 13 | Advanced | no | 4.0 | None | None |
| 24 | Advanced | None | NaN | Novice | 1 |
| 24 | Advanced | None | NaN | Novice | 1 |
| 19 | Advanced | None | NaN | Advanced | 0 |
| 29 | Novice | yes | 4.0 | None | None |
| 15 | Advanced | yes | 4.0 | None | None |

- Set allow_duplicates = False

```
>>> cdf = concat([cdf,df2], allow_duplicates=False)
>>> cdf
```

| | stats | masters | gpa | programming | admitted |
|----|----------|---------|-----|-------------|----------|
| id | | | | | |
| 19 | Advanced | None | NaN | Advanced | 0 |
| 29 | Novice | yes | 4.0 | None | None |
| 24 | Advanced | None | NaN | Novice | 1 |
| 15 | Advanced | yes | 4.0 | None | None |
| 13 | Advanced | no | 4.0 | None | None |

Example 4: Run concat() with optional argument "sort"

Set sort=True

```
>>> cdf = concat([df1,df2], sort=True)
>>> cdf
```

| | admitted | gpa | masters | programming | stats |
|----|----------|-----|---------|-------------|----------|
| id | | | | | |
| 19 | 0 | NaN | None | Advanced | Advanced |
| 24 | 1 | NaN | None | Novice | Advanced |
| 13 | None | 4.0 | no | None | Advanced |
| 29 | None | 4.0 | yes | None | Novice |
| 15 | None | 4.0 | yes | None | Advanced |

Example 5: Perform concatenation of two GeoDataFrames

- Create GeoDataFrames

```
>>> geo_dataframe = GeoDataFrame('sample_shapes')

>>> geo_dataframe1 = geo_dataframe[geo_dataframe.skey
== 1004].select(['skey','linestrings'])
```

```
>>> geo_dataframe1
skey      linestrings
1004  LINESTRING (10 20 30,40 50 60,70 80 80)

>>> geo_dataframe2 = geo_dataframe[geo_dataframe.skey < 1010].select(['skey', 'polygons'])
>>> geo_dataframe2
skey      polygons
1009      MULTIPOLYGON (((0 0 0,0 20 20,20 20 20,20 0 0 0)),((50 50 50,50 90
90,90 90 90,90 50 90,50 50 50)))
1005  POLYGON ((0 0 0,0 0 20.435,0.0 20.435 0,0.0 20.435 20.435,20.435 0.0 0,20.435 0.0 20.435,20.435 20.435
0,20.435 20.435 20.435,0 0 0))
1004      POLYGON ((0 0 0,0 10 20,20 20 30,20 10 0,0 0 0),(5 5 5,5
10 10,10 10 10,10 10 5,5 5 5))
1002      POLYGON ((0 0,0 20,20 20,20 0,0
0),(5 5,5 10,10 10,10 5,5 5))
1001  POLYGON ((0 0,0 20,20 20,20 0,0 0))
1003      POLYGON ((0.6 0.8,0.6
20.8,20.6 20.8,20.6 0.8,0.6 0.8))
1007      MULTIPOLYGON (((1 1,1 3,6 3,6 0,1 1)),
((10 5,10 10,20 10,20 5,10 5)))
1006      POLYGON ((0 0 0,0 20,0 20 0,0 20 20,20 0 0,20
0 20,20 20 0,20 20 0 0 0))
1008      MULTIPOLYGON (((0 0,0 20,20 20,20 0,0 0)),((0.6 0.8,0.6
20.8,20.6 20.8,20.6 0.8,0.6 0.8)))
```

- Perform concatenation

```
>>> concat([geo_dataframe1,geo_dataframe2])
skey      linestrings      polygons
1009      None      MULTIPOLYGON (((0 0 0,0 20 20,20 20
20,20 0 0 0)),((50 50 50,50 90 90,90 90,90 50 90,50 50 50)))
1005      None  POLYGON ((0 0 0,0 20.435,0.0 20.435 0,0.0 20.435 20.435,20.435 0.0 0,20.435 0.0 20.435,20.435
0.0 0,20.435 0.0 20.435,20.435 20.435 0,20.435 20.435 20.435,0 0 0))
1004  LINESTRING (10 20 30,40 50 60,70 80
80)
1004      None      POLYGON ((0 0 0,0
10 20,20 20 30,20 10 0,0 0 0),(5 5 5,5 10 10,10 10 10,10 10 5,5 5 5))
1003      None
POLYGON ((0.6 0.8,0.6 20.8,20.6 20.8,20.6 0.8,0.6 0.8))
1001      None      POLYGON
((0 0,0 20,20 20,20 0,0 0))
1002      None
POLYGON ((0 0,0 20,20 20,20 0,0 0),(5 5,5 10,10 10,10 5,5 5))
1007      None
MULTIPOLYGON (((1 1,1 3,6 3,6 0,1 1)),((10 5,10 10,20 10,20 5,10 5)))
1006      None      POLYGON
((0 0 0,0 20,0 20 0,0 20 20,20 0 0,20 0 20,20 20 0,20 20 0 0 0))
1008      None      MULTIPOLYGON (((0 0,0
20,20 20,20 0,0 0)),((0.6 0.8,0.6 20.8,20.6 20.8,20.6 0.8,0.6 0.8)))
```

Example 6: Perform concatenation of a DataFrame and GeoDataFrame

- >>> normal_df=df.select(['id','stats'])

```
>>> normal_df
stats
id
34  Advanced
32  Advanced
11  Advanced
40   Novice
38  Advanced
36  Advanced
7   Novice
26  Advanced
```

19 Advanced

13 Advanced

```
>>> geo_df = geo_dataframe[geo_dataframe.skey < 1010].select(['skey', 'polygons'])
>>> geo_df
skey                                polygons
1003                                POLYGON ((0.6 0.8,0.6
20.8,20.6 20.8,20.6 0.8,0.6 0.8))
1008                                MULTIPOLYGON (((0 0,0 20,20 20,20 0,0 0)),((0.6 0.8,0.6
20.8,20.6 20.8,20.6 0.8,0.6 0.8)))
1006                                POLYGON ((0 0 0,0 0 20,0 20 0,0 20 20,20 0 0,20
0 20,20 0 0,20 20 0 0))
1009                                MULTIPOLYGON (((0 0 0,0 20 20,20 20 20,20 0 20,0 0 0)),((50 50 50,50 90
90,90 90 90,90 50 90,50 50 50)))
1005 POLYGON ((0 0 0,0 0 20.435,0.0 20.435 0,0.0 20.435 20.435,20.435 0.0 0,20.435 0.0 20.435,20.435 20.435
0,20.435 20.435 20.435,0 0 0))
1007                                MULTIPOLYGON (((1 1,1 3,6 3,6 0,1 1)),
((10 5,10 10,20 10,20 5,10 5)))
1001                                POLYGON ((0 0,0 20,20 20,20 0,0 0))
1002                                POLYGON ((0 0,0 20,20 20,20 0,0
0),(5 5,5 10,10 10,10 5,5 5))
1004                                POLYGON ((0 0 0,0 10 20,20 20 30,20 10 0,0 0 0),(5 5 5,5
10 10,10 10 10,10 5,5 5 5))
```

```
>>> idf = concat([normal_df,geo_df])
```

```
>>> idf
      stats      skey  polygons
id
38  Advanced  None    None
7    Novice   None    None
26  Advanced  None    None
17  Advanced  None    None
34  Advanced  None    None
13  Advanced  None    None
32  Advanced  None    None
11  Advanced  None    None
15  Advanced  None    None
36  Advanced  None    None
```

td_except

Use the `td_except()` function to return the rows that appear in the first teradataml DataFrame or GeoDataFrame, and not in other teradataml DataFrames or GeoDataFrames, along the index axis.

Note:

This function should be applied to data frames of the same type: either all teradataml DataFrames, or all GeoDataFrames.

Example Prerequisites

```
>>> from teradataml import load_example_data
```

```
>>> load_example_data("dataframe", "setop_test1")
>>> load_example_data("dataframe", "setop_test2")

>>> from teradataml.dataframe import dataframe
>>> from teradataml.dataframe.setop import td_except
```

Example 1: Run `td_except()` on rows from two DataFrames, using default signature

This example applies the except operation on rows from two teradataml DataFrames when using default signature of the function.

```
>>> df1 = DataFrame('setop_test1')
>>> df1
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 62 | no | 3.70 | Advanced | Advanced | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 69 | no | 3.96 | Advanced | Advanced | 1 |
| 61 | yes | 4.00 | Advanced | Advanced | 1 |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 66 | no | 3.87 | Novice | Beginner | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 59 | no | 3.65 | Novice | Novice | 1 |

```
>>> df2 = DataFrame('setop_test2')
>>> df2
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 12 | no | 3.65 | Novice | Novice | 1 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 14 | yes | 3.45 | Advanced | Advanced | 0 |
| 20 | yes | 3.90 | Advanced | Advanced | 1 |
| 18 | yes | 3.81 | Advanced | Advanced | 1 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 11 | no | 3.13 | Advanced | Advanced | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

```
>>> idf = td_except([df1[df1.id<55] , df2])
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 50 | yes | 3.95 | Beginner | Beginner | 0 |
| 54 | yes | 3.50 | Beginner | Advanced | 1 |
| 52 | no | 3.70 | Novice | Beginner | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |

Example 2: Run `td_except()` on rows from two DataFrames, discarding duplicate rows

This examples applies the except operation on rows from the two teradataml DataFrames from previous example, discarding duplicate rows from the result by passing `allow_duplicates = False`.

```
>>> idf = td_except([df1[df1.id<55] , df2], allow_duplicates=False)
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 54 | yes | 3.50 | Beginner | Advanced | 1 |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 50 | yes | 3.95 | Beginner | Beginner | 0 |
| 52 | no | 3.70 | Novice | Beginner | 1 |

Example 3: Run `td_except()` on more than two DataFrames

This example shows what happens when `td_except` is used on more than two teradataml DataFrames. In this example, you have three teradataml DataFrames as `df1`, `df2` & `df3`, the operation is applied on `df1` and `df2` first, and then the operation is applied again on the result and `df3`.

```
>>> df3 = df1[df1.gpa <= 3.9]

>>> # Effective operation here would be, (df1-df2)-df3
>>> idf = td_except([df1, df2, df3])
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 61 | yes | 4.00 | Advanced | Advanced | 1 |
| 50 | yes | 3.95 | Beginner | Beginner | 0 |
| 69 | no | 3.96 | Advanced | Advanced | 1 |

Example 4: Run `td_except` on two GeoDataFrames

- Create GeoDataFrames

```

>>> geo_dataframe = GeoDataFrame('sample_shapes')

>>> geo_dataframe1 = geo_dataframe[geo_dataframe.skey
== 1004].select(['skey','linestrings'])

>>> geo_dataframe1
skey      linestrings
1004  LINESTRING (10 20 30,40 50 60,70 80 80)

>>> geo_dataframe2 = geo_dataframe[geo_dataframe.skey
< 1010].select(['skey','linestrings'])

>>> geo_dataframe2
skey      linestrings
1009      MULTILINESTRING ((10 20 30,40 50 60),(70 80 80,90 100 110))
1005      LINESTRING (1 3 6,3 0 6,6 0 1)
1004      LINESTRING (10 20 30,40 50 60,70 80 80)
1002      LINESTRING (1 3,3 0,0 1)
1001      LINESTRING (1 1,2 2,3 3,4 4)
1003      LINESTRING (1.35 3.6456,3.6756 0.23,0.345 1.756)
1007      MULTILINESTRING ((1 1,1 3,6 3),(10 5,20 1))
1006      LINESTRING (1.35 3.6456 4.5,3.6756 0.23 6.8,0.345 1.756 8.9)
1008      MULTILINESTRING ((1 3,3 0,0 1),(1.35 3.6456,3.6756 0.23,0.345 1.756))

```

- Run `td_except`

```

>>> td_except([geo_dataframe2,geo_dataframe1])
skey      linestrings
1008      MULTILINESTRING ((1 3,3 0,0 1),(1.35 3.6456,3.6756 0.23,0.345 1.756))
1003      LINESTRING (1.35 3.6456,3.6756 0.23,0.345 1.756)
1005      LINESTRING (1 3 6,3 0 6,6 0 1)
1006      LINESTRING (1.35 3.6456 4.5,3.6756 0.23 6.8,0.345 1.756 8.9)
1009      MULTILINESTRING ((10 20 30,40 50 60),(70 80 80,90 100 110))
1001      LINESTRING (1 1,2 2,3 3,4 4)
1007      MULTILINESTRING ((1 1,1 3,6 3),(10 5,20 1))
1002      LINESTRING (1 3,3 0,0 1)

```

td_intersect

Use the `td_intersect()` function to find the data at the intersection of the list of teradataml DataFrames or GeoDataFrames along the index axis, and returns a DataFrame or a GeoDataFrame with rows common to all input DataFrames or GeoDataFrames.

Note:

This function should be applied to data frames of the same type: either all teradataml DataFrames, or all GeoDataFrames.

Example Prerequisites

```
>>> from teradataml import load_example_data

>>> load_example_data("dataframe", "setop_test1")
>>> load_example_data("dataframe", "setop_test2")

>>> from teradataml.dataframe import dataframe
>>> from teradataml.dataframe.setop import td_intersect
```

Example 1: Run `td_intersect()` on rows from two DataFrames, using default signature

This example gets the intersection of rows from two teradataml DataFrames when using default signature of the function.

```
>>> df1 = DataFrame('setop_test1')
>>> df1
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 62 | no | 3.70 | Advanced | Advanced | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 69 | no | 3.96 | Advanced | Advanced | 1 |
| 61 | yes | 4.00 | Advanced | Advanced | 1 |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 66 | no | 3.87 | Novice | Beginner | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 59 | no | 3.65 | Novice | Novice | 1 |

```
>>> df2 = DataFrame('setop_test2')
>>> df2
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 12 | no | 3.65 | Novice | Novice | 1 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 14 | yes | 3.45 | Advanced | Advanced | 0 |
| 20 | yes | 3.90 | Advanced | Advanced | 1 |

| | | | | | |
|----|-----|------|----------|----------|---|
| 18 | yes | 3.81 | Advanced | Advanced | 1 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 11 | no | 3.13 | Advanced | Advanced | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

```
>>> idf = td_intersect([df1, df2])
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 64 | yes | 3.81 | Advanced | Advanced | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 66 | no | 3.87 | Novice | Beginner | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 62 | no | 3.70 | Advanced | Advanced | 1 |

Example 2: Run `td_intersect()` on rows from two DataFrames, discarding duplicate rows

This examples applies the intersect operation on rows from the two teradataml DataFrames from previous example, discarding duplicate rows from the result by passing `allow_duplicates = False`.

```
>>> idf = td_intersect([df1, df2], allow_duplicates=False)
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 64 | yes | 3.81 | Advanced | Advanced | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 66 | no | 3.87 | Novice | Beginner | 1 |
| 62 | no | 3.70 | Advanced | Advanced | 1 |

Example 3: Run `td_intersect()` on more than two DataFrames

This example shows what happens when `td_intersect` is used on more than two teradataml DataFrames. In this example, you have three teradataml DataFrames as `df1`, `df2` & `df3`, the operation is applied on `df1` & `df2` first, and then the operation is applied again on the result & `df3`.

```
>>> df3 = df1[df1.gpa <= 3.5]
>>> df3
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 67 | yes | 3.46 | Novice | Beginner | 0 |
| 54 | yes | 3.50 | Beginner | Advanced | 1 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |

```
>>> # Effective operation here would be, (df1-df2)-df3
>>> idf = td_intersect([df1, df2, df3])
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 68 | no | 1.87 | Advanced | Novice | 1 |

Example 4: Perform intersection of two GeoDataFrames

- Create GeoDataFrames

```
>>> geo_dataframe = GeoDataFrame('sample_shapes')

>>> geo_dataframe1 = geo_dataframe[geo_dataframe.skey
== 1004].select(['skey', 'linestrings'])

>>> geo_dataframe1
skey      linestrings
1004  LINESTRING (10 20 30,40 50 60,70 80 80)

>>> geo_dataframe2 = geo_dataframe[geo_dataframe.skey
< 1010].select(['skey', 'linestrings'])

>>> geo_dataframe2
skey      linestrings
1009  MULTILINESTRING ((10 20 30,40 50 60),(70 80 80,90 100 110))
1005  LINESTRING (1 3 6,3 0 6,6 0 1)
1004  LINESTRING (10 20 30,40 50 60,70 80 80)
1002  LINESTRING (1 3,3 0,0 1)
1001  LINESTRING (1 1,2 2,3 3,4 4)
1003  LINESTRING (1.35 3.6456,3.6756 0.23,0.345 1.756)
1007  MULTILINESTRING ((1 1,1 3,6 3),(10 5,20 1))
1006  LINESTRING (1.35 3.6456 4.5,3.6756 0.23 6.8,0.345 1.756 8.9)
1008  MULTILINESTRING ((1 3,3 0,0 1),(1.35 3.6456,3.6756 0.23,0.345 1.756))
```

- Perform intersection

```
>>> td_intersect([geo_dataframe1,geo_dataframe2])
skey          linestrings
1004  LINESTRING (10 20 30,40 50 60,70 80 80)
```

td_minus

Use the `td_minus()` function to return the rows that appear in the first teradataml DataFrame or GeoDataFrame, and not in other teradataml DataFrames or GeoDataFrames, along the index axis.

Note:

This function should be applied to data frames of the same type: either all teradataml DataFrames, or all GeoDataFrames.

Example Prerequisites

```
>>> from teradataml import load_example_data

>>> load_example_data("dataframe", "setop_test1")
>>> load_example_data("dataframe", "setop_test2")

>>> from teradataml.dataframe import dataframe
>>> from teradataml.dataframe.setop import td_minus
```

Example 1: Run `td_minus()` on rows from two DataFrames, using default signature

This example applies the minus operation on rows from two teradataml DataFrames when using default signature of the function.

```
>>> df1 = DataFrame('setop_test1')
>>> df1
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 62 | no | 3.70 | Advanced | Advanced | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 69 | no | 3.96 | Advanced | Advanced | 1 |
| 61 | yes | 4.00 | Advanced | Advanced | 1 |
| 58 | no | 3.13 | Advanced | Advanced | 1 |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 68 | no | 1.87 | Advanced | Novice | 1 |
| 66 | no | 3.87 | Novice | Beginner | 1 |

| | | | | | |
|----|----|------|----------|--------|---|
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 59 | no | 3.65 | Novice | Novice | 1 |

```
>>> df2 = DataFrame('setop_test2')
>>> df2
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 12 | no | 3.65 | Novice | Novice | 1 |
| 15 | yes | 4.00 | Advanced | Advanced | 1 |
| 14 | yes | 3.45 | Advanced | Advanced | 0 |
| 20 | yes | 3.90 | Advanced | Advanced | 1 |
| 18 | yes | 3.81 | Advanced | Advanced | 1 |
| 17 | no | 3.83 | Advanced | Advanced | 1 |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 11 | no | 3.13 | Advanced | Advanced | 1 |
| 60 | no | 4.00 | Advanced | Novice | 1 |
| 19 | yes | 1.98 | Advanced | Advanced | 0 |

```
>>> idf = td_minus([df1[df1.id<55] , df2])
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 50 | yes | 3.95 | Beginner | Beginner | 0 |
| 54 | yes | 3.50 | Beginner | Advanced | 1 |
| 52 | no | 3.70 | Novice | Beginner | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |

Example 2: Run `td_minus()` on rows from two DataFrames, discarding duplicate rows

This examples applies the minus operation on rows from the two teradataml DataFrames from previous example, discarding duplicate rows from the result by passing `allow_duplicates = False`.

```
>>> idf = td_minus([df1[df1.id<55] , df2], allow_duplicates=False)
>>> idf
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 54 | yes | 3.50 | Beginner | Advanced | 1 |
| 51 | yes | 3.76 | Beginner | Beginner | 0 |
| 53 | yes | 3.50 | Beginner | Novice | 1 |
| 50 | yes | 3.95 | Beginner | Beginner | 0 |
| 52 | no | 3.70 | Novice | Beginner | 1 |

Example 3: Run `td_minus()` on more than two DataFrames

This example shows what happens when `td_minus` is used on more than two teradataml DataFrames. In this example, you have three teradataml DataFrames as `df1`, `df2` & `df3`, the operation is applied on `df1` and `df2` first, and then the operation is applied again on the result and `df3`.

```
>>> df3 = df1[df1.gpa <= 3.9]

>>> idf = td_minus([df1, df2, df3])
>>> idf
   masters   gpa   stats programming  admitted
id
61     yes  4.00  Advanced    Advanced         1
50     yes  3.95  Beginner    Beginner         0
69     no   3.96  Advanced    Advanced         1
>>>
```

Example 4: Run `td_minus` on two GeoDataFrames

- Create GeoDataFrames

```
>>> geo_dataframe = GeoDataFrame('sample_shapes')

>>> geo_dataframe1 = geo_dataframe[geo_dataframe.skey
== 1004].select(['skey', 'linestrings'])

>>> geo_dataframe1
   skey      linestrings
1004  LINESTRING (10 20 30,40 50 60,70 80 80)

>>> geo_dataframe2 = geo_dataframe[geo_dataframe.skey
< 1010].select(['skey', 'linestrings'])

>>> geo_dataframe2
   skey      linestrings
1009  MULTILINESTRING ((10 20 30,40 50 60),(70 80 80,90 100 110))
1005  LINESTRING (1 3 6,3 0 6,6 0 1)
1004  LINESTRING (10 20 30,40 50 60,70 80 80)
1002  LINESTRING (1 3,3 0,0 1)
1001  LINESTRING (1 1,2 2,3 3,4 4)
1003  LINESTRING (1.35 3.6456,3.6756 0.23,0.345 1.756)
1007  MULTILINESTRING ((1 1,1 3,6 3),(10 5,20 1))
1006  LINESTRING (1.35 3.6456 4.5,3.6756 0.23 6.8,0.345 1.756 8.9)
1008  MULTILINESTRING ((1 3,3 0,0 1),(1.35 3.6456,3.6756 0.23,0.345 1.756))
```

- Run `td_minus`

```
>>> td_minus([geo_dataframe2,geo_dataframe1])
skey                linestrings
1005                                LINESTRING (1 3 6,3 0 6,6 0 1)
1009                MULTILINESTRING ((10 20 30,40 50 60),(70 80 80,90 100 110))
1002                                LINESTRING (1 3,3 0,0 1)
1007                                MULTILINESTRING ((1 1,1 3,6 3),(10 5,20 1))
1008 MULTILINESTRING ((1 3,3 0,0 1),(1.35 3.6456,3.6756 0.23,0.345 1.756))
1006                LINESTRING (1.35 3.6456 4.5,3.6756 0.23 6.8,0.345 1.756 8.9)
1003                                LINESTRING (1.35 3.6456,3.6756 0.23,0.345 1.756)
1001                                LINESTRING (1 1,2 2,3 3,4 4)
```

Table Operators

Script

The `teradataml Script` function is an interface to the `SCRIPT` table operator (STO) object in the Analytics Database. See the `SCRIPT` section of Table Operators in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210 for details about `SCRIPT` table operator.

You can use `Script` function to execute natively in Vantage a user-installed script against data from an Analytics Database table via the `SCRIPT` table operator. To do this, the Teradata In-nodes Interpreter and Add-ons language packages must be installed in advance in the target Analytics Database nodes.

Required arguments:

- *script_command*: specifies the command or script to run,.
- *script_name*: specifies the name of the user script.
- *files_local_path* specifies the absolute local path where user script and all supporting files like model files, input data file reside.
- *returns*: specifies output column definition, which is a dictionary specifying column-name to `teradata sqlalchemy`-type mapping.

Note:

Users can pass a dictionary (`dict` or `OrderedDict`) to the *returns* argument, with the keys ordered to represent the order of the output columns. The preferred type is `OrderedDict`.

Optional arguments:

- *data*: specifies a `teradataml DataFrame` containing the input data for the script.
- *data_hash_column*: specifies the column to be used for hashing.

The rows in the data will be redistributed to AMPs based on the hash value of the column specified. The user-installed script file then runs once on each AMP. If there is no *data_partition_column*, then the entire result set, delivered by the function, constitutes a single group or partition.

Note:

data_hash_column can not be specified along with *data_partition_column*, *is_local_order* and *data_order_column*.

- *data_partition_column*: specifies Partition By columns for data.
-

Note:

- *data_partition_column* can not be specified along with *data_hash_column*.
 - *data_partition_column* can not be specified along with "*is_local_order* = True".
-

- *is_local_order*: specifies a boolean value to determine whether the input data is to be ordered locally or not.

Default value is False. When set to True, data is ordered locally.

Note:

- This argument is ignored, if *data_order_column* is None.
 - *is_local_order* can not be specified along with *data_hash_column*.
 - When *is_local_order* is set to 'True', *data_order_column* should be specified, and the columns specified in *data_order_column* are used for local ordering.
-

- *data_order_column*: specifies the Order By column for *data*, and can be used no matter *is_local_order* is set to 'True' or 'False'.
-

Note:

data_order_column can not be specified along with *data_hash_column*.

- *sort_ascending*: specifies a boolean value to determine if the result set is to be sorted on the *data_order_column* column in ascending or descending order.

When set to the default value 'True', the sorting is ascending. When set to 'False', the sortin is descending.

Note:

This argument is ignored, if *data_order_column* is None.

- *nulls_first*: specifies a boolean value to determine whether NULLS are listed first or last during ordering.

NULLS are listed first when this argument is set to 'True', and last when set to 'False'.

Note:

This argument is ignored, if *data_order_column* is None.

- *delimiter*: specifies a delimiter to use when reading columns from a row and writing result columns. The default value is '\t' (tab).

Note:

- This argument cannot be the same as *quotechar* argument.
 - This argument cannot be a newline character, which is '\n'.
-

- *auth*: specifies an authorization to use when running the script.
- *charset*: specifies the character encoding for data, could be either 'utf16' or 'latin'.
- *quotechar*: specifies a character that forces all input and output of the script to be quoted using this specified character.

Using this argument enables the Analytics Database to distinguish between NULL fields and empty strings. A string with length zero is quoted, while NULL fields are not.

If this character is found in the data, it will be escaped by a second quote character.

Note:

- This argument cannot be the same as *delimiter* argument.
 - This argument cannot be a newline character, which is '\n'.
-

For details of all arguments, see [Teradata Package for Python Function Reference](#).

Required permissions

Prior to running Script functions, specify search path and permissions as listed here:

Note:

Assume the user is 'TDAPUSER', and the database is 'TDAPUSERDB'.

- Teradata user should be given permission to execute Script Table Operator and the default authentication.

```
GRANT EXECUTE FUNCTION ON td_sysfnlib.script TO TDAPUSER;
```



```
GRANT EXECUTE ON SYSUIF.DEFAULT_AUTH to TDAPUSER;
```

- Additional permissions required to run Script on the database.

```
GRANT SELECT ON TDAPUSERDB TO TDAPUSER;
```

- Additional permissions required to install, remove and replace files.

```
GRANT EXECUTE PROCEDURE ON SYSUIF.INSTALL_FILE TO TDAPUSER;
```

```
GRANT EXECUTE PROCEDURE ON SYSUIF.REPLACE_FILE TO TFDAPUSER;
```

```
GRANT EXECUTE PROCEDURE ON SYSUIF.REMOVE_FILE TO TDAPUSER;
```

- You must set SEARCHUIFDBPATH prior to accessing any installed scripts in the target database, so that the Linux system user 'tdatuser' can find the scripts and run them through the STO. If the scripts reside in 'TDAPUSERDB', then set search path in corresponding Python script or Python console as follows:

```
get_connection().execute("SET SESSION SEARCHUIFDBPATH = TDAPUSERDB;")
```

- If the user ('me' in the example here) is using another database ('TDAPUSERDB' in the example here) as default database, the user should have the execute permissions on the Script function to the default database.

Here is the quick example.

1. Create the required users and databases as database administrator.

```
-- Create a user 'sysdba' with default database as 'sysdba'.
CREATE USER sysdba FROM DBC
AS PERM = 14e9
PASSWORD = sysdba
DEFAULT DATABASE = sysdba
NO BEFORE JOURNAL
NO AFTER JOURNAL;

-- Create a new database from the newly created user 'sysdba'.
CREATE DATABASE TDAPUSERDB FROM sysdba
AS PERM=1e9
NO BEFORE JOURNAL
NO AFTER JOURNAL;

-- Create a user with default database as newly created
database 'TDAPUSERDB'.
CREATE USER me FROM sysdba
AS PERM = 100e6
```

```
PASSWORD = me
DEFAULT DATABASE = TDAPUSERDB
NO BEFORE JOURNAL
NO AFTER JOURNAL;
```

2. Make sure the file 'mapper.py' is installed in the database 'TDAPUSERDB'. Refer to the [Database Utility](#) section for how to use `install_file()` to install file in Vantage. Ignore if the file is already present in Vantage.
3. The database administrator should provide the following permissions to the user 'me', if not already provided.

```
-- Permissions to the user 'me'.
GRANT EXECUTE FUNCTION ON td_sysfnlib.script TO me;
GRANT EXECUTE ON SYSUIF.DEFAULT_AUTH to me;

-- Additional permissions on the database 'TDAPUSERDB'.
GRANT CREATE TABLE ON TDAPUSERDB TO me; -- Needed if the table and required
permissions are not present.
GRANT CREATE VIEW ON TDAPUSERDB TO me; -- Needed if the permissions for view
creation is not present.
GRANT SELECT ON TDAPUSERDB TO me;
```

4. Set the search path and run the Script function as the user 'me'.

```
-- Set the search path.
SET SESSION SEARCHUIFDBPATH = TDAPUSERDB;

-- Create view on top of Script function.
-- Note that view creation is successful.
create view vw3 as SELECT * FROM Script(
  ON "TDAPUSERDB"."usecase5" AS "input"
  PARTITION BY ANY
  SCRIPT_COMMAND('Rscript ./TDAPUSERDB/mapper.R')
  delimiter(',')
  returns('oc1 VARCHAR(10), oc2 VARCHAR(10)')
) as sqlmr;
*** View has been created.
*** Total elapsed time was 1 second.

-- Seeing the contents of the view is not possible as the user 'me'
don't have execute permissions on TD_SYSFNLIB.SCRIPT and default auth
i.e., SYSUIF.DEFAULT_AUTH.
select * from vw3;
```

```
*** Failure 3523 An owner referenced by user does not have EXECUTE FUNCTION
WITH GRANT OPTION access to TD_SYSFNLIB.SCRIPT.
```

5. The user 'me' has to be given the following permissions, for the SELECT on view 'vw3' to work in step 4.

```
GRANT EXECUTE FUNCTION ON td_sysfnlib.script TO TDAPUSERDB WITH GRANT OPTION;
```

```
GRANT EXECUTE ON SYSUIF.DEFAULT_AUTH TO TDAPUSERDB WITH GRANT OPTION;
```

Sandbox Environment

teradataml provides user a sandbox environment that can be used to run user scripts outside Vantage. Thus the user can test the scripts in a Vantage-like environment before uploading the scripts for execution in the target Analytics Database.

The sandbox environment is based on a Docker image that has:

- Python interpreter (Version 3.6)
- Add-on packages for Python
- A script that enables user to run Python script within the container environment. Input to this Python script can be:
 - A file with input data
 - Data read from all AMPs

The Docker image can be downloaded from the **Teradata Package for Python - teradataml** page on <https://downloads.teradata.com/>.

The size of the Docker image is about 3.6GB. Due to the large size of this Docker image, time to download depends on network bandwidth. Teradata recommends downloading the Docker image beforehand, and saving it into a local folder.

Users can set up the Docker environment and test Python scripts by running the scripts either inside the Docker container, or directly on Vantage.

For details on installing Docker on various platforms, see the following links:

- Windows: <https://docs.docker.com/desktop/install/windows-install/>
- macOS: <https://docs.docker.com/desktop/install/mac-install/>
- Ubuntu: <https://docs.docker.com/engine/install/ubuntu/>

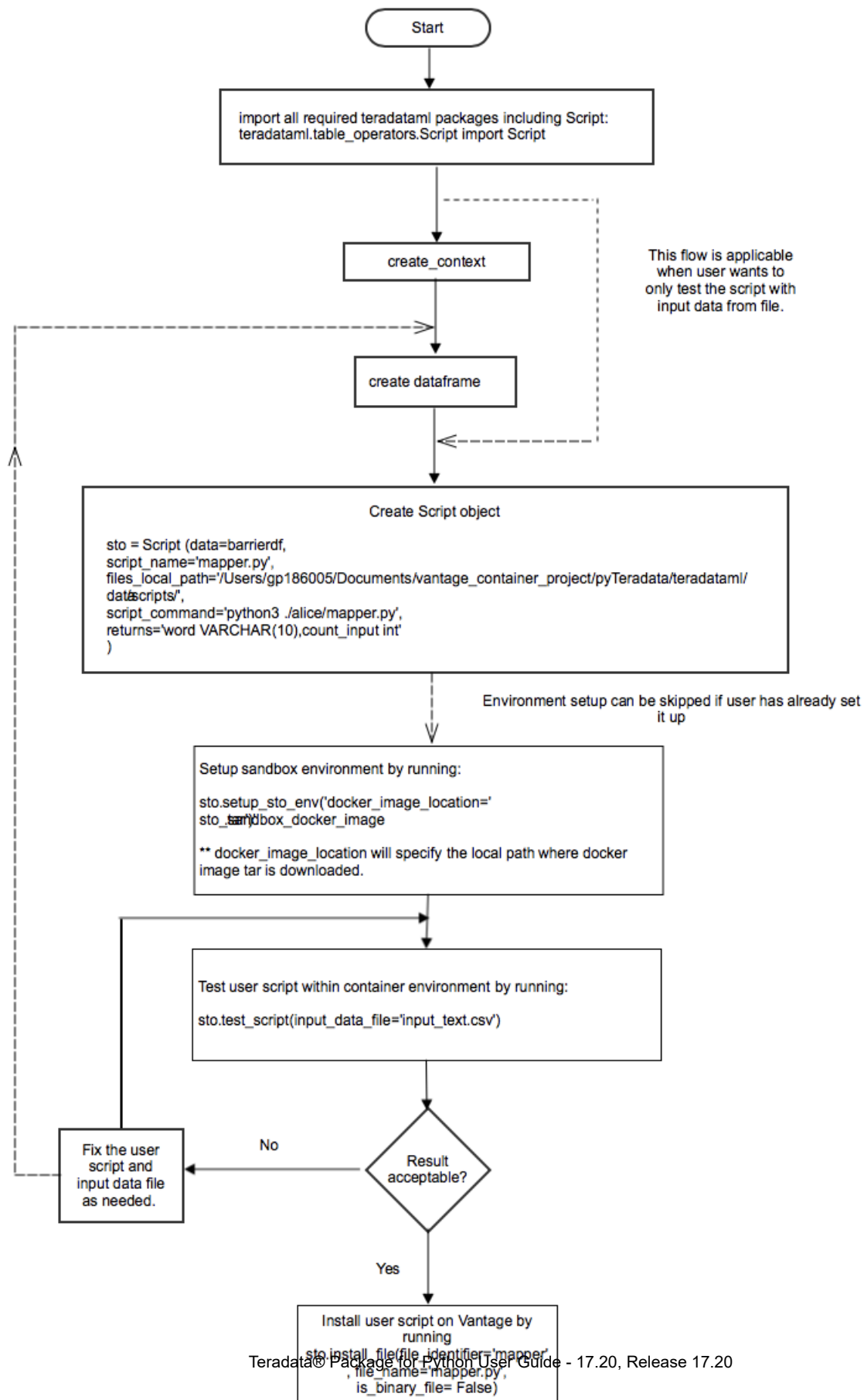
Note:

When configuring Docker on Windows 10, if users encounter the error Cannot enable Hyper-V service, run the following command from PowerShell window:

```
bcdedit /set hypervisorlaunchtype auto
```

Script() Workflow

The following is a typical workflow for Script().



The following example shows a workflow where user creates a Script object, sets up sandbox environment, tests Python script, installs it on Vantage, runs Python script on Vantage and then removes the script from Vantage.

Script() Workflow Example

The script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

To run this example, "mapper.py" and "barrier.csv" are required and must be present under the same location specified by the argument *files_local_path*.

- "barrier.csv" is present under <teradataml_install_location>/teradataml/data directory.
- "mapper.py" can be created as follows:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

1. Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

2. Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from terdatasqlalchemy import (VARCHAR)
```

3. Create teradataml DataFrame.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

4. Create a Script object.

```
>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
                  files_local_path= 'data/scripts',
                  script_command='python3 ./<database name>/mapper.py',
                  data_order_column="Id",
                  is_local_order=False
                  delimiter=',',
                  nulls_first=False,
                  sort_ascending=False,
```

```
        charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
        ("count_input", VARCHAR(2))])
    )
```

5. Setup the sandbox environment by providing local path to the Docker image file.

```
>>>
sto.setup_sto_env(docker_image_location='/tmp/sto_sandbox_docker_image.tar'))
Loading image from /tmp/sto_sandbox_docker_image.tar. It may take
few minutes.
Image loaded successfully.
Container c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
started successfully.
```

6. Run user script locally within Docker container using data from csv.

This helps user to fix script level issues outside Vantage.

```
>>> sto.test_script(input_data_file='../
barrier.csv', data_file_delimiter=',')
##### STDOUT Output #####

      word count_input
0         1           1
1       Old           1
2 Macdonald           1
3       Had           1
4         A           1
5       Farm           1
```

Script results look good.

7. Install the user script file on Vantage.

```
>>> sto.install_file(file_identifier='mapper',
file_name='mapper.py', is_binary=False)
```

8. Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

9. Run the user script on Vantage.

```
>>> sto.execute_script()
##### STDOUT Output #####

      word count_input
```

| | | |
|---|-----------|---|
| 0 | Macdonald | 1 |
| 1 | A | 1 |
| 2 | Farm | 1 |
| 3 | Had | 1 |
| 4 | Old | 1 |
| 5 | 1 | 1 |

Note:

The same script can run in SQL using SCRIPT table operator with the following SQL code:

```
SELECT * FROM Script(
  ON "barrier" AS "input"
  PARTITION BY ANY
  SCRIPT_COMMAND('python3 ./<database name>/mapper.py')
  delimiter(' ')
  returns('word VARCHAR(15), count_input VARCHAR(2)')
) as sqlmr
```

10. Remove the installed file from Vantage.

```
>>> sto.remove_file(file_identifier='mapper', force_remove=True)
```

Script Methods

As an interface to the SCRIPT table operator object in Analytics Database, Script offers execution in two modes:

- Test/Debug mode: to test user scripts locally in a containerized environment.

Supported methods:

- [setup_sto_env](#): set up test environment.
- [test_script](#): test user script in containerized environment.
- [set_data](#): set test data parameters.

- In-Database Script Execution mode: to execute user scripts in database.

Supported methods:

- [execute_script](#): execute user script in Vantage.
- [install_file](#): install or replace file in database.
- [remove_file](#): remove installed files from database.
- [set_data](#): set test data parameters.

Required Docker and OS Versions

setup_sto_env method and test_script method are supported on the following platforms with Docker 19.03.5 or later versions:

- Ubuntu 16.04.2 LTS or later versions
- CentOS Linux 7 or later versions
- RHEL 7.1 or later versions
- Windows 10 or later versions
- macOS 13 or later versions

All other methods: set_data, execute_script, install_file, remove_file, do not require Docker and work on all teradataml supported platforms.

Note:

setup_sto_env method and test_script method will not work on VMs that do not have docker setup. Users will encounter issues with docker setup on VMs that do not support nested virtualization. For example, "Failed to start the virtual machine *name of virtual machine* because one of the Hyper-V components is not running."

See <https://docs.docker.com/desktop/vm-vdi/> for details about this issue.

set_data

Use the set_data method to set data and data related arguments without having to re-create Script object.

Some of the set_data method arguments used in the examples:

- Required argument *data* specifies a teradataml DataFrame containing the input data for the script.
- Optional argument *is_local_order* specifies a boolean value to determine whether the input data is to be ordered locally or not.
- Optional argument *data_order_column* specifies the Order By column for *data*, and can be used no matter *is_local_order* is set to 'True' or 'False'.
- Optional argument *sort_ascending* specifies a boolean value to determine if the result set is to be sorted on the column specified in *data_order_column*, in ascending or descending order. This argument is ignored, if *data_order_column* is None.
- Optional argument *nulls_first* specifies a boolean value to determine whether NULLS are listed first or last during ordering.

For details of all arguments, see [Teradata Package for Python Function Reference](#).

Example Prerequisites

- To run the examples, "mapper.py" and "barrier.csv" are required and must be present under the same location specified by the argument *files_local_path*.

- "barrier.csv" is present under <teradataml_install_location>/teradataml/data directory.
- "mapper.py" can be created as follows:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

- Before running the examples, import required packages:

```
>>> from collections import OrderedDict
```

```
>>> from terdatasqlalchemy import (VARCHAR)
```

Example 1

This example uses [test_script](#) to test a Python script, then reset data and related arguments using `set_data` method and run Python script on Vantage with new parameters.

Note:

All data related arguments that are not specified in `set_data()` will be reset to default values.

- Create teradataml DataFrame.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

- Create a Script object without data and its arguments.

```
>>> sto = Script(
    script_name='mapper.py',
    files_local_path= 'data/scripts',
    script_command='python3 ./<database name>/mapper.py',
    charset='latin',
    returns=OrderedDict([("word", VARCHAR(15)),
    ("count_input", VARCHAR(2))])
)
```

- Test script using data from file.

```
>>> sto.test_script(input_data_file='../barrier.csv')
##### STDOUT Output #####
```

```

      word count_input
0  Macdonald      1
1      A          1
2      Farm       1
3      Had        1
4      Old        1
5      1          1

```

- Set data and related arguments to run and test the script on actual data on Vantage.

```

>>> sto.set_data(data='barrier',
data_order_column="Id",
is_local_order=False,
nulls_first=False,
sort_ascending=False
)

```

- Set the search path to the database where the file is installed.

```

>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")

```

- Run the user script on Vantage.

```

>>> sto.execute_script()
##### STDOUT Output #####

```

```

      word count_input
0  Macdonald      1
1      A          1
2      Farm       1
3      Had        1
4      Old        1
5      1          1

```

Example 2

In this example, script is tested using [test_script](#) and run on Vantage. User can reset data and related arguments and execute script on Vantage again.

Note:

All data related arguments that are not specified in `set_data()` will be reset to default values.

- Create a Script object that allows user to run script on Vantage.

```

>>> sto = Script(data=barrierdf,
script_name='mapper.py',

```

```

files_local_path= 'data/scripts',
script_command='python3 ./<database name>/mapper.py',
data_order_column="Id",
is_local_order=False
delimiter=',',
nulls_first=False,
sort_ascending=False,
charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
("count_input", VARCHAR(2))])
)

```

- Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

- Test the script using test_script running on Vantage.

```

>>> sto.execute_script()
##### STDOUT Output #####

```

| | word | count_input |
|---|-----------|-------------|
| 0 | Macdonald | 1 |
| 1 | A | 1 |
| 2 | Farm | 1 |
| 3 | Had | 1 |
| 4 | Old | 1 |
| 5 | 1 | 1 |

- Run the set_data to re-set data and some related parameter, to run the script with a different dataset.

- Create a new DataFrame.

```
>>> barrierdf_new = DataFrame.from_table("barrier_new")
```

- Set data and related arguments with different values.

```

>>> sto.set_data(data=barrierdf_new, data_order_column='Id',
is_local_order=True, nulls_first=True)

```

- Run the script again.

```

>>> sto.execute_script()
      word  count_input
0  Macdonald           1
1           A           1
2       Farm           1
3           2           1

```

| | | |
|---|------|---|
| 4 | his | 1 |
| 5 | farm | 1 |
| 6 | On | 1 |
| 7 | Had | 1 |
| 8 | Old | 1 |
| 9 | 1 | 1 |

Example 3

In this example, script is tested using [test_script](#) and run on Vantage. User run the script again with same dataset but different data related arguments by using `set_data()` to reset arguments.

Note:

All data related arguments that are not specified in `set_data()` will be reset to default values.

- Set related arguments with different values, but use same data.

```
>>> sto.set_data(data=barrierdf, data_order_column='Id',
is_local_order=True, nulls_first=True)
```

- Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

- Run the script again.

```
>>> sto.execute_script()
##### STDOUT Output #####

      word count_input
0  Macdonald         1
1           A         1
2         Farm         1
3          Had         1
4          Old         1
5           1         1
```

setup_sto_env

Use the `setup_sto_env` method to load already downloaded sandbox image. See [Script](#) for details about sandbox environment.

Note:

Teradata recommends using [setup_sandbox_env\(\)](#) function available from teradataml 17.00.00.02, instead of this setup_sto_env method from previous release.

The setup_sto_env method features only one argument: *docker_image_location*. This required argument specifies the location of the sandbox image on user's system.

Example

To run this example, "mapper.py" and "barrier.csv" are required and must be present under the same location specified by the argument *files_local_path*.

- "barrier.csv" is present under <teradataml_install_location>/teradataml/data directory.
- "mapper.py" can be created as follows:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

- Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

- Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from teradatasqlalchemy import (VARCHAR)
```

- Create teradataml DataFrame.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

- Create a Script object to run script on Vantage.

```
>>> sto = Script(data=barrierdf,
                 script_name='mapper.py',
                 files_local_path= 'data/scripts',
                 script_command='python3 ./<database name>/mapper.py',
                 data_order_column="Id",
                 is_local_order=False,
                 delimiter=',',
                 nulls_first=False,
```

```

        sort_ascending=False,
        charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
        ("count_input", VARCHAR(2))])
    )

```

- Run the script locally within Docker container using data from csv.

This helps users to fix script level issues outside Vantage.

```

>>> sto.setup_sto_env(docker_image_location='/tmp/
sto_sandbox_docker_image.tar'))
Loading image from /tmp/sto_sandbox_docker_image.tar. It may take few
minutes.
Image loaded successfully.
Starting a container for stosandbox:1.0 image.
Container c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
started successfully.

```

test_script

Use the `test_script` method to run script in docker container environment outside Vantage. Input data for user script is read from csv file or from table.

Note:

The purpose of `test_script()` function is to enable users to test their scripts for possible errors without installing it on Vantage.

- If called with input data file, `test_script()` function will simply read the data from the file and provide the data as input to the user script.
 - There is no data partitioning when input is from a file.
 - The `test_script()` function may produce different output if input is read from a file than input from database.
-

Note:

When executing a python script that consumes another file, for example, a binary model file `modelFile.out`, the script code typically needs to specify the filepath of the accompanying file.

- In in-database execution, the language script must include a filepath specification that includes the **SEARCHUIFDBPATH** before the supporting file. For example, `./myDatabase/modelFile.out`.
 - In sandbox execution, the filepath specification will not work. You need to modify the script to change the filepath specification to just `./modelFile.out`.
-

Required arguments:

- *input_data_file*: Specifies the absolute local path of input data file.

If set to None, data is read from AMP, else data is from the file passed in the argument *input_data_file*.

Optional arguments:

- *supporting_files*: Specifies a file or list of supporting files like model files to be copied to the container.
- *script_args*: Specifies command line arguments required by the user script.
- *exec_mode*: Specifies the mode in which user wants to test the script.

If set to 'sandbox', which is the default value, the user script will run within the sandbox environment.

If set to 'local', it will run locally on user's system.

Note:

When 'local' execution mode is used, Teradata recommends having the same Python packages and versions installed on local machine as those installed on the Vantage cluster. Python packages installed on Vantage and the corresponding version information can be found using the [db_python_package_details\(\)](#) function.

- ***kwargs*: Specifies the keyword arguments required for testing.

Possible keys:

- *data_row_limit*: Specifies the number of rows to be taken from all AMPs when reading from Vantage.
It is ignored when data is read from file.
- *password*: Specifies the password to connect to Vantage where the data resides.
It is required when reading from database.
- *data_file_delimiter*: Specifies the delimiter used in the input data file.
It can be specified when data is read from file.
- *data_file_header*: Specifies whether the input data file contains header.
It can be specified when data is read from file.
- *timeout*: Specifies the timeout for docker API calls when running in sandbox mode.
- *data_file_quote_char*: Specifies the quotechar used in the input data file.
It can be specified when data is read from file.
- *logmech*: Specifies the type of logon mechanism to establish a connection to Vantage. It can be specified only when data is read from AMP and execution mode is 'sandbox'.

Permitted values include:

- TD2: The Teradata 2 (TD2) mechanism provides authentication using a Vantage username and password. This is the default logon mechanism using which the connection is established to Vantage.

- TDNEGO: This is a security mechanism that automatically determines the actual mechanism required, based on policy, without user's involvement. The actual mechanism is determined by the TDGSS server configuration and by the security policy's mechanism restrictions.
- LDAP: This is a directory-based user logon to Vantage with a directory username and password and is authenticated by the directory.
- KRB5 (Kerberos): This is a directory-based user logon to Vantage with a domain username and password and is authenticated by Kerberos (KRB5 mechanism).

Note:

User must have a valid ticket-granting ticket in order to use this logon mechanism.

- JWT: The JSON Web Token (JWT) authentication mechanism enables single sign-on (SSO) to the Vantage after the user successfully authenticates to Teradata UDA User Service.

Note:

User must use *logdata* parameter when using 'JWT' as the logon mechanism.

Note:

teradataml expects the client environments are already set up with appropriate security mechanisms and are in working conditions. See *Teradata Vantage™ - Analytics Database Security Administration*, B035-1100 for more information.

- *logdata*: Specifies parameters to the LOGMECH command beyond those needed by the logon mechanism, such as user ID, password and tokens (in case of JWT) to successfully authenticate the user.

Note:

When data is read from file, even if these arguments are passed, they will be ignored.

For details of all arguments, see [Teradata Package for Python Function Reference](#).

Example 1: Test script in sandbox environment and run script on Vantage

This example shows a workflow where user creates a Script object, tests script on sandbox environment, installs Python script on Vantage, runs Python script on Vantage and then removes the script from Vantage.

The script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

By running user script locally within docker container and using data from csv, it helps the user to fix script level issues outside Vantage.

1. Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

2. Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from terdatasqlalchemy import (VARCHAR)
```

3. Create teradataml DataFrame.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

4. Create a Script object.

```
>>> sto = Script(data=barrierdf,
                 script_name='mapper.py',
                 files_local_path= 'data/scripts',
                 script_command='python3 ./<database name>/mapper.py',
                 data_order_column="Id",
                 is_local_order=False
                 delimiter=',',
                 nulls_first=False,
                 sort_ascending=False,
                 charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
                 ("count_input", VARCHAR(2))]))
```

5. Setup the sandbox environment by providing local path to the Docker image file.

```
>>> sto.setup_sto_env(docker_image_location='/tmp/
sto_sandbox_docker_image.tar'))
Loading image from /tmp/sto_sandbox_docker_image.tar. It may take few
minutes.
Image loaded successfully.
Starting a container for stosandbox:1.0 image.
Container c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
started successfully.
```

6. Run user script in sandbox mode with input from data file.

```
>>> sto.test_script(input_data_file='../barrier.csv',
...                 data_file_delimiter=',',
...                 data_file_quote_char='"',
...                 data_file_header=True,
...                 exec_mode='sandbox')
```

```
##### STDOUT Output #####
      word  count_input
0         1           1
1       Old           1
2 Macdonald           1
3       Had           1
4         A           1
5       Farm           1
```

Script results look good.

7. Install the user script file on Vantage.

```
>>> sto.install_file(file_identifier='mapper',
file_name='mapper.py', is_binary=False)
```

8. Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

9. Run the user script on Vantage.

```
>>> sto.execute_script()
##### STDOUT Output #####
      word  count_input
0 Macdonald           1
1         A           1
2       Farm           1
3       Had           1
4       Old           1
5         1           1
```

10. Remove the installed file from Vantage.

```
>>> sto.remove_file(file_identifier='mapper', force_remove=True)
```

Example 2: Test script in local mode with input from table

In this example, the script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

1. Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

2. Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from terdatasqlalchemy import (VARCHAR)
```

3. Create teradataml DataFrame objects.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

4. Create a Script object that allows user to execute script on Vantage.

```
>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
                  files_local_path= 'data/scripts',
                  script_command='python3 ./<database name>/mapper.py',
                  data_order_column="Id",
                  is_local_order=False,
                  delimiter=',',
                  nulls_first=False,
                  sort_ascending=False,
                  charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
                  ("count_input", VARCHAR(2))]))
```

5. Run user script in local mode with input from table.

```
>>> sto.test_script(data_row_limit=300,
                    password='<password>', exec_mode='local')
##### STDOUT Output #####
      word  count_input
0         1           1
1      Old           1
2 Macdonald           1
3      Had           1
4      A           1
5     Farm           1
```

Example 3: Test script in sandbox mode with different logon mechanisms

In this example, the script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

1. Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

2. Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from terdatasqlalchemy import (VARCHAR)
```

3. Create teradataml DataFrame objects.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

4. Create a Script object that allows user to execute script on Vantage.

```
>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
                  files_local_path= 'data/scripts',
                  script_command='python3 ./<database name>/mapper.py',
                  data_order_column="Id",
                  is_local_order=False
                  delimiter=',',
                  nulls_first=False,
                  sort_ascending=False,
                  charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
                  ("count_input", VARCHAR(2))]))
```

5. Run user script in sandbox mode with different logmech.

- Run user script in sandbox mode with logmech as 'TD2'.

```
>>> sto.test_script(script_args="4 5 10 6 480",
                    password="<password>", logmech="TD2")
```

- Run user script in sandbox mode with logmech as 'TDNEGO'.

```
>>> sto.test_script(script_args="4 5 10 6 480",
                    password="<password>", logmech="TDNEGO")
```

- Run user script in sandbox mode with logmech as 'LDAP'.

```
>>> sto.test_script(script_args="4 5 10 6 480",
                    password="<password>", logmech="LDAP")
```

- Run user script in sandbox mode with logmech as 'KRB5'.

```
>>> sto.test_script(script_args="4 5 10 6 480",
                    password="<password>", logmech="KRB5")
```

- Run user script in sandbox mode with logmech as 'JWT'.

```
>>> sto.test_script(script_args="4 5 10 6 480", password="<password>",
logmech='JWT', logdata='token=eyJpc...h8dA')
```

execute_script

Use the `execute_script` to run script on Vantage.

This method does not have any arguments.

Example

This example shows a workflow to create a Script object, install Python script on Vantage, runs it and then removes the script from Vantage.

The script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

To run this example, "mapper.py" and "barrier.csv" are required and must be present under the same location specified by the argument `files_local_path`.

- "barrier.csv" is present under <teradataml_install_location>/teradataml/data directory.
- "mapper.py" can be created as follows:

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

- Load example data.

```
>>> load_example_data("Script", ["barrier"])
```

- Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from teradatasqlalchemy import (VARCHAR)
```

- Create teradataml DataFrame.

```
>>> barrierdf = DataFrame.from_table("barrier")
```

- Create a Script object that allows user to run script on Vantage.

```
>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
                  files_local_path= 'data/scripts',
                  script_command='python3 ./<database name>/mapper.py',
                  data_order_column="Id",
                  is_local_order=False
                  delimiter=',',
                  nulls_first=False,
                  sort_ascending=False,
                  charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
                  ("count_input", VARCHAR(2))])
                  )
```

- Install script on Vantage.

```
>>> sto.install_file(file_identifier='mapper',
                     file_name='mapper.py', is_binary=False)
```

- Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

- Run the user script on Vantage.

```
>>> sto.execute_script()
##### STDOUT Output #####
```

| | word | count_input |
|---|-----------|-------------|
| 0 | Macdonald | 1 |
| 1 | A | 1 |
| 2 | Farm | 1 |
| 3 | Had | 1 |
| 4 | Old | 1 |
| 5 | 1 | 1 |

- Remove the installed file from Vantage.

```
>>> sto.remove_file(file_identifier='mapper', force_remove=True)
```

install_file

Use the `install_file` method to install or replace script on Vantage.

Required arguments:

- *file_identifier* specifies the name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database. The name should be unique within the database. It can be any valid Teradata identifier.
- *file_name* specifies the name of the file user wants to install.

Optional arguments:

- *is_binary* specifies if the file to be installed is a binary file.
- *force_replace* specifies if system should check for the file being used before replacing it:
 - If set to True, then the file is replaced even if it is being executed.
 - If set to False, then an error is thrown if it is being executed.

Note:

File can be on client or remote server. The file location should be specified accordingly.

Example 1: Install a file found at the relative path using default text mode

Note:

To run this example, "mapper.py" is required on client at the path specified by argument *file_local_path*.

- Create the "mapper.py" file.

```
#!/usr/bin/python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print ('%s\t%s' % (word, 1))
```

- Set the search path to the database where the file is installed.

```
>>> get_context().execute("SET SESSION SEARCHUIFDBPATH = <database name>;")
```

- Import required packages.

```
>>> from collections import OrderedDict
```

```
>>> from teradatasqlalchemy import (VARCHAR)
```

- Create a Script object that allows user to run script on Vantage.

```
>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
```



```

files_local_path= 'data/scripts',
script_command='python3 ./<database name>/mapper.py',
data_order_column="Id",
is_local_order=False
delimiter=',',
nulls_first=False,
sort_ascending=False,
charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
("count_input", VARCHAR(2))])
)

```

- Install the file on Vantage.

```

>>> sto.install_file(file_identifier='mapper',
file_name='mapper.py', is_binary=False)
File mapper.py installed in Vantage

```

- After making changes to the file "mapper.py", user can use the install_file method with "*replace=True*" to replace the existing file with the new file.

```

>>> sto.install_file(file_identifier='mapper', file_name='mapper.py',
is_binary=False, replace=True, force_replace=True)
File mapper.py replaced in Vantage

```

Example 2: Install a file found at the relative path using binary mode

```

>>> sto.install_file (file_identifier='binaryfile',
file_name='binary_file.dms', is_binary = True)
File binary_file.dms installed in Vantage

```

remove_file

Use the remove_file method to remove user installed files or scripts from Vantage.

Required arguments:

- *file_identifier* specifies the name associated with the user installed file. It cannot have a database name associated with it, as the file is always installed in the current database.
- *force_remove* specifies if system should check for the file being used before removing it.
 - If set to True, then the file is removed even if it is being executed.
 - If set to False, which is the default value, then an error is thrown if it is being executed.

Example 1: Install a file found at the relative path using default text mode and then remove the file

- Follow steps in [install_file](#) to install a file using default text mode.

- Remove the installed file.

```
>>> sto.remove_file (file_identifier='mapper', force_remove=True)
File mapper removed from Vantage
```

Example 2: Install a file found at the relative path using binary mode and then remove the file

- Follow steps in [install_file](#) to install a file using binary mode.
- Remove the installed file.

```
>>> sto.remove_file (file_identifier='binaryfile', force_remove=False)
File binaryfile removed from Vantage
```

Sandbox Container Utility

Sandbox container utility is a library of functions for setting up and cleaning up the sandbox environment and copying files from the container to localhost. It has the following functions:

- [setup_sandbox_env\(\)](#)
- [copy_files_from_container\(\)](#)
- [cleanup_sandbox_env\(\)](#)

setup_sandbox_env()

Use the `setup_sandbox_env()` function to load a sandbox image and start a new container, or start an existing container.

Note:

- Teradata recommends running this function once before running the Script method [test_script](#).
 - Sandbox image loading on user's system is skipped if the image is already loaded.
 - At any given point, only one container can be started from within teradataml.
 - At the end of a session, the image loaded and the container started by this function will be cleaned up.
 - Users also have the option to manually clean up the image and container using [cleanup_sandbox_env\(\)](#).
-

Note:

User should be careful when providing *sandbox_image_location* and *sandbox_image_name*. If an incorrect *sandbox_image_name* is provided that does not match the name and tag of the image to be loaded, then:

- The image will not be loaded. User should re-run the function with correct *sandbox_image_name*.
- If the user's system already has an image that the image name and tag are the same as the image specified in *sandbox_image_location*, then the new image will be loaded and it might replace the old image.

Optional arguments:

- To load a sandbox image and start a new container:
 - *sandbox_image_location* specifies the path to the sandbox image, including the image file name, on user's system.
 - *sandbox_image_name* specifies the name (tag) of the sandbox image
This name (tag) is used to start a container.

Note:

- *sandbox_image_location* and *sandbox_image_name* must be specified together.
- After loading the image, a container will be created and started.
- If the value provided to the argument *sandbox_image_name* does not match the name (tag) of the image to be loaded, the image will not be loaded. User needs to re-run this function with the correct *sandbox_image_name* value.

- To start an existing container:
container_id specifies the id of an existing Docker container.

Note:

This argument is required, if the arguments *sandbox_image_location* and *sandbox_image_name* are not specified.

- *timeout* specifies timeout value for docker API calls.
This argument is useful when loading large sandbox images. The default value is 5000. Increase this value if image loading fails because of timeout.

Example 1: Load a sandbox image and start a new container

This example loads an image from the location specified in the *sandbox_image_location* argument and starts a container. This is useful when the user wants to setup sandbox environment with a new image.

1. Import required module.

```
>>> from teradataml.table_operators.sandbox_container_util
import setup_sandbox_env
```

2. Load sandbox image and start a container.

```
>>> setup_sandbox_env(sandbox_image_location='/tmp/stosandbox.tar.gz',
                      sandbox_image_name='stosandbox:1.0')
    Loading image from /tmp/stosandbox.tar.gz. It may take few minutes.
    Image loaded successfully.
    Starting a container for stosandbox:1.0 image.
    Container c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
started successfully.
```

Example 2: Start an existing container

This example starts an existing container specified in the argument *container_id*. This is useful when the user wants to run user script in an existing container.

1. Import required module.

```
>>> from teradataml.table_operators.sandbox_container_util
import setup_sandbox_env
```

2. Load sandbox image and start a container.

```
>>> setup_sandbox_env(container_id='3fadb1e8bac3')
    Container 3fadb1e8bac3 started successfully.
```

copy_files_from_container()

Use the `copy_files_from_container()` function to copy files from a container to local file system.

Files are copied to a directory under the `.teradataml` directory in user's home directory. User is responsible for cleaning up this directory.

The container can be specified by the argument *container_id* or indicated by `figure.sandbox_container_id`.

See [sandbox_container_id](#) for details about this teradataml configuration option.

Required argument:

- *files_to_copy* specifies the names of the files to be copied from a container.

Note:

The files to be copied should be in the `/home/tdatuser` directory inside the container.

Optional argument:

- `container_id` specifies the container from which files are to be copied.
If this argument is not specified, then files are copied from the container indicated in `configure.sandbox_container_id`.

Example 1: Copy a single file

This example copies a single file from `/home/tdatuser` inside a container started by `teradataml` to local host.

1. Import required module.

```
>>> from teradataml.table_operators.sandbox_container_util
import copy_files_from_container
```

2. Copy a file from the container indicated in `configure.sandbox_container_id`.

```
>>> copy_files_from_container(files_to_copy="file1.txt")
Files copied to: /<user's home
directory>/./teradataml/files_from_container_210110_19394
```

Example 2: Copy multiple files

This example copies multiple files from `/home/tdatuser` inside a container started by `teradataml` to local host.

1. Import required module.

```
>>> from teradataml.table_operators.sandbox_container_util
import copy_files_from_container
```

2. Copy a file from the container indicated in `configure.sandbox_container_id`.

```
>>> copy_files_from_container(files_to_copy=["output/
file1.txt", "file2.txt"])
Files copied to: /<user's home
directory>/./teradataml/files_from_container_210110_19394
```

cleanup_sandbox_env()

Use the `cleanup_sandbox_env()` function to clean up sandbox environment set up by `teradataml` using the `setup_sandbox_env()` function. This function removes the image and container used to set up the sandbox environment.

Example 1: Clean up sandbox environment set up by the setup_sandbox_env() function

In this example, you first set up a sandbox environment using the `setup_sandbox_env()` function and then clean it up using this `cleanup_sandbox_env()` function.

1. Import the required modules.

```
>>> from teradataml.table_operators.sandbox_container_util import *
```

2. Load a sandbox image and start a container, to set up a sandbox environment.

```
>>> setup_sandbox_env(sandbox_image_location='/tmp/stosandbox.tar.gz',
                      sandbox_image_name='stosandbox:1.0')
Loading image from /tmp/stosandbox.tar.gz. It may take few minutes.
Image loaded successfully.
Starting a container for stosandbox:1.0 image.
Container
c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
started successfully.
```

3. Clean up the sandbox environment created in step 2.

```
>>> cleanup_sandbox_env()
Removed
container: c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345
Removed image: stosandbox:1.0
```

Example 2: Clean up sandbox environment set up by the setup_sandbox_env() function, not affecting other containers

In this example, you first set `configure.sandbox_container_id` manually, then run `setup_sandbox_env()` and clean it up using `cleanup_sandbox_env()` function. This only cleans up image and container created by `setup_sandbox_env()` and does not affect the sandbox that is manually set in `configure.sandbox_container_id`.

1. Load the required modules.

```
>>> from teradataml.table_operators.sandbox_container_util import *
```

2. Set `configure.sandbox_container_id` manually to use an existing container.

```
>>> configure.sandbox_container_id = '<container_id_set_by_user>'
```

3. Load a sandbox image and start a container, to set up a new sandbox environment.

```
>>> setup_sandbox_env(sandbox_image_location='/tmp/stosandbox.tar.gz',
                      sandbox_image_name='stosandbox:1.0')
Loading image from /tmp/stosandbox.tar.gz. It may take few minutes.
```

```
Image loaded successfully.  
Starting a container for stosandbox:1.0 image.  
Container  
c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345  
started successfully.
```

4. Clean up the sandbox environment created by the `setup_sandbox_env()` function in step 3.

```
>>> cleanup_sandbox_env()  
Removed  
container: c1dd4d4b722cc54b643ab2bdc57540a3a3e6db98c299defc672227de97d2c345  
Removed image: stosandbox:1.0
```

Note:

The `cleanup_sandbox_env()` function won't affect the container in step 2.

teradataml Options

Configuration Options

sandbox_container_id

The **sandbox_container_id** configuration property indicates the Docker container that will be used by the `test_script()` method of the `Script` class. This property is automatically set to the container id of the container started when running [setup_sandbox_env\(\)](#).

User script is run on the container indicated by this property in sandbox mode. The container is cleaned up by Garbage Collector at the end of a session and this property is set back to `None`.

- By default, it is set to `None`, which means a sandbox container is not started from within `teradataml`.
- When a sandbox container is started from within `teradataml`, this property is set to the id of that sandbox container.
- If a user wants to use any other container, this property needs to be manually set to that container id using `configure.sandbox_container_id = <container id>`.

Note:

In this case, `teradataml` is not responsible for container cleanup at the end of a session.

Example

1. Import the required modules.

```
>>> from teradataml.table_operators.sandbox_container_util
import setup_sandbox_env
>>> from teradataml.table_operators.Script import Script
>>> from collections import OrderedDict
>>> from teradatasqlalchemy import (VARCHAR)
```

2. Check the configuration property value for `teradataml`.

```
>>> print("configure.sandbox_container_id is set
to: {}".format(configure.sandbox_container_id))
configure.sandbox_container_id is set to: None
```

3. Load sandbox image and start a container.

```
>>> setup_sandbox_env(sandbox_image_location="/tmp/
sto_sandbox_image.tar.gz", sandbox_image_name="stosandbox:1.0")
```



```

Loading image from /tmp/sto_sandbox_image.tar.gz. It may take few minutes.
Image loaded successfully.
Container abcd8211241e801ecb584e9382f2e581312f805fddcf8507f9c8731dfbffa38
started successfully.

```

4. Check the configuration property value for teradataml, after a container is started.

```

>>> print("configure.sandbox_container_id is set
to: {}".format(configure.sandbox_container_id))
configure.sandbox_container_id is set
to: abcd8211241e801ecb584e9382f2e581312f805fddcf8507f9c8731dfbffa38

```

5. Load example data.

```

>>> load_example_data("Script", ["barrier"])

```

6. Create teradataml DataFrame objects.

```

>>> barrierdf = DataFrame.from_table("barrier")

```

7. Create a Script object to run a script on Vantage.

The script "mapper.py" reads in a line of text input ("Old Macdonald Had A Farm") from csv and splits the line into individual words, emitting a new row for each word.

```

>>> sto = Script(data=barrierdf,
                  script_name='mapper.py',
                  files_local_path= 'data/scripts',
                  script_command='python3 ./alice/mapper.py',
                  data_order_column="Id",
                  is_local_order=False,
                  delimiter=',',
                  nulls_first=False,
                  sort_ascending=False,
                  charset='latin', returns=OrderedDict([("word", VARCHAR(15)),
                  ("count_input", VARCHAR(2))])
                )

```

8. Run test_script() on the container created in step 3.

```

>>> sto.test_script(input_data_file='../barrier.csv')
##### STDOUT Output #####

      word count_input
0  Macdonald          1
1           A          1

```

| | | |
|---|------|---|
| 2 | Farm | 1 |
| 3 | Had | 1 |
| 4 | Old | 1 |
| 5 | 1 | 1 |

Note:

If the user has started a container outside teradataml with container_id '3a938ac820be' and wants to run script on that container:

- a. Set configure.sandbox_container_id to the new container id.

```
>>> configure.sandbox_container_id = '3a938ac820be'
```

- b. Check the configuration property value for teradataml.

```
>>> print("configure.sandbox_container_id is set
to: {}".format(configure.sandbox_container_id))
configure.sandbox_container_id is set to: 3a938ac820be
```

- c. Run test_script() on container with id '3a938ac820be'.

```
>>> sto.test_script(input_data_file='../barrier.csv')
##### STDOUT Output #####
```

```

      word count_input
0  Macdonald         1
1         A          1
2       Farm         1
3       Had          1
4       Old          1
5         1          1
```

9. Exit the session.

```
>>> remove_context()
```

Note:

This cleans up the container created in step 3.

10. Check the configuration property value for teradataml again, after the container is cleaned up.

```
>>> print("configure.sandbox_container_id is set
to: {}".format(configure.sandbox_container_id))
configure.sandbox_container_id is set to: None
```

temp_table_database and temp_view_database

teradataml internally creates database objects (tables and views) as required while executing several functions. The **temp_table_database** and **temp_view_database** configurable options allow users to specify database names where tables and views are created, respectively. These two options give users flexibility to specify database names other than the default database to create views and tables.

The [create_context](#) function provides *temp_database_name* and *database* arguments that also allow users to specify the database name, where database objects can be created. If none of them are specified then the objects are created in users' default database.

While using these options,

- If the configuration option **temp_table_database** is None, then tables are created in the database specified at the time of context creation.
- If the configuration option **temp_view_database** is None, then views are created in the database specified at the time of context creation.
- Teradata recommends using these options when following conditions are true, otherwise, use the arguments in the `create_context()` function:
 - When a user has permissions to create tables in one database while views in other database.
 - In case a user wishes to create these temporary objects in a database different than the one specified at the time of context creation. This allows the user to change the database name without re-creating the context.
 - In case a user forgets to use the `create_context()` arguments to set the *temp_database_name*, then use these options.

Example

```
>>> from teradataml import *

>>> # Set the teradataml Configuration property 'temp_table_database' True
>>> configure.temp_table_database = "tempdb"

>>> load_example_data("dataframe", "admissions_train")

>>> # Load example loads sample tables to "tempdb" database as option is set.
>>> admissions_train
= DataFrame.from_table(in_schema("tempdb","admissions_train"))

>>> glm_out1 = GLM(formula = "admitted ~ stats + masters + gpa + programming",
                    family = "LOGISTIC",
                    linkfunction = "LOGIT",
                    data = admissions_train,
                    weights = "1",
```

```

        threshold = 0.01,
        maxit = 25,
        step = False,
        intercept = True
    )

```

```

>>> # If we observe query, we can find that output table is created in "tempdb"
>>> glm_out1.output.show_query()
select * from "tempdb"."teradataml_td_glm_mle0_1623374476821412";

```

byom_install_location

The **byom_install_location** configuration option indicates the name of the database where Bring Your Own Model functions are installed. This option is internally used by BYOM functions like `H2OPredict()` and `PMMLPredict()`.

Example

- Import all modules.

```
>>> from teradataml import *
```

- Set the configuration option **byom_install_location** to "mldb".

```

>>> # Set the teradataml Configuration option 'byom_install_location'
to "mldb"
>>> configure.byom_install_location = "mldb"

```

- Load example data.

```
>>> load_example_data("byom", "iris_test")
```

- Load model files.

```

>>> # Load model file into Vantage.
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
"models", "iris_mojo_glm_h2o_model")

```

- Save the model.

```
>>> save_byom("iris_mojo_glm_h2o_model", model_file, "byom_models")
```

- Retrieve the saved model and directly pass the output as an input to the `H2OPredict` function.

```

>>> modeldata =
retrieve_byom("iris_mojo_glm_h2o_model", table_name="byom_models")

```

```
>>> result = H2OPredict(newdata=iris_test,
                        newdata_partition_column='id',
                        newdata_order_column='id',
                        modeldata=modeldata,
                        modeldata_order_column='model_id',
                        model_output_fields=['label', 'classProbabilities'],
                        accumulate=['id', 'sepal_length', 'petal_length'],
                        overwrite_cached_models='*',
                        enable_options='stageProbabilities',
                        model_type='OpenSource'
                        )
```

- Print the results of the H2OPredict function.

```
>>> # Print the results.
>>> print(result.result)
```

val_install_location

The **val_install_location** configuration option indicates the name of the database where Vantage Analytic Library functions are installed. This option is internally used by VAL functions.

Example

- Import all modules.

```
>>> from teradataml import *
```

- Set the configuration option **val_install_location** to "SYSLIB".

```
>>> # Set the teradataml configuration option 'val_install_location'
to "SYSLIB"
>>> configure.val_install_location = "SYSLIB"
```

- Create a data frame.

```
>>> df = DataFrame("credit_tran")
```

- Perform Association analysis using default values.

```
>>> # Perform Association analysis using default values.
>>> obj = valib.Association(data=df,
group_column="cust_id", item_column="channel")
```

- Print the affinity result.

```
>>> # Print the affinity result. Only affinity result for default
combination 11 is produced.
>>> print(obj.result_11)
```

database_version

The **database_version** configuration option specifies the actual database version of the system teradataml is connected to.

Example

```
>>> teradataml.options.configure.database_version = "17.05a.00.147"
```

read_nos_function_mapping

The **read_nos_function_mapping** configuration option specifies the function mapping name for the READ_NOS table operator function.

Example

```
>>> teradataml.options.configure.read_nos_function_mapping = "read_nos_fm"
```

write_nos_function_mapping

The **write_nos_function_mapping** configuration option specifies the function mapping name for the WRITE_NOS table operator function.

Example

```
>>> teradataml.options.configure.write_nos_function_mapping = "write_nos_fm"
```

auth_token

The **auth_token** specifies the authentication token to connect to VantageCloud Lake.

Example

```
>>> teradataml.options.configure.auth_token='JWT_TOKEN'
```

ues_url

The **ues_url** specifies the URL for User Environment Service (UES) in VantageCloud Lake.

Example

```
>>> teradataml.options.configure.ues_url='http://vantagelake.cloud/account_id'
```

certificate_file

The **certificate_file** specifies the path of the certificate file, which is used in encrypted REST service calls.

Example

```
>>> teradataml.options.configure.certificate_file='certificate.crt'
```

Display Options

You can control the display of a teradataml DataFrame by setting options in the display module.

```
>>> from teradataml.options.display import display
```

Use the help command to show the built-in document about the display module.

```
>>> help(display)
```

byte_encoding

The **byte_encoding** option controls the encoding used for printing byte-like types in a DataFrame. The default is Base16. The following encodings are supported:

- BaseX
X is a power of 2 (for example, 2, 8, 16).
- BaseY
Y is not a power of 2 (for example, 10 and 36).
- Base64M (MIME)
- ASCII

Note:

The `from_bytes` function is used to display byte-like types. See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates* for more information on specifying arguments for `FROM_BYTES`.

Example: Default is Base16

```
>>> display.byte_encoding
'base16'
```

```
>>> DataFrame('byte')
```

| | byte | varbyte | blob |
|----|-------------|-------------|-------------|
| id | | | |
| 2 | b'32616263' | b'32616263' | b'32616263' |
| 1 | b'31616263' | b'31616263' | b'31616263' |
| 0 | b'30616263' | b'30616263' | b'30616263' |

Example: Set the value to ascii

```
>>> display.byte_encoding = 'ascii'
>>> DataFrame('byte')
```

| | byte | varbyte | blob |
|----|---------|---------|---------|
| id | | | |
| 2 | b'2abc' | b'2abc' | b'2abc' |
| 1 | b'1abc' | b'1abc' | b'1abc' |
| 0 | b'0abc' | b'0abc' | b'0abc' |

max_rows

The **max_rows** option controls the default value for the `head()` method of the `DataFrame` and the number of rows printed from a `DataFrame`. The default value is 10.

Example: Set the value to 5

```
>>> display.max_rows = 5
>>> DataFrame('iris')
```

| primary_index | SepalLength | SepalWidth | PetalLength | PetalWidth | Name |
|---------------|-------------|------------|-------------|------------|-----------------|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 40 | 5.0 | 3.5 | 1.3 | 0.3 | Iris-setosa |
| 80 | 5.5 | 2.4 | 3.8 | 1.1 | Iris-versicolor |
| 101 | 5.8 | 2.7 | 5.1 | 1.9 | Iris-virginica |
| 141 | 6.9 | 3.1 | 5.1 | 2.3 | Iris-virginica |

precision

The **precision** option controls the number of places to round a numeric column when printing the `DataFrame`. The default value is 3.

This option only affects data when printed. When retrieving actual values from the DataFrame (such as with `to_pandas()`), the numeric column is returned with precision as specified in the schema.

Note:

The **precision** setting does not affect columns of type float.

Example: Default value is 3

```
>>> display.precision
3

>>> DataFrame('numeric')
```

| | integer | smallint | bigint | decimal | float | number | number2 | number3 | byteint |
|----|---------|----------|--------|---------|----------|--------|---------|---------|---------|
| id | | | | | | | | | |
| 2 | 2 | 3 | 4 | 4.860 | 5.313308 | 5.737 | .46 | .687 | 2 |
| 1 | 1 | 2 | 2 | 2.430 | 2.656654 | 2.869 | .23 | .344 | 1 |
| 3 | 3 | 4 | 8 | 7.290 | 7.969962 | 8.606 | .69 | 1.031 | 3 |

Example: Set the value to 1

```
>>> display.precision = 1
>>> DataFrame('tips')
```

| | integer | smallint | bigint | decimal | float | number | number2 | number3 | byteint |
|----|---------|----------|--------|---------|----------|--------|---------|---------|---------|
| id | | | | | | | | | |
| 2 | 2 | 3 | 4 | 4.9 | 5.313308 | 5.7 | .5 | .7 | 2 |
| 1 | 1 | 2 | 2 | 2.4 | 2.656654 | 2.9 | .2 | .3 | 1 |
| 3 | 3 | 4 | 8 | 7.3 | 7.969962 | 8.6 | .7 | 1 | 3 |

print_sqlmr_query

The **print_sql_mr_query** option is a boolean that controls whether or not to print the query being sent to when an analytic function is called. The default is False.

Example: Set the value to True

```
>>> display.print_sqlmr_query = True
>>> from teradataml.analytics.KMeans import KMeans
```

```
>>> computers_train1 = DataFrame.from_table("computers_train1")
>>> kmeanssample_centroid = DataFrame.from_table("kmeanssample_centroid")
>>> result = KMeans(data=computers_train1,
                    centroids_table=kmeanssample_centroid,
                    centers=8,
                    threshold=0.0395,
                    iter_max=10,
                    unpack_columns=False,
                    seed=10,
                    data_sequence_column='primary_index'
                    )
```

```
SELECT * FROM KMeans(
  ON "computers_train1" AS InputTable
  ON "kmeanssample_centroid" AS CentroidsTable
  OUT TABLE OutputTable(TDML.ml__td_kmeans0_1538706050004791)
  OUT TABLE ClusteredOutput(TDML.ml__td_kmeans1_1538702918024205)
  USING
  NumClusters('8')
  MaxIterNum('10')
  Seed('10')
  SequenceInputBy('InputTable:primary_index')
) as sqlmr
```

blob_length

The **blob_length** option specifies the default length of BLOB column to display in a teradataml DataFrame. You can set this option to None to display the complete BLOB data. Default value is 10.

Use the default value

```
>>> display.blob_length
10
```

```
>>> df = DataFrame("byom_models")
>>> df
                                model
model_id
model17  b'504B03041400080808...'
```

Set the length to 20

```
>>> display.blob_length=20

>>> df
                                     model
model_id
model17  b'504B0304140008080800B36DC252000000000...'
```

suppress_vantage_runtime_warnings

The **suppress_vantage_runtime_warnings** option specifies whether to display the warnings raised by Vantage or not.

When set to True, warnings raised by Vantage are not displayed. Otherwise, warnings are displayed. Default value is False.

Note:

This option may also suppress valid warnings. So, Teradata does not recommend using this option unless it is really necessary.

Example 1: Suppress warnings

```
>>> display.suppress_vantage_runtime_warnings = True
```

Example: Compute the average values in a series, using the 'C' moving average type.

```
>>> obj1 = MovingAverage(data=titanic_data,
                          data_partition_column='fare',
                          data_order_column='passenger',
                          include_first=False,
                          alpha=0.1,
                          start_rows=2,
                          window_size=3,
                          mavgtype='C')
```

Example 2: Enable runtime warnings

```
>>> display.suppress_vantage_runtime_warnings = False
```

Example: Compute the average values in a series, using the 'C' moving average type.

```
>>> obj2 = MovingAverage(data=titanic_data,
                          data_partition_column='fare',
```

```

        data_order_column='passenger',
        include_first=False,
        alpha=0.1,
        start_rows=2,
        window_size=3,
        mavgtype='C')
VantageRuntimeWarning: [Teradata][teradataml](TDML_2086) Following warning
raised from Vantage with warning code: 4862
[Teradata Database] [Warning 4862] Query executed on Primary Cluster as the
request is multi-statement request
warnings.warn(msg_.format(warnRes[5], warnRes[6]), VantageRuntimeWarning)

```

set_config_params

Use the `set_config_params` function to set the configurations in Vantage.

Alternatively, you can set the configuration parameters independently using the `teradataml.configure` module.

Optional arguments "kwargs" specifies keyword arguments:

- *auth_token*: Specifies the authentication token to connect to VantageCloud Lake.

Note:

Authentication token will expire after a specific time. You can get the new authentication token and set it again.

- *ues_url*: Specifies the URL for User Environment Service in VantageCloud Lake.
- *certificate_file*: Specifies the path of the certificate file, which is used in encrypted REST service calls.
- *default_varchar_size*: Specifies the size of varchar datatype in Vantage.

The default size is 1024.

- *vantage_version*: Specifies the version of the Vantage system teradataml is connected to.
- *val_install_location*: Specifies the database name where Vantage Analytic Library (VAL) functions are installed.
- *byom_install_location*: Specifies the database name where Bring Your Own Model (BYOM) functions are installed.
- *sandbox_container_id*: Specifies the id of the sandbox container that will be used by the `Script.test_script()` method.
- *database_version*: Specifies the database version of the system teradataml is connected to.
- *read_nos_function_mapping*: Specifies the function mapping name for the READ_NOS table operator function.
- *write_nos_function_mapping*: Specifies the function mapping name for the WRITE_NOS table operator function.

Example 1: Set configuration parameters using set_config_params() function

```
>>> from teradataml import set_config_params

>>> set_config_params(auth_token="abc-pqr-123",

                        ues_url="https://teracloud/v1/accounts/xyz-234-76085/open-
analytics",
                        certificate_file="cert.crt",
                        default_varchar_size=512,
                        val_install_location="VAL_USER",
                        sandbox_container_id="bgf1233csdh123",
                        read_nos_function_mapping="read_nos_fm",
                        write_nos_function_mapping="write_nos_fm")

True
```

Example 2: Set configuration parameters using configure module

```
>>> from teradataml import configure

>>> configure.auth_token="abc-pqr-123"
>>> configure.ues_url="https://teracloud/v1/accounts/xyz-234-76085/open-
analytics"

>>> configure.certificate_file="cert.crt"

>>> configure.default_varchar_size=512

>>> configure.val_install_location="VAL_USER"

>>> configure.sandbox_container_id="bgf1233csdh123"

>>> configure.read_nos_function_mapping="read_nos_fm"

>>> configure.write_nos_function_mapping="write_nos_fm"
```

Series from teradataml DataFrame

The Series object can be created from a DataFrame using the [squeeze\(\) Method](#).

Series is a one-dimensional labeled representation of a single column DataFrame.

Example

```
>>> df = DataFrame("admissions_train")
>>> gpa = df.select(["gpa"])
>>> gpa.squeeze()
0    4.00
1    2.33
2    3.46
3    3.83
4    4.00
5    2.65
6    3.57
7    3.44
8    3.85
9    3.95
Name: gpa, dtype: float64
```

Series Manipulation

head()

Use the head() function to print the first n rows of a teradataml Series.

The function takes the optional argument n , and prints the first n number of rows. The default value for n is 10.

Note:

The Series object is sorted on the column. The column type must support sorting.

Unsupported types include: 'BLOB', 'CLOB', 'ARRAY', 'VARRAY'.

Example: Print default number of rows

This example prints the default 10 rows of Series "gpa":

```
>>> df = DataFrame("admissions_train")
>>> gpa = df.select(["gpa"]).squeeze()
```

```
>>> gpa.head()
0    4.00
1    2.33
2    3.46
3    3.83
4    4.00
5    2.65
6    3.57
7    3.44
8    3.85
9    3.95
Name: gpa, dtype: float64
```

Examples: Print n rows

The following example prints 5 rows of "gpa":

```
>>> gpa.head(5)
0    2.33
1    3.00
2    2.65
3    1.98
4    1.87
Name: gpa, dtype: float64
```

The following example prints 15 rows of "gpa":

```
>>> gpa.head(15)
0    2.33
1    3.00
2    3.13
3    3.44
4    3.46
5    3.46
6    3.50
7    3.50
8    3.50
9    3.52
```

```

10    3.55
11    3.45
12    2.65
13    1.98
14    1.87
Name: gpa, dtype: float64

```

unique()

Use the `unique()` function to return a new Series object with only the the unique or distinct values in the Series.

```

>>> df = DataFrame.from_query('select admitted from admissions_train')
>>> s = df.squeeze(axis = 1)
>>> s
0    1
1    1
2    0
3    0
4    1
5    0
6    1
7    0
8    0
9    1
Name: admitted, dtype: object

```

```

>>> s.unique()
0    0
1    1
Name: admitted, dtype: object

```

Series Metadata

name

Use the `name` property to get the name or the label of a Series.

```

>>> df = DataFrame("admissions_train")
>>> gpa = df.select(["gpa"]).squeeze()
>>> gpa
0    4.00

```



```
1    2.33
2    3.46
3    3.83
4    4.00
5    2.65
6    3.57
7    3.44
8    3.85
9    3.95
Name: gpa, dtype: float64
```

```
>>> gpa.name
'gpa'
```

The Bring Your Own Model feature in Vantage provides flexibility to score data in Vantage using external models.

To score data using external models, the model files should be stored in Vantage. Users can store the models in any table in Vantage. The table should have at least the following two columns:

| Column Name | Column Type |
|-------------|------------------------|
| model_id | VARCHAR(of any length) |
| model | BLOB |

Note:

- These functions are available based on Vantage version.
 - You should import these functions only after establishing the connection to Vantage.
These functions may not work properly if these are imported before context creation.
 - You can check whether these functions are available or not by running the function `display_analytic_functions()`.
 - If these functions are available for the corresponding Vantage version, then `display_analytic_functions()` displays those functions.
 - Otherwise, these functions are not displayed in `display_analytic_functions()` output.
-

teradataml offers APIs that enable end users to process BYOM data:

- Save a BYOM model in Vantage, using [save_byom\(\)](#);
- List all available BYOM models from Vantage, using [list_byom\(\)](#);
- Retrieve a specific model from Vantage, using [retrieve_byom\(\)](#);
- Delete a BYOM model from Vantage, using [delete_byom\(\)](#);
- Set the BYOM model catalog information, using [set_byom_catalog\(\)](#);
- Set the License information, using [set_license\(\)](#);
- Get the License information, using [get_license\(\)](#).

Supported External Model Types

The BYOM feature in Vantage supports the following types of external models:

- H2O
- PMML
- ONNX

PMMLPredict

PMML is the most popular standard serialization format for exchange of Machine Learning models. Most customers train their models in tools external to Vantage, such as Scikit-learn. Vantage Analytics enables customers to bring their models to Vantage (by inserting the model as a blob into a table) and apply them to data stored in Analytics Database for scoring. Users can use these external models for scoring through teradataml by using PMMLPredict() function.

The following are examples of PMMLPredict() function call.

Example Setup

- Import necessary modules.

```
>>> import os, teradataml
```

```
>>> from teradataml.options.configure import configure
```

```
>>> from teradataml import DataFrame, load_example_data,
save_byom, retrieve_byom
```

- Load example data.

```
>>> load_example_data("byom", "iris_input")
```

- Create teradataml DataFrame object.

```
>>> iris_test = DataFrame("iris_input")
```

- Set install location of the BYOM functions.

```
>>> configure.byom_install_location = "mldb"
```

Example 1: Run a query with GLM model and overwrite cached models

- Load model file into Vantage.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
"models", "iris_db_glm_model.pmml")
```

- Save the model.

```
>>> save_byom("iris_db_glm_model", model_file, "byom_models")
```

- Retrieve the model.

```
>>> modeldata = retrieve_byom("iris_db_glm_model", table_name="byom_models")
```

- Pass the output of the retrieve_model API as an input to the PMMLPredict function to score data.

```
>>> result = PMMLPredict(modeldata = modeldata,
                          newdata = iris_test,
                          accumulate = ['id',
                                        'sepal_length', 'petal_length'],
                          overwrite_cached_models = '*')
```

Example 2: Run a query with XGBoost model and overwrite cached models

- Load model file into Vantage.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                              "models", "iris_db_xgb_model.pmml")
```

- Save the model.

```
>>> save_byom("iris_db_xgb_model", model_file, "byom_models")
```

- Retrieve the model.

```
>>> modeldata = retrieve_byom("iris_db_xgb_model", table_name="byom_models")
```

- Pass the output of the retrieve_model API as an input to the PMMLPredict function to score data.

```
>>> result = PMMLPredict(modeldata = modeldata,
                          newdata = iris_test,
                          accumulate = ['id',
                                        'sepal_length', 'petal_length'],
                          overwrite_cached_models = '*')
```

H2OPredict

H2OPredict performs a prediction on each row of the input table using a model previously trained in H2O and then loaded into the database. The model uses an interchange format called MOJO and it is loaded as a blob to a table in the database by the user.

The following are examples of H2OPredict() function call.

Example Setup

- Import necessary modules.

```
>>> import os, teradataml
```

```
>>> from teradataml.options.configure import configure
```

```
>>> from teradataml import DataFrame, load_example_data,
save_byom, retrieve_byom
```

- Load example data.

```
>>> load_example_data("byom", "iris_input")
```

- Create teradataml DataFrame object.

```
>>> iris_test = DataFrame("iris_input")
```

- Set install location of the BYOM functions.

```
>>> configure.byom_install_location = "mldb"
```

Example 1: Run a query with GLM model and overwrite cached models

The query also includes arguments *model_type*, *enable_options* and *model_output_fields*.

- Load model file into Vantage.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
"models", "iris_mojo_glm_h2o_model")
```

- Save the model.

```
>>> save_byom("iris_mojo_glm_h2o_model", model_file, "byom_models")
```

- Retrieve the model.

```
>>> modeldata =
retrieve_byom("iris_mojo_glm_h2o_model", table_name="byom_models")
```

- Pass the output of the retrieve_model API as an input to the PMMLPredict function to score data.

```
>>> result = H2OPredict(newdata=iris_test,
                        newdata_partition_column='id',
                        newdata_order_column='id',
                        modeldata=modeldata,
                        modeldata_order_column='model_id',
                        model_output_fields=['label', 'classProbabilities'],
                        accumulate=['id', 'sepal_length', 'petal_length'],
                        overwrite_cached_models='*',
                        enable_options='stageProbabilities',
                        model_type='OpenSource')
```

- Print the results.

```
print(result.result)
```

Example 2: Run a query with XGBoost model and overwrite cached models

The query also includes arguments *model_type*, *enable_options* and *model_output_fields*.

- Load model file into Vantage.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                                "models", "iris_mojo_xgb_h2o_model")
```

- Save the model.

```
>>> save_byom("iris_mojo_xgb_h2o_model", model_file, "byom_models")
```

- Retrieve the model.

```
>>> modeldata =
retrieve_byom("iris_mojo_xgb_h2o_model", table_name="byom_models")
```

- Pass the output of the retrieve_model API as an input to the PMMLPredict function to score data.

```
>>> result = H2OPredict(newdata=iris_test,
                        newdata_partition_column='id',
                        newdata_order_column='id',
                        modeldata=modeldata,
                        modeldata_order_column='model_id',
                        model_output_fields=['label', 'classProbabilities'],
                        accumulate=['id', 'sepal_length', 'petal_length'],
                        overwrite_cached_models='*',
                        enable_options='stageProbabilities',
                        model_type='OpenSource')
```

- Print the results.

```
print(result.result)
```

Example 3: Run a query with a licensed model

This example runs a query with a licensed model with id 'licensed_model1' from the table 'byom_licensed_models' and associated license key stored in column 'license_key' of the table 'license' present in the schema 'mldb'.

- Retrieve model.

```
modeldata = retrieve_byom('licensed_model1',
                        table_name='byom_licensed_models',
                        license='license_key',
```

```
is_license_column=True,
license_table_name='license',
license_schema_name='mldb')
```

- Pass the output of the retrieve_model API as an input to the H2OPredict function to score data.

```
result = H2OPredict(newdata=iris_test,
                    newdata_partition_column='id',
                    newdata_order_column='id',
                    modeldata=modeldata,
                    modeldata_order_column='model_id',
                    model_output_fields=['label', 'classProbabilities'],
                    accumulate=['id', 'sepal_length', 'petal_length'],
                    overwrite_cached_models='*',
                    enable_options='stageProbabilities',
                    model_type='OpenSource')
```

- Print the results.

```
print(result.result)
```

ONNXPredict

ONNXPredict performs a prediction on each row of the input table using a model previously trained in ONNX and then loaded into the database. The model uses an interchange format called as ONNX and it is loaded to the database in Vantage in a table by the user as a blob.

The following are examples of ONNXPredict() function call.

Example Setup

- Import necessary modules.

```
>>> import os, teradataml
```

```
>>> from teradataml.options.configure import configure
```

```
>>> from teradataml import DataFrame, load_example_data,
save_byom, retrieve_byom
```

- Load example data.

```
>>> load_example_data("byom", "iris_test")
```

- Create teradataml DataFrame object.

```
>>> iris_test = DataFrame("iris_test")
```

- Set install location of the BYOM functions.

```
>>> configure.byom_install_location = "mldb"
```

Example 1: Predict the flower species using trained 'skl_model' model

- Load model file into Vantage.

```
>>> model_file_path = os.path.join(os.path.dirname(teradataml.__file__),
    "data", "models")
```

```
>>> skl_model_file = os.path.join(model_file_path,
    "iris_db_dt_model_sklearn.onnx")
```

- Save the model.

```
>>> save_byom("iris_db_dt_model_sklearn",
    skl_model_file, "byom_models")
```

- Retrieve model.

```
>>> skl_model = retrieve_byom("iris_db_dt_model_sklearn",
    table_name="byom_models")
```

- Using ONNXPredict to score data.

```
>>> ONNXPredict_out = ONNXPredict(accumulate="id",
    newdata=iris_test,
    modeldata=skl_model)
```

save_byom()

The `save_model()` API allows a user to save externally trained models in Vantage in a specified table. This function allows users to save various models stored in different formats such as PMML, MOJO, and so on. If the specified model table exists in Vantage, model data is saved in that table. Otherwise, model table is created first based on the user parameters and then model data is saved.

If the user specified table exists, then

- The table must have at least two columns with names and types as specified in the following:
 - 'model_id' column of type VARCHAR of any length
 - 'model' column of type BLOB
- User can choose to have additional columns to store additional information of the model. This information can be passed using *additional_columns* parameter. See the following argument description for details.

If the user specified table does not exist, then

- This function creates the table with the name specified in *table_name*.
- The table is created in the schema specified in *schema_name*.
If *schema_name* is not specified, then current schema is used.
- The table is created with the following columns:
 - 'model_id' column with type specified in *additional_columns_types*.
If type is not specified, table is created with 'model_id' column as VARCHAR(128).
 - 'model' column with type specified in "additional_columns_types".
If type is not specified, table is created with 'model' column as BLOB.
 - Columns specified in *additional_columns* parameter.
See *additional_columns* argument description for more details.
 - Data types of these additional columns are either taken from the values passed to *additional_columns_types* or inferred from the values passed to the *additional_columns*.
See *additional_columns_types* argument description for more details.

Required Arguments:

- *model_id* specifies the unique model identifier for this model.
- *model_file* specifies the absolute path of the file which has model information.

Optional Arguments:

- *table_name* specifies the name of the table where model is saved.

If a table with this *table_name* does not exist, this function creates the table according to *additional_columns* and *additional_columns_types*.

Note:

- You must either specify this argument, or set the byom model catalog table name using [set_byom_catalog\(\)](#).
 - If none of them are set, exception is raised; If both of them are set, the settings in `save_byom()` take precedence and is used for function execution when saving an model.
-

- *schema_name* specifies the name of the schema/database in which the table specified in *table_name* is looked up.

If this argument is not specified, then table is looked up in the schema associated with the current context.

Note:

- You must either specify this argument and the *table_name* argument, or set the byom model catalog schema name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `save_byom()` take precedence and is used for function execution when saving an model.
- If you specify *schema_name*, *table_name* has to be specified, else exception is raised.

The following table shows the system behavior based on different input combinations.

| In <code>save_byom()</code> | | In <code>set_byom_catalog()</code> | | System Behavior |
|-----------------------------|--------------------|------------------------------------|--------------------|---|
| <i>table_name</i> | <i>schema_name</i> | <i>table_name</i> | <i>schema_name</i> | |
| Set | Set | Set | Set | <i>schema_name</i> and <i>table_name</i> in <code>save_byom()</code> are used for function execution. |
| | | Not set | Not set | <i>schema_name</i> and <i>table_name</i> in <code>save_byom()</code> is used for function execution. |
| | Not set | Set | Set | <i>table_name</i> from <code>save_byom()</code> is used and schema name associated with the current context is used. |
| Not set | Set | Set | Set | Exception is raised. |
| | Not set | Set | Set | <i>table_name</i> and <i>schema_name</i> from <code>set_byom_catalog()</code> are used for function execution. |
| | | Set | Not set | <i>table_name</i> from <code>set_byom_catalog()</code> is used and schema name associated with the current context is used. |
| | | Not set | Not set | Exception is raised. |

- *additional_columns* specifies the additional information about the model to be saved in the model table.

Additional information about the model is passed as key value pair, where key is the name of the column and value is data to be stored in that column for the model being saved. Allowed types for the values passed in dictionary are:

- int
- float
- str
- bool
- datetime.datetime
- datetime.date

- `datetime.time`

Note:

additional_columns does not accept keys `model_id` and `model`.

- *additional_columns_types* specifies the column type of additional columns. These column types are used while creating the table using the columns specified in the *additional_columns* argument.

Additional column datatype information is passed as key value pair, where key is the column name and value is `teradatasqlalchemy.types`.

Note:

- If any of the column type for additional columns are not specified in *additional_columns_types*, then it derives the column types. To get more information on column type mapping, refer to parameters using `help(save_byom)`.
- For columns `model_id` and `model`, column type must be mandatorily set to `VARCHAR` and `BLOB` respectively for the table. Thus, for the columns `model_id` and `model`, acceptable values for *additional_columns_types* are `VARCHAR` and `BLOB` respectively.
- This argument is ignored if the table exists.

Example Setup

- Import necessary modules.

```
>>> import teradataml, os, datetime

>>> # import save_byom from teradataml
>>> from teradataml import save_byom
```

- Get the model file path to use in the examples.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                              "models", "iris_kmeans_model")
```

Example 1: Create a table with additional columns and column types specified

This example creates a table "byom_models" with additional columns by specifying the type of the columns as listed in the following table and save the model in it.

| Column Name | Column Type |
|--------------------------|----------------------------|
| <code>model_id</code> | <code>VARCHAR(128)</code> |
| <code>model</code> | <code>BLOB</code> |
| <code>Description</code> | <code>VARCHAR(2000)</code> |

| Column Name | Column Type |
|--------------------|---------------|
| UserId | NUMBER(5) |
| ProductionReady | BYTEINT |
| ModelEfficiency | NUMBER(11,10) |
| ModelSavedTime | TIMESTAMP |
| ModelGeneratedDate | DATE |
| ModelGeneratedTime | TIME |

```
>>> save_byom("model1",
              model_file,
              "byom_models",
              additional_columns={"Description": "KMeans model",
                                "UserId": "12345",
                                "ProductionReady": False,
                                "ModelEfficiency": 0.67412,
                                "ModelSavedTime": datetime.datetime.now(),
                                "ModelGeneratedDate": datetime.date.today(),
                                "ModelGeneratedTime": datetime.time(hour=0,minute=5,second=45,microsecond=110)
                                },
              additional_columns_types={"Description": VARCHAR(2000),
                                       "UserId": NUMBER(5),
                                       "ProductionReady": BYTEINT,
                                       "ModelEfficiency": NUMBER(11,10),
                                       "ModelSavedTime": TIMESTAMP,
                                       "ModelGeneratedDate": DATE,
                                       "ModelGeneratedTime": TIME}
              )
Created the table 'byom_models' as it does not exist.
Model is saved.
```

Example 2: Create a table with additional columns and column types not specified

This example creates a table "byom_models1" in the "test" database, with additional columns by not specifying the type of the columns. Once table is created, save the model in it.

```
>>> save_byom("model1",
              model_file,
              "byom_models1",
              additional_columns={"Description": "KMeans model",
```

```

        "UserId": "12346",
        "ProductionReady": False,
        "ModelEfficiency": 0.67412,
        "ModelSavedTime": datetime.datetime.now(),
        "ModelGeneratedDate": datetime.date.today(),

"ModelGeneratedTime": datetime.time(hour=0,minute=5,second=45,microsecond=110)
    },
    schema_name="test"
)
Created the table 'byom_models1' as it does not exist.
Model is saved.

```

Example 3: Save a model in an existing table

This example saves a model in the existing table "byom_models".

```

>>> save_byom("model2",
               model_file,
               "byom_models",
               additional_columns={"Description": "KMeans model duplicated"})
Model is saved.

```

Example 4: Use the BYOM model catalog information set at session level by set_byom_catalog()

This example shows when save_byom() function is called without arguments *table_name* and *schema_name*, system uses the BYOM model catalog information set at session level by the set_byom_catalog() function.

- The model cataloging parameters are set using the set_byom_catalog() function.

```

>>> set_byom_catalog(table_name='byom_models', schema_name='alice')

The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'

```

- The model is saved in the existing table specified by the arguments in set_byom_catalog() function.

```

>>> save_byom('model3', model_file=model_file)
Model is saved.

```

Example 5: Override session level parameters for BYOM model cataloging

This example shows that you can override the session level BYOM model catalog information set by the `set_byom_catalog()`, by passing the table as well as schema arguments explicitly.

- The model cataloging table name is set to 'byom_models' using the `set_byom_catalog()` function.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')

The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'.
```

- The model is saved in a different table 'byom_licensed_models' which is specified by the *table_name* argument in this `save_byom()` function.

```
>>> save_byom('licensed_model2',
model_file=model_file, table_name='byom_licensed_models',
additional_columns={"license_data": "A5sUL9KU_kP35Vq"})

Created the model table 'byom_licensed_models' as it does not exist.
Model is saved.
```

list_byom()

The `list_byom()` API allows a user to list saved models, filtering the results based on the optional arguments.

Optional Arguments:

- table_name* specifies the name of the table to list models from.

Note:

- You must either specify this argument, or set the byom model catalog table name using [set_byom_catalog\(\)](#).
 - If none of them are set, exception is raised; If both of them are set, the settings in `list_byom()` take precedence and is used for function execution.
-

- schema_name* specifies the name of the schema in which the table specified in *table_name* is looked up.

If this argument is not specified, then table is looked up in the schema associated with the current context.

Note:

- You must either specify this argument and the *table_name* argument, or set the byom model catalog schema name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `list_byom()` take precedence and is used for function execution.
- If you specify *schema_name*, *table_name* has to be specified, else exception is raised.

See table in [save_byom\(\)](#) that shows different system behaviors based on different input combinations.

- *model_id* specifies the unique model identifier of the model(s).

If specified, the models with either exact match or a substring match, are listed.

Example Setup

- Import necessary modules.

```
>>> import teradataml, os, datetime
```

```
>>> from teradataml import save_byom, list_byom
```

- Get the model file path.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                              "models", "iris_kmeans_model")
```

- Save three models with different parameters.

```
>>> save_byom("model7", model_file, "byom_models")
Model is saved.
```

```
>>> save_byom("iris_model1", model_file, "byom_models")
Model is saved.
```

```
>>> save_byom("model8", model_file, "byom_models", schema_name="test")
Model is saved.
```

Example 1: List all models in a table

This example lists all the models saved in the table "byom_models".

```
>>> list_byom(table_name="byom_models")
              model
model_id
```

```
model7      b'504B03041400080808...'
iris_model1 b'504B03041400080808...'
```

Example 2: List all models with model_id containing a specific string

This example lists all the models with *model_id* containing the string "iris", from the "byom_models" table.

```
>>> list_byom(table_name="byom_models", model_id="iris")
      model
model_id
iris_model1 b'504B03041400080808...'
```

Example 3: List all models with model_id containing one of the two strings

This example lists all the models with *model_id* containing either "iris" or "7" string, from the "byom_models" table.

```
>>> list_byom(table_name="byom_models", model_id=["iris", "7"])
      model
model_id
model7      b'504B03041400080808...'
iris_model1 b'504B03041400080808...'
```

Example 4: List all models in a table in the specified database

This example lists all the models from the "byom_models" in the "test" database.

```
>>> list_byom(table_name="byom_models", schema_name="test")
      model
model_id
model8      b'504B03041400080808...'
```

Example 5: List all the models from the model cataloging table set by set_byom_catalog()

- The model cataloging parameters are set using the `set_byom_catalog()` function.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')

The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'
```

- List all the models.

```
>>> list_byom()
      model
```



```
model_id
model18 b'504B03041400080808...'
```

Example 6: Override session level parameters for BYOM model cataloging

This example shows that you can override the session level BYOM model catalog information set by the `set_byom_catalog()`, by passing the table as well as schema arguments explicitly.

- The model cataloging table name is set to 'byom_models' using the `set_byom_catalog()` function.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')

The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'.
```

- List all the models from the table specified by the *table_name* argument in this `list_byom()` function.

```
>>> list_byom(table_name='byom_licensed_models')
model
model_id
iris_model11 b'504B03041400080808...'
```

retrieve_byom()

The `retrieve_byom()` API allows a user to retrieve a saved model. Output of this function can be directly passed as input to the `PMMLPredict` and `H2OPredict` functions.

Note:

Some models, such as H2O-DAI, have licenses associated with the models.

When these models are used for scoring, users must retrieve the model by passing relevant license information. See the *license* argument for details.

Required Arguments:

- model_id* specifies the unique model identifier of the model to be retrieved.

Optional Argument:

-
- table_name* specifies the name of the table to retrieve external model from.

Note:

- You must either specify this argument, or set the byom model catalog table name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `retrieve_byom()` take precedence and is used for function execution.

- *schema_name* specifies the name of the schema in which the table specified in *table_name* is looked up.

If this argument is not specified, then table is looked up in the schema associated with the current context.

Note:

- You must either specify this argument and the *table_name* argument, or set the byom model catalog schema name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `retrieve_byom()` take precedence and is used for function execution.
- If you specify *schema_name*, *table_name* has to be specified, else exception is raised.

See table in [save_byom\(\)](#) that shows different system behaviors based on different input combinations.

License related optional arguments:

- *license* specifies either the license key itself or name of the column that contains the license key, based on where the license key is stored:
 - If the license key is stored in a variable, users can pass it as string.
license contains the license key itself.
 - If the license key is stored in a table, users pass the name of the column containing the license key.
license contains the name of the column containing the license key. *is_license_column* must be set to True.

Also, based on the table where the license information is stored,

 - If the license information is stored in the same model table as that of the model specified in the argument *table_name*, users only need to specify the name of the column containing the license key.
 - If the license information is stored in a table different from the one specified in the argument *table_name*, in addition to the column name, users can specify the table name and schema name using *license_table_name* and *license_schema_name* respectively.
- *is_license_column* specifies whether the argument *license* is a license key or column name.
 - When set to True, *license* contains the name of the column that contains the license key.

- Otherwise, *license* contains the actual license key.

- *license_table_name* specifies the name of the table that holds the license key, if the license key is stored in a table other than the table specified in *table_name*.
- *license_schema_name* specifies the name of the database associated with the *license_table_name*.

If not specified, current database is used.

- *require_license* specifies whether the model to be retrieved is associated with a license.

The default value is False. If set to True, license information set by [set_license\(\)](#) is retrieved.

Note:

If license parameters are passed, then this argument is ignored.

Example Setup

- Import necessary modules.

```
>>> import teradataml, os, datetime
```

```
>>> from teradataml import save_byom, retrieve_byom, get_context
```

- Get the model file path.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                               "models", "iris_kmeans_model")
```

- Save four models with different parameters.

- Save a model in the same database without license.

```
>>> save_byom("model9", model_file, "byom_models")
Model is saved.
```

- Save a model in another database without license.

```
>>> save_byom("model6", model_file, "byom_models", schema_name="test")
Model is saved.
```

- Save a model in the same database with associated license stored in a variable.

```
>>> save_byom("model5", model_file, "byom_models")
Model is saved.
```

- Save a model in the same database with license key stored in an additional column "license_data" in the same table, for example 4.

```
>>> save_byom("licensed_model1", model_file, "byom_licensed_models",
               additional_columns={"license_data": "A5sUL9KU_kP35Vq"})
```

```
Created the model table "byom_licensed_models" as it does not exist.
Model is saved.
```

- Store a license in a variable "lic_key", for "model5" in example 3.

```
>>> lic_key = "eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI"
```

- Store a license in a table "license" in the column "license_key", for example 5.

```
>>> # Store the license in a table.
>>> lic_table = "create table license (id integer between 1 and 1,license_key
varchar(2500)) unique primary index(id);"
>>> get_context().execute(lic_table)
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000014AAFF27080>
>>> get_context().execute("insert into license values (1, "peBVRtjA-ib)")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000014AAFF27278>
```

This table is also created in database ""mldb", for example 6.

Example 1: Retrieve a model from a specific table

This example retrieves a model with id "model9" from the table "byom_models".

```
>>> df = retrieve_byom("model9", table_name="byom_models")
```

```
>>> df
              model
model_id
model9    b"504B03041400080808..."
```

Example 2: Retrieve a model from a specific table in a specific database

This example retrieves a model with id "model6" from the table "byom_models" in the "test" database.

```
>>> df = retrieve_byom("model6", table_name="byom_models", schema_name="test")
```

```
>>> df
              model
model_id
model6    b"504B03041400080808..."
```

Example 3: Retrieve a model from a specific table, with license key stored in a variable

This example retrieves a model with id "model5" from the table "byom_models" with license key stored in a variable "lic_key".

```
>>> df = retrieve_byom("model5", table_name="byom_models", license=lic_key)
```

```
>>> df
```

| model | license |
|----------|---|
| model_id | |
| model5 | b"504B03041400080808..." eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI |

Example 4: Retrieve a model from a specific table, with license key stored in the same table

This example retrieves a model with id "licensed_model1" from the table "byom_licensed_models", and associated license key stored in the same table in the column "license_data".

```
>>> df = retrieve_byom("licensed_model1",
                        table_name="byom_licensed_models",
                        license="license_data",
                        is_license_column=True)
```

```
>>> df
```

| | model | license |
|-----------------|--------------------------|-----------------|
| model_id | | |
| licensed_model1 | b"504B03041400080808..." | A5sUL9KU_kP35Vq |

Example 5: Retrieve a model from a specific table, with license key stored in another table in the same database

This example retrieves a model with id "licensed_model1" from the table "byom_licensed_models", and associated license key is stored in the column "license_key" of another table "license".

```
>>> df = retrieve_byom("licensed_model1",
                        table_name="byom_licensed_models",
                        license="license_key",
                        is_license_column=True,
                        license_table_name="license")
```

```
>>> df
```

| | model | license |
|-----------------|--------------------------|-------------|
| model_id | | |
| licensed_model1 | b"504B03041400080808..." | peBVRtjA-ib |

Example 6: Retrieve a model from a specific table, with license key stored in a table in another database

```
>>> df = retrieve_byom('licensed_model1',
                        table_name='byom_licensed_models',
                        license='license_key',
                        is_license_column=True,
                        license_table_name='license',
                        license_schema_name='mldb')
```

```
>>> df
```

| | model | license |
|-----------------|--------------------------|-------------|
| model_id | | |
| licensed_model1 | b'504B03041400080808...' | peBVRtjA-ib |

Example 7: Retrieve a model with license key set at session level by set_license(), license is passed as a variable

In this example, retrieve a model with id 'model5' from the table 'byom_models', with license key stored by set_license() in a variable 'license'.

The catalog information is set using set_byom_catalog() to table_name='byom_models', schema_name='alice', and is used to retrieve the model.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')
```

The model cataloging parameters are set to table_name='byom_models' and schema_name='alice'

```
>>> set_license(license=license, source='string')
```

The license parameters are set.

The license is: eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI

```
>>> df = retrieve_byom('model5', require_license=True)
```

```
>>> df
```

| | model | license |
|----------|--------------------------|--|
| model_id | | |
| model5 | b'504B03041400080808...' | eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI |

Example 8: Retrieve a model with license key set at session level by set_license() where license is passed as a file

In this example, retrieve a model with id 'model5' from the table 'byom_models', with license key stored by set_license() in a file.

Since the schema name is not provided, default schema is used.

```
>>> license_file = os.path.join(os.path.dirname(teradataml.__file__), 'data',
'models', 'License_file.txt')
```

```
>>> set_license(license=license_file, source='file')
```

```
The license parameters are set.
The license is: license_string
```

```
>>> df = retrieve_byom('model5', table_name='byom_models', require_license=True)
```

```
>>> df
```

```
model license
model_id
model5 b'504B03041400080808...' license_string
```

Example 9: Retrieve a model with license key stored in a column of another table set at session level by set_license()

In this example, retrieve a model with id 'licensed_model1' with associated license key stored in the column 'license_key' of the table 'license' present in the schema 'alice'.

The byom catalog and license information is set using set_byom_catalog() and set_license() respectively.

Function is executed with license parameters passed, which overrides the license information set at the session level.

```
>>> set_byom_catalog(table_name='byom_licensed_models', schema_name='alice')
```

```
The model cataloging parameters are set to table_name='byom_licensed_models'
and schema_name='alice'
```

```
>>> set_license(license=license, source='string')
```

```
The license parameters are set.
The license is: eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI
```

```
>>> df = retrieve_byom('licensed_model1', license='license_key',
is_license_column=True, license_table_name='license')
```

```
>>> df
```

```
model license
model_id
licensed_model1 b'504B03041400080808...' peBVRtjA-ib
```

Example 10: Retrieve a model with license key stored in a column of same table set at session level by set_license()

In this example, retrieve a model with id 'licensed_model1' from the table 'byom_licensed_models' with associated license key stored in the column 'license_data' of the table 'byom_licensed_models'.

The byom catalog and license information is already set at the session level, passing the table_name to the function call overrides the byom catalog information at the session level.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')
```

The model cataloging parameters are set to table_name='byom_models' and schema_name='alice'

```
>>> set_license(license='license_data', table_name='byom_licensed_models',
schema_name='alice', source='column')
```

The license parameters are set.

The license is present in the table='byom_licensed_models', schema='alice' and column='license_data'.

```
>>> df = retrieve_byom('licensed_model1',
table_name='byom_licensed_models', require_license=True)
```

```
>>> df
```

```
model license
model_id
licensed_model1 b'504B03041400080808...' A5sUL9KU_kP35Vq
```


Note:

If `require_license=False`, which is the default value, the license information is not retrieved.

```
>>> df = retrieve_byom('licensed_model1', table_name='byom_licensed_models')
```

```
>>> df
model
model_id
licensed_model1 b'504B03041400080808...'
```

delete_byom()

The `delete_byom()` API allows a user to delete a model from the user specified table in Vantage.

Required Arguments:

- *model_id* specifies the unique model identifier of the model to be deleted.

Optional Argument:

-
- *table_name* specifies the name of the table to delete the model from.

Note:

- You must either specify this argument, or set the byom model catalog table name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `delete_byom()` take precedence and is used for function execution.

- *schema_name* specifies the name of the schema in which the table specified in *table_name* is looked up.

If this argument is not specified, then table is looked up in the schema associated with the current context.

Note:

- You must either specify this argument and the *table_name* argument, or set the byom model catalog schema name using [set_byom_catalog\(\)](#).
- If none of them are set, exception is raised; If both of them are set, the settings in `delete_byom()` take precedence and is used for function execution.
- If you specify *schema_name*, *table_name* has to be specified, else exception is raised.

See table in [save_byom\(\)](#) that shows different system behaviors based on different input combinations.

Example Setup

- Import necessary modules.

```
>>> import teradataml, os, datetime
```

```
>>> # importing delete_byom from teradataml
>>> from teradataml from teradataml import save_byom, delete_byom
```

- Get the model file path to use it in examples.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), "data",
                               "models", "iris_kmeans_model")
```

- Save some models with different parameters.

```
>>> save_byom("model3", model_file, "byom_models")
Model is saved.
```

```
>>> save_byom("model4", model_file, "byom_models", schema_name="test")
Model is saved.
```

Example 1: Delete a model with specific model_id from a specific table

This example deletes a model with id "model3" from the table "byom_models".

```
>>> delete_byom(model_id="model3", table_name="byom_models")
Model is deleted.
```

Example 2: Delete a model with specific model_id from a specific table in a specific database

This example deletes a model with id "model4" from the table "byom_models" is in the "test" database.

```
>>> delete_byom(model_id="model4", table_name="byom_models", schema_name="test")
Model is deleted.
```

Example 3: Delete a model with specific model_id from a table specified by set_byom_catalog() function

- The model cataloging parameters are set using the set_byom_catalog() function.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')
```

```
The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'
```

- Delete the a model with id 'model4' from the model catalog table specified by set_byom_catalog() function.

```
>>> delete_byom(model_id='model4')
```

```
Model is deleted.
```

Example 4: Override session level parameters for BYOM model cataloging

This example shows that you can override the session level BYOM model catalog information set by the set_byom_catalog(), by passing the table as well as schema arguments explicitly.

- The model cataloging table name is set to 'byom_models' using the set_byom_catalog() function.

```
>>> set_byom_catalog(table_name='byom_models', schema_name='alice')
```

```
The model cataloging parameters are set to table_name='byom_models'
and schema_name='alice'.
```

- The model is saved in a different table 'byom_licensed_models' which is specified by the *table_name* argument in the save_byom() function.

```
>>> save_byom('licensed_model2',
model_file=model_file, table_name='byom_licensed_models')
```

```
Created the model table 'byom_licensed_models' as it does not exist.
Model is saved.
```

- Delete the model from the table specified by the *table_name* argument in this delete_byom() function.

```
>>> delete_byom(model_id='licensed_model2', table_name='byom_licensed_models')
```

```
Model is deleted.
```

set_byom_catalog()

The set_byom_catalog() function allows a user to set the BYOM model catalog information, such as a table name and schema name, at the session level for the following BYOM model cataloging APIs:

- [save_byom\(\)](#)
- [list_byom\(\)](#)
- [retrieve_byom\(\)](#)
- [delete_byom\(\)](#)

- [set_license\(\)](#)

Required Argument:

- *table_name* specifies the name of the table to be used for BYOM model cataloging. This table is used for saving, retrieving BYOM model information by BYOM model cataloging APIs.

Optional Argument:

- *schema_name* specifies the name of the schema/database in which the table specified in *table_name* is looked up.

If not specified, then table is looked up in current schema/database.

Example Setup

```
>>> from teradataml import set_byom_catalog
```

Example 1: Set global parameters *table_name* and *schema_name*

```
>>>
set_byom_catalog(table_name='model_table_name', schema_name='model_schema_name')
The model cataloging parameters are set to table_name='model_table_name'
and schema_name='model_schema_name'
```

set_license()

The `set_license()` function allows a user to set the license information associated with the externally generated model in a session level variable which is required by H2O DAI models. It is used by the [retrieve_byom\(\)](#) function to retrieve the license information while retrieving the specific model.

If specified table name does not exist and is not the same as BYOM catalog tables, then this function creates the table and stores the license information; otherwise, this function just validates and sets the license information.

The license can be set by passing the license in the following ways:

- Passing the license as a variable;
- Passing the column name in the model table itself;
- Passing the table and the column name containing the license;
- Passing the license in a file.

Required Arguments:

- *license* specifies the license key information that can be passed as:
 - a variable;
 - in a file;
 - name of the column containing license information in a table specified by the *table_name* argument.

Note:

Argument *source* must be set accordingly.

- *source* specifies whether the license key specified in *license* is a string, file or column name. The default value is string.

Optional Arguments:

- *table_name* specifies the name of the table containing the license information.
- *schema_name* specifies the name of the schema in which the table specified in *table_name* is looked up.

If not specified, then table is looked up in current schema/database.

Note:

Arguments *table_name* and *schema_name* should be either both specified or both not specified.

Example Setup

```
>>> import os, teradataml
```

```
>>> from teradataml import save_byom, retrieve_byom, get_context,
set_license, set_byom_catalog
```

Example 1: License is passed as a string

```
>>> set_license(license='eZSy3peBVRtjA-
ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI',
table_name=None, schema_name=None, source='string')
```

The license parameters are set.

The license is : eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI

Example 2: License is stored in a file

The file that contains the license is passed as input to *license* and *source* must be set to 'file'.

```
>>> license_file = os.path.join(os.path.dirname(teradataml.__file__), 'data',
'models', 'License_file.txt')
```

```
>>> set_license(license=license_file, source='file')
```

The license parameters are set.

The license is: license_string

Example 3: License is present in the byom model catalog table itself

- Store a model with license information in the model table.

```
>>> model_file = os.path.join(os.path.dirname(teradataml.__file__), 'data',
                                'models', 'iris_kmeans_model')
```

```
>>> save_byom('licensed_model1', model_file, 'byom_licensed_models',
              additional_columns={"license_data": "A5sUL9KU_kP35Vq"})
```

Created the model table 'byom_licensed_models' as it does not exist.
Model is saved.

```
>>> set_byom_catalog(table_name='byom_licensed_models', schema_name='alice')
```

The model cataloging parameters are set to table_name='byom_licensed_models' and schema_name='alice'.

- Set license.

```
>>> set_license(license='license_data', source='column')
```

The license parameters are set.
The license is present in the table='byom_licensed_models', schema='alice' and column='license_data'.

Example 4: License is stored in a column

The license information is stored in the column 'license_key' of a table 'license_table'.

- Create a table and insert the license information in the table.

```
>>> license = 'eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI'
```

```
>>> lic_table = 'create table license (id integer between 1 and 1,
                                license_key varchar(2500)) unique primary index(id);'
```

```
>>> get_context().execute(lic_table)
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x000001DC4F2EE9A0>
```

```
>>> get_context().execute("insert into license values (1, 'peBVRtjA-ib')")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x000001DC4F2EEF10>
```

- Set license.

```
>>> set_license(license='license_key', table_name='license',
schema_name='alice', source='column')
```

The license parameters are set.

The license is present in the table='license', schema='alice' and column='license_key'.

get_license()

The `get_license()` function allows a user to get the license information set by the [set_license\(\)](#) at the session level.

Example Setup

```
>>> import os, teradataml
```

```
>>> from teradataml import save_byom, get_license, set_license
```

Example 1: License is passed as a string

```
>>> set_license(license='eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI',
source='string')
```

The license parameters are set.

The license is: eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI

```
>>> get_license()
```

The license is: eZSy3peBVRtjA-ibVuvNw5A5sUL9KU_kP35Vq4ZNBQ3iGY6oVSpE6g97sFY2LI

Example 2: License is present in a column of a table

In this example, license is present in column 'license_data' of the table 'byom_licensed_models'.

```
>>> set_license(license='license_data',
table_name='byom_licensed_models', schema_name='alice',
source='column')
```

The license parameters are set.

The license is present in the table='byom_licensed_models', schema='alice' and column='license_data'.

```
>>> get_license()
```

The license is stored in:

```
table = 'byom_licensed_models'
```

```
schema = 'alice'
```

```
column = 'license_data'
```

BYOM Workflows

Download the `teradataml_workflows_byom.zip` from the attachment in the left sidebar. The zip file includes Jupyter notebooks and HTML files for sample BYOM workflows in the `byom` folder.

Working with Geospatial Data

Geospatial information identifies the geographic location of features and boundaries on the planet. Vantage provides Geospatial types that address the need to store, manage, retrieve, manipulate, and analyze geospatial information by governmental bodies and commercial enterprises. For example, local governments might use geospatial data for city planning, traffic management, or accident investigation.

Vantage provides geospatial types to represent geometries with up to three dimensions. Vantage provides the ST_Geometry, MBB, and MBR data types for creating and manipulating geometric shapes in the database. ST_Geometry is implemented as a user-defined type (UDT). ST_Geometry can be used as the data type of a table column to represent most of the geospatial types specified in the standard.

Vantage supports the following types of geometry shapes, which can be stored in a table column of ST_Geometry data type.

| Type | Description |
|--------------------|--|
| ST_Point | 0-dimensional geometry that represents a single location in two-dimensional coordinate space. |
| ST_LineString | 1-dimensional geometry usually stored as a sequence of points with a linear interpolation between points. |
| ST_Polygon | 2-dimensional geometry consisting of one exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole. |
| ST_GeomCollection | Collection of zero or more ST_Geometry values. |
| ST_MultiPoint | 0-dimensional geometry collection where the elements are restricted to ST_Point values. |
| ST_MultiLineString | 1-dimensional geometry collection where the elements are restricted to ST_LineString values. |
| ST_MultiPolygon | 2-dimensional geometry collection where the elements are restricted to ST_Polygon values. |
| GeoSequence | Extension of ST_LineString that can contain tracking information, such as time stamps, in addition to geospatial information. GeoSequence is a Teradata extension to SQL/MM Spatial. |

teradataml provides following generic functionalities to access and process geospatial information stored in Vantage:

- [teradataml Geometry Types](#)
- [teradataml GeoDataFrame](#)
- [teradataml GeoDataFrameColumn](#)

teradataml Geometry Types

teradataml provides support to create the following geometry objects:

- [Point](#)
- [LineString](#)
- [Polygon](#)
- [MultiPoint](#)
- [MultiLineString](#)
- [MultiPolygon](#)
- [GeometryCollection](#)
- [GeoSequence](#)

This allows user to create a geometry object of any type, that can be used for processing [teradataml GeoDataFrame](#) and [teradataml GeoDataFrameColumn](#). These objects can be passed as argument values or can be used in slice filtering with Geospatial specific methods and properties of teradataml GeoDataFrame and teradataml GeoDataFrameColumn.

See [Teradata Package for Python Function Reference](#) for detailed information and examples for geometry objects.

Example: Geometry type generic usage

```
from teradataml import GeoDataFrame, LineString

# Create a GeoDataFrame on table "sample_streets"
geoDF = GeoDataFrame("sample_streets")

# Create a LineString object.
l1 = LineString([(0, 0), (0, 20), (20, 20)])

# Case 1: Using Geometry type object as argument value for the Geospatial
specific functions.
geoDF[geoDF.streetShape.crosses(l1) < 1E0].select("streetName")

# Case 2: Using Geometry type object in slice filtering.
geoDF[geoDF.streetShape == l1]
```

Point

This function enables end user to create an object for the single Point using the coordinates. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

coordinates: Specifies the coordinates of a point. Coordinates must be specified in positional fashion.

- If coordinates are not passed, an object for empty point is created.
- When coordinates are passed, you must pass one of the following:
 - Two values to define a point in 2-dimensions;
 - Three values to define a point in 3-dimensions.

Example 1: Create a Point in 2D, using x and y coordinates

```
>>> from teradataml import Point
```

```
>>> p1 = Point(0, 20)
```

```
>>> # Print the coordinates.
```

```
>>> print(p1.coords)
(0, 20)
```

```
>>> # Print the geometry type.
```

```
>>> p1.geom_type
'Point'
```

Example 2: Create a Point in 3D, using x, y and z coordinates

```
>>> from teradataml import Point
```

```
>>> p2 = Point(0, 20, 30)
```

```
>>> # Print the coordinates.
```

```
>>> print(p2.coords)
(0, 20, 30)
```

Example 3: Create an empty Point

```
>>> from teradataml import Point
```

```
>>> pe = Point()
```

```
>>> # Print the coordinates.
```

```
>>> print(pe.coords)
EMPTY
```

LineString

This function enables end user to create an object for the single LineString using the coordinates. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

coordinates: Specifies the coordinates of a line.

- If coordinates are not passed, an object for empty line is created.
- When coordinates are passed, you must pass a list of the following types:
 - Two-tuple of int or float;
 - Three-tuple of int or float;
 - Point geometry objects;
 - Mix of any of these three types.

Example 1: Create a LineString in 2D, using x and y coordinates

```
>>> from teradataml import Point, LineString

>>> l1 = LineString([(0, 0), (0, 20), (20, 20)])

>>> # Print the coordinates.
>>> print(l1.coords)
[(0, 0), (0, 20), (20, 20)]

>>> # Print the geometry type.
>>> l1.geom_type
'LineString'
```

Example 2: Create a LineString in 3D, using x, y and z coordinates

```
>>> from teradataml import Point, LineString

>>> l2 = LineString([(0, 0, 1), (0, 1, 3), (1, 3, 6), (3, 3, 6), (3, 6, 1), (6,
3, 3), (3, 3, 0)])

>>> # Print the coordinates.
>>> print(l2.coords)
[(0, 0, 1), (0, 1, 3), (1, 3, 6), (3, 3, 6), (3, 6, 1), (6, 3, 3), (3, 3, 0)]
```

Example 3: Create a LineString using Point geometry objects

```
>>> from teradataml import Point, LineString

# Create some Points in 2D, using x and y coordinates.
>>> p1 = Point(0, 20)
>>> p2 = Point(0, 0)
>>> p3 = Point(20, 20)
```

```
>>> l3 = LineString([p1, p2, p3])
```

```
>>> # Print the coordinates.
>>> print(l3.coords)
[(0, 20), (0, 0), (20, 20)]
```

Example 4: Create a LineString using mix of Point geometry objects and coordinates

```
>>> from teradataml import Point, LineString
```

```
>>> p1 = Point(0, 20)
>>> p2 = Point(20, 20)
```

```
>>> l4 = LineString([(0, 0), p1, p2, (20, 0)])
```

```
>>> # Print the coordinates.
>>> print(l4.coords)
[(0, 0), (0, 20), (20, 20), (20, 0)]
```

Example 5: Create an empty LineString

```
>>> from teradataml import Point, LineString
```

```
>>> le = LineString()
```

```
>>> # Print the coordinates.
>>> print(le.coords)
EMPTY
```

Polygon

This function enables end user to create an object for the single Polygon using the coordinates. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

coordinates: Specifies the coordinates of a polygon.

- If coordinates are not passed, an object for empty polygon is created.
- When coordinates are passed, you must pass a list of the following types:
 - Two-tuple of int or float;
 - Three-tuple of int or float;
 - Point geometry objects;

- Polygon geometry objects;
- Mix of any of these four types.

Example 1: Create a Polygon in 2D, using x and y coordinates

```
>>> from teradataml import Point, LineString, Polygon

>>> go1 = Polygon([(0, 0), (0, 20), (20, 20), (20, 0), (0, 0)])

>>> # Print the coordinates.
>>> print(go1.coords)
[(0, 0), (0, 20), (20, 20), (20, 0), (0, 0)]

>>> # Print the geometry type.
>>> go1.geom_type
'Polygon'
```

Example 2: Create a Polygon in 3D, using x, y and z coordinates

```
>>> from teradataml import Point, LineString, Polygon

>>> go2 = Polygon([(0, 0, 0), (0, 0, 20), (0, 20, 0), (0, 20, 20), (20, 0, 0),
(20, 0, 20), (20, 20, 0), (20, 20, 20), (0, 0, 0)])

>>> # Print the coordinates.
>>> print(go2.coords)
[(0, 0, 0), (0, 0, 20), (0, 20, 0), (0, 20, 20), (20, 0, 0), (20, 0, 20), (20,
20, 0), (20, 20, 20), (0, 0, 0)]
```

Example 3: Create a Polygon using Point geometry objects

```
>>> from teradataml import Point, LineString, Polygon

# Create Point objects in 2D, using x and y coordinates.
>>> p1 = Point(0, 20)
>>> p2 = Point(0, 0)
>>> p3 = Point(20, 20)
>>> p4 = Point(20, 0)

>>> go3 = Polygon([p1, p2, p3, p4, p1])
```

```
>>> # Print the coordinates.
>>> print(go3.coords)
[(0, 20), (0, 0), (20, 20), (20, 0), (0, 0)]
```

Example 4: Create a Polygon using LineString geometry objects

```
>>> from teradataml import Point, LineString, Polygon

# Create LineString objects in 2D, using x and y coordinates.
>>> l1 = LineString([(0, 0), (0, 20), (20, 20)])
>>> l2 = LineString([(20, 0), (0, 0)])

>>> go4 = Polygon([l1, l2])

>>> # Print the coordinates.
>>> print(go4.coords)
[(0, 20), (0, 0), (20, 20), (20, 0), (0, 0)]
```

Example 5: Create a Polygon using mix of Point, LineString geometry objects and coordinates

```
>>> from teradataml import Point, LineString, Polygon

>>> p1 = Point(0, 0)
>>> l1 = LineString([p1, (0, 20), (20, 20)])

>>> go5 = Polygon([l1, (20, 0), p1])

>>> # Print the coordinates.
>>> print(go5.coords)
[(0, 0), (0, 20), (20, 20), (20, 0)]
```

Example 5: Create an empty Polygon

```
>>> from teradataml import Point, LineString, Polygon

>>> goe = Polygon()

>>> # Print the coordinates.
>>> print(poe.coords)
EMPTY
```

MultiPoint

This function enables end user to create an object holding the multiple Point geometry objects. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

points: Specifies the list of Point objects. If no points are passed, an object for empty MultiPoint is created.

Example 1: Create a MultiPoint in 2D, using x and y coordinates

```
>>> from teradataml import Point, MultiPoint
```

```
>>> p1 = Point(0, 0)
>>> p2 = Point(0, 20)
>>> p3 = Point(20, 20)
>>> p4 = Point(20, 0)
```

```
>>> go1 = MultiPoint([p1, p2, p3, p4, p1])
```

```
>>> # Print the coordinates.
>>> print(go1.coords)
[(0, 0), (0, 20), (20, 20), (20, 0), (0, 0)]
```

```
>>> # Print the geometry type.
>>> print(go1.geom_type)
MultiPoint
```

Example 2: Create an empty MultiPoint

```
>>> from teradataml import Point, MultiPoint
```

```
>>> poe = MultiPoint()
```

```
>>> # Print the coordinates.
>>> print(poe.coords)
EMPTY
```

MultiLineString

This function enables end user to create an object holding the multiple LineString geometry objects. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

lines: Specifies the list of LineString objects. If no lines are passed, an object for empty MultiLineString is created.

Example 1: Create a MultiLineString in 2D, using x and y coordinates

```
>>> from teradataml import LineString, MultiLineString

>>> l1 = LineString([(1, 3), (3, 0), (0, 1)])
>>> l2 = LineString([(1.35, 3.6456), (3.6756, 0.23), (0.345, 1.756)])

>>> go1 = MultiLineString([l1, l2])

>>> # Print the coordinates.
>>> print(go1.coords)
[[[1, 3), (3, 0), (0, 1)], [(1.35, 3.6456), (3.6756, 0.23), (0.345, 1.756)]]

>>> # Print the geometry type.
>>> print(go1.geom_type)
MultiLineString
```

Example 2: Create an empty MultiLineString

```
>>> from teradataml import LineString, MultiLineString

>>> mls = MultiLineString()

>>> # Print the coordinates.
>>> print(mls.coords)
EMPTY
```

MultiPolygon

This function enables end user to create an object holding the multiple Polygon geometry objects. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

polygon: Specifies the list of Polygon objects. If no polygons are passed, an object for empty MultiPolygon is created.

Example 1: Create a MultiPolygon in 2D, using x and y coordinates

```
>>> from teradataml import Polygon, MultiPolygon
```

```
>>> po1 = Polygon([(0, 0), (0, 20), (20, 20), (20, 0), (0, 0)])
>>> po2 = Polygon([(0.6, 0.8), (0.6, 20.8), (20.6, 20.8), (20.6, 0.8),
(0.6, 0.8)])

>>> go1 = MultiPolygon([po1, po2])

>>> # Print the coordinates.
>>> print(go1.coords)
[[ (0, 0), (0, 20), (20, 20), (20, 0), (0, 0) ], [ (0.6, 0.8), (0.6, 20.8), (20.6,
20.8), (20.6, 0.8), (0.6, 0.8) ]]

>>> # Print the geometry type.
>>> print(go1.geom_type)
MultiPolygon
```

Example 2: Create an empty MultiPolygon

```
>>> from teradataml import Polygon, MultiPolygon

>>> poe = MultiPolygon()

>>> # Print the coordinates.
>>> print(poe.coords)
EMPTY
```

GeometryCollection

This function enables end user to create an object holding the multiple types of geometry objects. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional argument:

geoms: Specifies a list of different types of geometry objects.

- If no geometry objects are passed, an object for empty GeometryCollection is created.
- When geometry objects are passed, you must pass a list of the following types:
 - Point;
 - LineString;
 - Polygon;
 - MultiPoint;
 - MultiLineString;
 - MultiPolygon;
 - GeometryCollection;

- Mix of any of these seven types.

Example 1: Create a GeometryCollection object with all geometry types

```
>>> from teradataml import Point, LineString, Polygon, MultiPoint,
MultiLineString, MultiPolygon, GeometryCollection

>>> # Create Point objects.
>>> p1 = Point(1, 1)
>>> p2 = Point()
>>> p3 = Point(6, 3)
>>> p4 = Point(10, 5)
>>> p5 = Point()

>>> # Create LineString Objects.
>>> l1 = LineString([(1, 3), (3, 0), (0, 1)])
>>> l2 = LineString([(1.35, 3.6456), (3.6756, 0.23), (0.345, 1.756)])
>>> l3 = LineString()

>>> # Create Polygon Objects.
>>> po1 = Polygon([(0, 0, 0), (0, 0, 20), (0, 20, 0), (0, 20, 20),
(20, 0, 0), (20, 0, 20), (20, 20, 0), (20, 20, 20),
(0, 0, 0)])
>>> po2 = Polygon([(0, 0, 0), (0, 0, 20.435), (0, 20.435, 0),
(0, 20.435, 20.435), (20.435, 0, 0), (20.435, 0, 20.435),
(20.435, 20.435, 0), (20.435, 20.435, 20.435),
(0, 0, 0)])
>>> po3 = Polygon()

>>> # Create MultiPolygon Object.
>>> mpol = MultiPolygon([po1, po3, po2])

>>> # Create MultiLineString Object.
>>> mlin = MultiLineString([l1, l2, l3])

>>> # Create MultiPoint Object.
>>> mpnt = MultiPoint([p1, p2, p3, p4, p5])

>>> # Create a GeometryCollection object.
>>> gc1 = GeometryCollection([p1, p2, l1, l3, po2, po3, po1, mpol, mlin, mpnt])

>>> # Print the coordinates.
>>> print(gc1.coords)
[(1, 1), 'EMPTY', [(1, 3), (3, 0), (0, 1)], 'EMPTY', [(0, 0, 0), (0, 0,
```

```
20.435), (0, 20.435, 0), (0, 20.435, 20.435), (20.435, 0, 0), (20.435, 0,
20.435), (20.435, 20.435, 0), (20.435, 20.435, 20.435), (0, 0, 0)], 'EMPTY',
[(0, 0, 0), (0, 0, 20), (0, 20, 0), (0, 20, 20), (20, 0, 0), (20, 0, 20), (20,
20, 0), (20, 20, 20), (0, 0, 0)], [[(0, 0, 0), (0, 0, 20), (0, 20, 0), (0, 20,
20), (20, 0, 0), (20, 0, 20), (20, 20, 0), (20, 20, 20), (0, 0, 0)], 'EMPTY',
[(0, 0, 0), (0, 0, 20.435), (0, 20.435, 0), (0, 20.435, 20.435), (20.435, 0,
0), (20.435, 0, 20.435), (20.435, 20.435, 0), (20.435, 20.435, 20.435), (0,
0, 0)]]], [[(1, 3), (3, 0), (0, 1)], [(1.35, 3.6456), (3.6756, 0.23), (0.345,
1.756)], 'EMPTY'], [(1, 1), 'EMPTY', (6, 3), (10, 5), 'EMPTY']]
```

```
>>> # Print the geometry type.
>>> print(gc1.geom_type)
GeometryCollection
```

Example 2: Create an empty GeometryCollection

```
>>> gc2 = GeometryCollection()

>>> # Print the coordinates.
>>> print(gc2.coords)
EMPTY
```

GeoSequence

This function enables end user to create an object holding the LineString geometry objects with tracking information such as timestamps. It allows user to use the same in GeoDataFrame manipulation and processing using any Geospatial function.

Optional arguments:

- *coordinates*: Specifies the coordinates of a Point.
 - If coordinates are not passed, an object for empty line is created.
 - When coordinates are passed, you must pass a list of the following types:
 - Two-tuple of int or float;
 - Three-tuple of int or float;
 - Point geometry objects;
 - Mix of any of these three types.
- *timestamps*: Specifies a list of timestamp values for each coordinate in the following format: yyyy-mm-dd hh:mi:ss.ms.

The first timestamp value is associated with the first point, the second timestamp value is associated with the second point, and so forth.

Note:

Total number of timestamp values must match the total number of points in the GeoSequence.

- *link_ids*: Specifies the list of values for the ID of the link on the road network for a point in the GeoSequence.

The first link ID value is associated with the first point, the second link ID value is associated with the second point, and so forth.

Note:

Total number of link ID values must match the total number of points in the GeoSequence.

- *user_field_count*: Specifies the value that represents the number of user field elements for each point in the GeoSequence.

The default value 0 indicates that no user field elements appear after count in the character string.

- *user_fields*: Specifies the list of user field tuples that represents a value associated with a Point.

For example, certain tracking systems may associate velocity, direction, and acceleration values with each point.

Note:

- You must specify count groups of n user field values (where n is the number of points in the geosequence).
- The first group provides the first user field values for each point, the second group provides the second user field values for each point, and so forth.
- Each group can be formed using a tuple.

Example 1: Create a GeoSequence with 2D Points and no user fields

```
>>> from teradataml import Point, GeoSequence

>>> coordinates = [(1, 3), (3, 0), (0, 1)]

>>> timestamps = ["2008-03-17 10:34:03.53", "2008-03-17 10:38:25.21",
"2008-03-17 10:41:41.48"]

>>> link_ids = [1001, 1002, 1003]

>>> gs1 = GeoSequence(coordinates=coordinates,
timestamps=timestamps, link_ids=link_ids)
```

```
>>> gs1.coords
[(1, 3), (3, 0), (0, 1)]

>>> str(gs1)
'GeoSequence((1 3, 3 0, 0 1), (2008-03-17 10:34:03.53, 2008-03-17 10:38:25.21,
2008-03-17 10:41:41.48), (1001, 1002, 1003), (0))'
```

Example 2: Create a GeoSequence with 3D Points and 2 user fields

Note:

Coordinates can be provided as tuple of ints or floats, or Point objects.

```
>>> from teradataml import Point, GeoSequence

>>> p1 = (3, 0, 6)
>>> coordinates = [(1, 3, 6), p1, (6, 0, 1)]

>>> timestamps = ["2008-03-17 10:34:03.53", "2008-03-17 10:38:25.21",
"2008-03-17 10:41:41.48"]

>>> link_ids = [1001, 1002, 1003]

>>> user_fields = [(1, 2), (3, 4), (5, 6)]

>>> gs2 = GeoSequence(coordinates=coordinates, timestamps=timestamps,
link_ids=link_ids, user_field_count=2, user_fields=user_fields)

>>> gs2.coords
[(1, 3, 6), (3, 0, 6), (6, 0, 1)]

>>> str(gs2)
'GeoSequence((1 3 6, 3 0 6, 6 0 1), (2008-03-17 10:34:03.53, 2008-03-17
10:38:25.21, 2008-03-17 10:41:41.48), (1001, 1002, 1003), (2, 1, 2, 3, 4,
5, 6))'
```

Example 3: Create an empty GeoSequence

```
>>> from teradataml import Point, GeoSequence

>>> gs3 = GeoSequence()
```

```
>>> # Print the coordinates.
>>> print(gc3.coords)
EMPTY
```

teradataml GeoDataFrame

User can create a GeoDataFrame on a Teradata table and view containing geospatial data. This allows user to retrieve, manipulate, process, and analyze geospatial information stored in the table with help of the [GeoDataFrame Methods](#) and [GeoDataFrame Properties](#) of teradataml GeoDataFrame.

Note:

- If GeoDataFrame is created on a table that does not contain geometry data, then exception is raised.
- teradataml GeoDataFrame extends the teradataml DataFrame that allows user to access and use teradataml DataFrame properties and methods.
- Methods and Properties of teradataml GeoDataFrame will return teradataml GeoDataFrame, if the result contains geometry data. If result does not contain geometry data, then it returns teradataml DataFrame.

Access Geospatial Data on Vantage

teradataml provides GeoDataFrame a couple ways to access a table containing Geospatial data in Vantage and create a GeoDataFrame object from the same.

Create GeoDataFrame object from a table

Load the necessary teradataml modules first.

```
from teradataml import GeoDataFrame
```

To access a table containing Geospatial data, user can use GeoDataFrame constructor. For example:

```
geoDF = GeoDataFrame("sample_shapes")
```

Or user can create a GeoDataFrame using from_table() API. For example:

```
geoDF = GeoDataFrame.from_table("sample_shapes")
```

Create GeoDataFrame object from a query

Load the necessary teradataml modules first.

```
from teradataml import GeoDataFrame
```

To create a GeoDataFrame from query user must use `from_query()` API. For example:

```
geoDF = GeoDataFrame.from_query("select * from sample_shapes")
```

GeoDataFrame Properties

The following tables list properties of teradataml GeoDataFrame. For more details and examples, see [Teradata Package for Python Function Reference](#).

Generic Usage

```
from teradataml import GeoDataFrame
```

```
geodf = GeoDataFrame("sample_shapes")
```

```
geodf.name_of_the_property
```

Properties inherited from teradataml DataFrame

| Property | Purpose | Return |
|----------|--|---|
| columns | Get the column names of GeoDataFrame. | List containing column names |
| dtypes | Return a MetaData containing the column names and types. | MetaData containing the column names and Python types |
| iloc | Access a group of rows and columns by integer values or a boolean array. | teradataml GeoDataFrame |
| index | Return the index_label of the teradataml GeoDataFrame. | str or List of Strings (str) representing the index_label of the GeoDataFrame |
| loc | Access a group of rows and columns by labels or a boolean array. | teradataml GeoDataFrame |
| shape | Return a tuple representing the dimensionality of the GeoDataFrame. | Tuple representing the dimensionality of this GeoDataFrame |
| size | Return a value representing the number of elements in the GeoDataFrame. | Value representing the number of elements in the GeoDataFrame. |
| tdtypes | Get the teradataml GeoDataFrame metadata containing column names and corresponding teradatasqlalchemy types. | Metadata containing the column names and Teradata types |

Properties specific to Geospatial Data (All Geometry Types)

| Property | Purpose | Return |
|-------------|--|---|
| geometry | Return a GeoColumnExpression for a column containing geometry data. Note: This property is used to run any geospatial operation on GeoDataFrame, that is, any geospatial function ran on the geometry column referenced by this property. | teradataml GeoDataFrameColumn |
| boundary | Return the boundary of the Geometry value. | GeoDataFrame with result column containing Geometry values |
| centroid | Return the mathematical centroid of an ST_Polygon or ST_MultiPolygon value. | GeoDataFrame with result column containing Geometry values |
| convex_hull | Return the convex hull of the Geometry value. | GeoDataFrame with result column containing Geometry values |
| coord_dim | Return the coordinate dimension of a geometry. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the input geometry is 1D • 2, if the input geometry is 2D • 3, if the input geometry is 3D |
| dimension | Return the dimension of the Geometry type. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 0, for a 0-dimensional geometry • 1, for a 1D geometry • 2, for a 2D geometry • -1, if the input geometry is empty |
| geom_type | Return the Geometry type of the Geometry value. | GeoDataFrame Resultant column contains any of the following strings: <ul style="list-style-type: none"> • 'ST_Point' • 'ST_LineString' • 'ST_Polygon' • 'ST_MultiPoint' • 'ST_MultiLineString' • 'ST_MultiPolygon' • 'ST_GeomCollection' • 'GeoSequence' |
| is_3D | Test if a Geometry value has Z coordinate value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the Geometry contains Z coordinates |

| Property | Purpose | Return |
|-----------|---|---|
| | | <ul style="list-style-type: none"> 0, if the Geometry does not contain Z coordinates |
| is_empty | Test if a Geometry value corresponds to the empty set. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> 1, if the geometry is empty 0, if the geometry is not empty |
| is_simple | Test if a Geometry value has no anomalous geometric points, such as self intersection tangency. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> 1, if the geometry is simple, with no anomalous points 0, if the geometry is not simple |
| is_valid | Test if a Geometry value is well-formed. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> 1, if the geometry is valid 0, if the geometry is not valid |
| max_x | Return the maximum X coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| max_y | Return the maximum Y coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| max_z | Return the maximum Z coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| min_x | Return the minimum X coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| min_y | Return the minimum Y coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| min_z | Return the minimum Z coordinate of a Geometry value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| srid | Get the spatial reference system identifier of the Geometry value. | GeoDataFrame |

Properties for Point Geometry

| Property | Purpose | Return |
|----------|--|--------------|
| x | Get the X coordinate of an ST_Point value. | GeoDataFrame |
| y | Get the Y coordinate of an ST_Point value. | GeoDataFrame |
| z | Get the Z coordinate of an ST_Point value. | GeoDataFrame |

Properties for LineString Geometry

| Property | Purpose | Return |
|--------------|---|--|
| is_closed_3D | Test whether a 3D LineString or 3D MultiLineString is closed, taking into account the Z coordinates in the calculation. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the 3D LineString or 3D MultiLineString is closed. • 0, if the 3D LineString or 3D MultiLineString is not closed or is empty. |
| is_closed | Test if a Geometry type that represents an ST_LineString, GeoSequence, or ST_MultiLineString value is closed. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the ST_LineString, GeoSequence, or ST_LineString components of an ST_MultiLineString are closed. • 0, if the ST_LineString, GeoSequence, or ST_LineString components of an ST_MultiLineString are not closed, or if the input geometry is empty. |
| is_ring | Test if a Geometry type that represents an ST_LineString or a GeoSequence value is a ring. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the ST_LineString or GeoSequence value is simple (has no anomalous geometric points, such as self intersection tangency) and is closed (the start point of the ST_LineString value is equal to the end point) • 0, in all other cases. |

Properties for Polygon Geometry

| Property | Purpose | Return |
|----------|---|--|
| area | Return the area measurement of an ST_Polygon or ST_MultiPolygon. For ST_MultiPolygon, returns the sum of the area measurements of the component polygons. | GeoDataFrame |
| exterior | Get the exterior ring of a Geometry type that represents an ST_Polygon value. | GeoDataFrame with result column containing ST_LineString Geometry values |

| Property | Purpose | Return |
|-----------|--|--------------|
| perimeter | Return the boundary length of an ST_Polygon, or the sum of the boundary lengths of the component polygons of an ST_MultiPolygon. | GeoDataFrame |

GeoDataFrame Methods

The following tables list methods of teradataml GeoDataFrame. For more details and examples, see [Teradata Package for Python Function Reference](#).

Generic Usage: Single Geospatial Function Usage

```
# Create GeoDataFrame on a table containing Geospatial data.
geoDF = GeoDataFrame("sample_shapes")

# Assumption:
#   geoDF.geometry points to "geom_col1"

# Invoke "intersects" function on GeoDataFrame and pass the argument as
# ColumnExpression a.k.a. GeoDataFrameColumn object of a column 'geom_col2'.
geoDF.intersects(geoDF.geom_col2)
#   where,
#       geoDF.geom_col2 is a ColumnExpression, i.e., GeoDataFrameColumn object
#       for 'geom_col2'.

# Instead of passing GeoDataFrameColumn object for 'geom_col2', user can pass
# the column name as is.
geoDF.intersects("geom_col2")
```

Methods inherited from teradataml DataFrame

| Method | Method Signature | Purpose | Return |
|-------------------------|---|--|-------------------------|
| <code>__init__</code> | <code>__init__(self, table_name=None, index=True, index_label=None, query=None, materialize=False)</code> | Constructor for teradataml GeoDataFrame. | teradataml GeoDataFrame |
| <code>from_table</code> | <code>from_table(cls, table_name, index=True, index_label=None)</code> | Class method for creating a GeoDataFrame from a table or a view. | teradataml GeoDataFrame |
| <code>from_query</code> | <code>from_query(cls, query, index=True, index_label=None, materialize=False)</code> | Class method for creating a GeoDataFrame using a query. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|--------------------------|---|---|--|
| <code>__getattr__</code> | <code>__getattr__(self, name)</code> | Return an attribute of the GeoDataFrame. | Return the value of the named attribute of object (if found). |
| <code>__getitem__</code> | <code>__getitem__(self, key)</code> | Return a column from the GeoDataFrame or filter the GeoDataFrame using an expression. The following operators are supported: <ul style="list-style-type: none"> • comparison: <code>==</code>, <code>!=</code>, <code><</code>, <code><=</code>, <code>></code>, <code>>=</code> • boolean: <code>&</code> (and), <code> </code> (or), <code>~</code> (not), <code>^</code> (xor) Operands can be Python literals and instances of ColumnExpressions from the GeoDataFrame. | GeoDataFrame or ColumnExpression instance |
| <code>assign</code> | <code>assign(self, drop_columns = False, **kwargs)</code> | Assign new columns to a teradataml GeoDataFrame. | teradataml GeoDataFrame A new GeoDataFrame is returned with: <ul style="list-style-type: none"> • New columns in addition to all the existing columns if "drop_columns" is False. • Only new columns if "drop_columns" is True. • New columns in addition to group by columns, i.e., columns used for grouping, if assign() is run on GeoDataFrame.groupby(). |
| <code>concat</code> | <code>concat(self, other, join='OUTER', allow_duplicates=True, sort=False, ignore_index=False)</code> | Concatenates two teradataml GeoDataFrames along the index axis. | teradataml GeoDataFrame |
| <code>drop</code> | <code>drop(self, labels=None, axis=0, columns=None)</code> | Drop specified labels from rows or columns. | teradataml GeoDataFrame |
| <code>dropna</code> | <code>dropna(self, how='any', thresh=None, subset=None)</code> | Removes rows with null values. | teradataml GeoDataFrame |
| <code>filter</code> | <code>filter(self, items = None, like = None, regex = None, axis = 1, **kw)</code> | Filter rows or columns of GeoDataFrame according to labels in the specified index. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|------------|--|--|--|
| | | The filter is applied to the columns of the index when axis is set to 'rows'. | |
| get | get(self, key) | Retrieve required columns from GeoDataFrame using column name(s) as key. Return a new teradataml GeoDataFrame with requested columns only. | teradataml GeoDataFrame |
| get_values | get_values(self, num_rows = 99999) | Retrieve all values (only) present in a teradataml GeoDataFrame. Values are retrieved as per a numpy.ndarray representation of a teradataml GeoDataFrame. This format is equivalent to the get_values() representation of a Pandas GeoDataFrame. | Numpy.ndarray representation of a teradataml GeoDataFrame |
| groupby | groupby(self, columns_expr) | Apply GroupBy to one or more columns of a teradataml GeoDataFrame. The result will always behaves like calling groupby with as_index=False in pandas. | teradataml DataFrameGroupBy Object |
| head | head(self, n=display.max_rows) | Print the first n rows of the sorted teradataml GeoDataFrame. | teradataml GeoDataFrame |
| info | info(self, verbose=True, buf=None, max_cols=None, null_counts=False) | Print a summary of the GeoDataFrame. | None |
| join | join(self, other, on=None, how="left", lsuffix=None, rsuffix=None) | Join two different teradataml GeoDataFrames together based on column comparisons specified in argument 'on' and type of join is specified in the argument 'how'. | teradataml GeoDataFrame |
| keys | keys(self) | Get the column names of a GeoDataFrame. | List containing the column names |
| merge | merge(self, right, on=None, how="inner", left_on=None, right_on=None, use_index=False, lsuffix=None, rsuffix=None) | Merge two teradataml GeoDataFrames together. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|------------|--|--|---|
| sample | sample(self, n = None, frac = None, replace = False, randomize = False, case_when_then = None, case_else = None) | Allow to sample few rows from GeoDataFrame directly or based on conditions. Create a new column 'sampleid' which has a unique id for each sample sampled, it helps to uniquely identify each sample. | teradataml GeoDataFrame |
| select | select(self, select_expression) | Select required columns from GeoDataFrame using an expression. Return a new teradataml GeoDataFrame with selected columns only. | teradataml GeoDataFrame /DataFrame |
| set_index | set_index(self, keys, drop = True, append = False) | Assign one or more existing columns as the new index to a teradataml GeoDataFrame. | teradataml GeoDataFrame |
| show_query | show_query(self, full_query = False) | Return underlying SQL for the teradataml GeoDataFrame. It is the same SQL that is used to view the data for a teradataml GeoDataFrame. | String |
| sort | sort(self, columns, ascending=True) | Get sorted data by one or more columns in either ascending or descending order for a GeoDataFrame. | teradataml GeoDataFrame |
| sort_index | sort_index(self, axis=0, ascending=True, kind='quicksort') | Gets sorted object by labels (along an axis) in either ascending or descending order for a teradataml GeoDataFrame. | teradataml GeoDataFrame |
| squeeze | squeeze(self, axis=None) | Squeeze one-dimensional axis objects into scalars. <ul style="list-style-type: none"> • teradataml GeoDataFrames with a single element are squeezed to a scalar. • teradataml GeoDataFrames with a single column are squeezed to a Series. • Otherwise the object is unchanged. | teradataml GeoDataFrame, teradataml Series, or scalar, the projection after squeezing 'axis' or all the axes. |
| tail | tail(self, n=display.max_rows) | Print the last n rows of the sorted teradataml GeoDataFrame. | teradataml GeoDataFrame |
| to_csv | to_csv | Get the data exported to CSV file. | None |
| to_pandas | to_pandas(self, index_column=None, num_rows=99999, all_ | Return a Pandas GeoDataFrame for the corresponding teradataml GeoDataFrame Object. | When "catch_errors_warnings" is set to True and if protocol used for |

| Method | Method Signature | Purpose | Return |
|--------|--|--|--|
| | rows=False, fastexport=False, catch_ errors_warnings=False, **kwargs) | | data transfer is fastexport, then the function returns a tuple containing: <ul style="list-style-type: none"> • Pandas GeoDataFrame. • Errors, if any, thrown by fastexport in a list of strings. • Warnings, if any, thrown by fastexport in a list of strings. Only Pandas GeoDataFrame otherwise. |
| to_sql | to_sql(self, table_ name, if_exists='fail', primary_index=None, temporary=False, schema_name=None, types = None, primary_ time_index_name=None, timecode_column=None, timebucket_ duration=None, timezero_date=None, columns_list=None, sequence_ column=None, seq_max=None, set_table=False) | Write records stored in a teradataml GeoDataFrame to Teradata Vantage. | None |

Methods specific to Geospatial Data (All Geometry Types)

| Method | Method Signature | Purpose | Return |
|----------|--------------------------------|--|--|
| buffer | buffer(self, distance) | Return all points whose distance from a Geometry value is less than or equal to a specified distance. | teradataml GeoDataFrame |
| contains | contains(self, geom_column) | Test if a Geometry value spatially contains another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the input geometry contains other geometry • 0, if the input geometry does not contain other geometry |
| crosses | crosses(self, geom_column) | Test if a Geometry value spatially crosses another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries cross |

| Method | Method Signature | Purpose | Return |
|--------------|---------------------------------|--|---|
| | | | <ul style="list-style-type: none"> • 0, if the geometries do not cross |
| difference | difference(self, geom_column) | Return a Geometry value that represents the point set difference of two Geometry values. | teradataml GeoDataFrame |
| disjoint | disjoint(self, geom_column) | Test if a Geometry value is spatially disjoint from another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries are spatially disjoint • 0, if the geometries are not spatially disjoint |
| distance | distance(self, geom_column) | Return the distance between two Geometry values. | teradataml GeoDataFrame |
| distance_3D | distance_3D(self, geom_column) | Return the distance between two Geometry values. Function considers Z coordinate values, if they exist in the geometries passed to it. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • a NULL, if the Geometry is an empty set. • 0, if the two geometries intersect. • a float in all other cases. The distance units are those of the two input geometries. |
| envelope | envelope(self) | Return the bounding rectangle for the Geometry value. | GeoDataFrame with result column containing Geometry values |
| geom_equals | geom_equals(self, geom_column) | Test if a Geometry value is spatially equal to another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries are spatially equal • 0, if the geometries are not spatially equal |
| intersection | intersection(self, geom_column) | Return a Geometry type where the value represents the point set intersection of two Geometry values. | teradataml GeoDataFrame |
| intersects | intersects(self, geom_column) | Test if a Geometry value spatially intersects another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries intersect • 0, if the geometries do not intersect |
| make_2D | make_2D(self, validate=False) | Converts a 3D Geometry value to a 2D Geometry value, and | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|----------------|-------------------------------------|---|---|
| | | optionally validates that the 2D geometry is well-formed. This method strips the z coordinate from 3D geometries. | |
| mbb | mbb(self) | Return the 3D minimum bounding box (MBB) that encloses a 3D Geometry. | teradataml GeoDataFrame |
| mbr | mbr(self) | Return the minimum bounding rectangle (MBR) of a Geometry value. | teradataml GeoDataFrame |
| overlaps | overlaps(self, geom_column) | Test if a Geometry value spatially overlaps another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries overlap • 0, if the geometries do not overlap |
| relates | relates(self, geom_column, amatrix) | Test if a Geometry value is spatially related to another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the spatial relationship between the geometries corresponds to one of the acceptable values represented by "amatrix" • 0, if the spatial relationship between the geometries does not correspond to one of the acceptable values represented by "amatrix" |
| set_srid | set_srid(self, srid) | Set the spatial reference system identifier of the Geometry value. | GeoDataFrame Resultant column contains the Geometry with the spatial reference system identifier set to the specified spatial reference system. |
| simplify | simplify(self, tolerance) | Simplify a geometry by removing points that would fall within a specified distance tolerance. | teradataml GeoDataFrame |
| sym_difference | sym_difference(self, geom_column) | Return a Geometry value that represents the point set symmetric difference of two Geometry values. | teradataml GeoDataFrame |
| to_binary | to_binary(self) | Return the well-known binary (WKB) representation of a Geometry value. | GeoDataFrame with result column containing BLOB value |

| Method | Method Signature | Purpose | Return |
|-----------------|--|---|---|
| to_text | to_text(self) | Return the well-known text (WKT) representation of a Geometry value. | GeoDataFrame with result column containing CLOB value |
| touches | touches(self, geom_column) | Test if a Geometry value spatially touches another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries touch • 0, if the geometries do not touch |
| transform | transform(self, to_wkt_srs, from_wkt_srs, to_srsid=-12345) | Return a Geometry value transformed to the specified spatial reference system. | teradataml GeoDataFrame |
| union | union(self, geom_column) | Return a Geometry value that represents the point set union of two Geometry values. | teradataml GeoDataFrame |
| within | within(self, geom_column) | Test if a Geometry value is spatially within another Geometry value. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometry is within other geometry • 0, if the geometry is not within other geometry |
| wkb_geom_to_sql | wkb_geom_to_sql(self, column) | Return a specified Geometry value. | teradataml GeoDataFrame |
| wkt_geom_to_sql | wkt_geom_to_sql(self, column) | Return a specified Geometry value. | teradataml GeoDataFrame |

Methods for Point Geometry

| Method | Method Signature | Purpose | Return |
|--------------------|---|--|-------------------------|
| spherical_buffer | spherical_buffer(self, distance, radius=6371000.0) | Return an MBR that represents the minimum and maximum latitude and longitude of all points within a given distance from a point. The earth is modeled as a sphere. | teradataml GeoDataFrame |
| spherical_distance | spherical_distance(self, geom_column) | Return the spherical distance between two spherical coordinates on the planet using the Haversine Formula. Both coordinates must be specified as ST_Point values. | teradataml GeoDataFrame |
| spheroidal_buffer | spheroidal_buffer(self, distance, semimajor=6378137.0, invflattening=298.257223563) | Return an MBR that represents the minimum and maximum latitude and longitude of all points within a given distance from the point. The earth is modeled as a spheroid. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|---------------------|--|--|-------------------------|
| spheroidal_distance | spheroidal_distance(self, geom_column, semimajor=6378137.0, invflattening=298.257223563) | Return the distance, in meters, between two spherical coordinates. | teradataml GeoDataFrame |
| set_x | set_x(self, xcoord) | Set the X coordinate of an ST_Point value. | teradataml GeoDataFrame |
| set_y | set_y(self, ycoord) | Set the Y coordinate of an ST_Point value. | teradataml GeoDataFrame |
| set_z | set_z(self, zcoord) | Set the Z coordinate of an ST_Point value. | teradataml GeoDataFrame |

Methods for LineString Geometry

| Method | Method Signature | Purpose | Return |
|------------------------|--|---|--|
| end_point | end_point(self) | Return the end point of an ST_LineString or GeoSequence value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| length | length(self) | Return the length measurement of a Geometry type that represents an ST_LineString, GeoSequence, or ST_MultiLineString value. | teradataml GeoDataFrame |
| length_3D | length_3D(self) | Return the length of a 3D LineString or MultiLineString, taking into account the Z coordinates in the calculation. | GeoDataFrame Resultant column contains a 0, if the LineString or MultiLineString is an empty set. |
| line_interpolate_point | line_interpolate_point(self, proportion) | Return a point interpolated along a Geometry type that represents an ST_LineString or GeoSequence value, given a proportional distance along that line. | GeoDataFrame with result column containing ST_Point Geometry values |
| num_points | num_points(self) | Return the number of points in a Geometry type that represents an ST_LineString or GeoSequence value. | GeoDataFrame Resultant column contains a NULL, if the Geometry is an empty set. |
| point | point(self, position) | Return the specified point from a Geometry type that represents an ST_LineString or GeoSequence value. | teradataml GeoDataFrame |
| start_point | start_point(self) | Return the start point of an ST_LineString or GeoSequence value. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|--------|------------------|---|--------|
| | | Note: This method returns a point having a Z coordinate if the input geometry is three-dimensional. | |

Methods for Polygon Geometry

| Method | Method Signature | Purpose | Return |
|-------------------|---------------------------|--|--|
| interiors | interiors(self, position) | Return the specified interior ring of a Geometry type that represents an ST_Polygon value. | GeoDataFrame with result column containing ST_LineString Geometry values |
| num_interior_ring | num_interior_ring(self) | Return the number of interior rings of a Geometry type that represents an ST_Polygon value. | teradataml GeoDataFrame |
| point_on_surface | point_on_surface(self) | Return a point that is guaranteed to spatially intersect an ST_Polygon, or at least one of the component polygons of an ST_MultiPolygon. | teradataml GeoDataFrame |

Methods for GeometryCollection Geometry

| Method | Method Signature | Purpose | Return |
|----------------|--------------------------------|---|-------------------------|
| geom_component | geom_component(self, position) | Return the geometry of one component member of a composite geometry type (ST_GeomCollection, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon). The element to be returned is specified by position with the collection. | teradataml GeoDataFrame |
| num_geometry | num_geometry(self) | Return the number of distinct geometries that constitute a composite geometry type (ST_GeomCollection, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon). | teradataml GeoDataFrame |

Methods for GeoSequence Geometry

| Method | Method Signature | Purpose | Return |
|--------|--|---|-------------------------|
| clip | clip(self, start_timestamp, end_timestamp) | Return a GeoSequence type containing the subset of points and associated data that lie between the two timestamp input arguments. | teradataml GeoDataFrame |

| Method | Method Signature | Purpose | Return |
|----------------------|---|---|-------------------------|
| get_final_timestamp | get_final_timestamp(self) | Return the TimeStamp of the last point of a GeoSequence. | teradataml GeoDataFrame |
| get_init_timestamp | get_init_timestamp(self) | Return the TimeStamp of the first point of a GeoSequence. | teradataml GeoDataFrame |
| get_link | get_link(self, index) | Get the link ID of a specified point in a GeoSequence. | teradataml GeoDataFrame |
| get_user_field | get_user_field(self, field_index, index) | Return the user field specified by "field_index" for the point specified by "index" for a GeoSequence type. | teradataml GeoDataFrame |
| get_user_field_count | get_user_field_count(self) | Return the number of user fields associated with each point of a GeoSequence. | teradataml GeoDataFrame |
| point_heading | point_heading(self, index) | Return the heading for the specified point of a GeoSequence. The value is calculated as the angle (in degrees) between a vertical line and the line segment from the specified point to the next point clockwise from the North. | teradataml GeoDataFrame |
| set_link | set_link(self, index, link_id) | Set the link ID of a specified point in a GeoSequence. | teradataml GeoDataFrame |
| speed | speed(self, index=None, begin_index=None, end_index=None) | Return the approximate speed at a specified point (speed(index INTEGER)) or between two points (speed(iBegin INTEGER, iEnd INTEGER)) for a GeoSequence type. | teradataml GeoDataFrame |

Filtering Functions and Methods

| Method | Method Signature | Purpose | Return |
|----------------|-----------------------------------|--|--|
| intersects_mbb | intersects_mbb(self, geom_column) | Test whether a 3D geometry spatially intersects a specified MBB. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the input 3D geometry intersects the MBB argument. • 0, if the input 3D geometry does not intersect the MBB argument. • Return an error if the input geometry is not 3D (does not have a z coordinate). |
| mbb_filter | mbb_filter(self, geom_column) | Test whether the MBBs of two 3D geometries spatially intersect. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the MBB of the input 3D geometry intersects the MBB of the geometry passed to the method. |

| Method | Method Signature | Purpose | Return |
|------------|-------------------------------|---|--|
| | | | <ul style="list-style-type: none"> • 0, if the MBB of the input 3D geometry does not intersect the MBB of the geometry passed to the method. • Return an error if either geometry is not 3D (does not have a z coordinate). |
| mbr_filter | mbr_filter(self, geom_column) | Test whether the MBRs of two 2D geometries spatially intersect. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the MBR of the input geometry intersects the MBR of the geometry passed to the method. • 0, if the MBR of the input geometry does not intersect the MBR of the geometry passed to the method. |
| within_mbb | within_mbb(self, geom_column) | Test whether a 3D geometry is spatially within the bounds of a specified MBB. | GeoDataFrame Resultant column contains: <ul style="list-style-type: none"> • 1, if the input 3D geometry is spatially within MBB argument. • 0, if the input 3D geometry is not spatially within the MBB argument. • Return an error if the input geometry is not 3D (does not have a z coordinate). |

teradataml GeoDataFrameColumn

teradataml GeoDataFrameColumn represents a column of teradataml GeoDataFrame. User can manipulate, process, and analyze the data using the [GeoDataFrameColumn Properties](#) and [GeoDataFrameColumn Methods](#) of teradataml GeoDataFrameColumn. This provides more flexibility to the user on processing the data.

Access Individual Geometry Columns

GeoDataFrame object can have non-geometry columns and at least one geometry column. Similar to teradataml DataFrame, teradataml GeoDataFrame allows user to access individual columns in a using `df_object.<column_name>`.

The following are two ways to access individual geometry columns in a GeoDataFrame.

Access columns similar to teradataml DataFrame column access

A GeoDataFrame can contain multiple columns with Geospatial data.

In the following example, GeoDataFrame contains two columns with geometry type 'geom_col1' and 'geom_col1'. The following shows how to access geometry columns.

- Create GeoDataFrame on a table containing Geospatial data.

```
geoDF = GeoDataFrame("sample_shapes")
```

- Access column 'geom_col1'.

Note:

The following function returns an object of class GeoDataFrameColumn.

```
geoDF.geom_col1
```

- Access column 'geom_col2'.

```
geoDF.geom_col2
```

Access using 'geometry' property

The most generic way to access the geometry data in any GeoDataFrame is using the 'geometry' property of a GeoDataFrame. This property always points to a column containing geometry data. Regardless of the column name, user can access the column with this property.

Note:

- If GeoDataFrame contains multiple columns with Geospatial data, then 'geometry' property points to only one of the columns.
When GeoDataFrame is created, 'geometry' property automatically refers to the first geometry column encountered in the GeoDataFrame.
 - User is allowed to set the 'geometry' property to point to another column of teradatasqlalchemy type Geometry in the same GeoDataFrame.
-

A GeoDataFrame can contain multiple columns with Geospatial data. In the following example, there are two columns with geometry type 'geom_col1' and 'geom_col1'. The following shows how to access geometry column, using 'geometry' property.

- Create GeoDataFrame on a table containing Geospatial data.

```
geoDF = GeoDataFrame("sample_shapes")
```

- Access the geometry data in GeoDataFrame using 'geometry' property.
-

Note:

This returns an object of class GeoDataFrameColumn for one of the column 'geom_col1' or 'geom_col2'.

```
geoDF.geometry
```


- To identify the name of the column the 'geometry' property points to, use 'name' property of GeoDataFrameColumn.

```
geoDF.geometry.name
```

If it returns 'geom_col1', that means geoDF.geometry is same as geoDF.geom_col1.

- User can change the geometry column to point to a different column.

For example, run the following command to point to 'geom_col2'.

```
geoDF.geometry = "geom_col2"
```

GeoDataFrameColumn Properties and Methods Specific to Geospatial Data

teradataml provides various methods and properties for GeoDataFrameColumn.

Note:

- Executing geospatial specific methods and properties always return a teradataml GeoDataFrameColumn object.
 - Combine these operations with either GeoDataFrame.assign() or GeoDataFrame[], that is, slice filtering to get the desired result.
-

Use Case 1: Run a Single Geospatial Function

To run a single Geospatial function, first create GeoDataFrame on a table containing Geospatial data.

```
geoDF = GeoDataFrame("sample_shapes")
```

Assume geoDF.geometry points to "geom_col1".

Note:

This approach requires user to use GeoDataFrame.assign() function.

Syntax 1: Using 'geometry' property to invoke the function

This syntax uses 'geometry' property of GeoDataFrame to invoke the function.

```
geoDF.assign(intersect_result = geoDF.geometry.intersects(geoDF.geom_col2))
```

Syntax 2: Using actual ColumnExpression of a column to invoke the function

This syntax uses actual ColumnExpression, also known as GeoDataFrameColumn object of a column 'geom_col1'.

```
geoDF.assign(intersect_result = geoDF.geom_col1.intersects(geoDF.geom_col2))
```

Syntax 3: Passing the column name instead of the GeoDataFrameColumn object as value

This syntax uses the column name as is, instead of passing GeoDataFrameColumn object for 'geom_col2'.

```
geoDF.assign(intersect_result = geoDF.geometry.intersects("geom_col2"))
```

Use Case 2: Run Multiple Geospatial Functions

There are scenarios when user needs to run multiple functions on single column or multiple columns. For example, there are two columns containing Geospatial data, 'geom_col1' and 'geom_col2', and you want to:

- Check if geometry values in 'geom_col1' column intersects with geometry values from column 'geom_col2' or not. (Use: intersects)
- Check if geometry values in 'geom_col1' column are spatially equal to geometry values from column 'geom_col2' or not. (Use: geom_Equals)
- Calculate point set symmetric difference between geometry values in columns 'geom_col1' and 'geom_col2'. (Use: sym_difference)
- Calculate point set difference between geometry values in columns 'geom_col1' and 'geom_col2'. (Use: difference)

With GeoDataFrame, several steps are required to get the desired result. With GeoDataFrame execution, each function is ran individually and then the results of all are joined to get the final result.

With teradataml GeoDataFrameColumn, this can be achieved using a single step to run multiple Geospatial functions, see Approach 2.

Approach 1: Run function individually on GeoDataFrameColumn and pass the results to 'assign()'

```
# Create GeoDataFrame on a table containing Geospatial data.
geoDF = GeoDataFrame("sample_shapes")
```

```
geo_intersect = geoDF.geom_col1.intersects(geoDF.geom_col2)
```

```
geo_equals = geoDF.geom_col1.equals("geom_col2")
```

```

geo_setdiff = geoDF.geom_col1.difference(geoDF.geom_col2)

geo_symdiff = geoDF.geom_col1.symmetric_difference(geoDF.geom_col2)

geoDF.assign(intersect_result = geo_intersect,
             equals_result = geo_equals,
             set_diff_result = geo_setdiff,
             symmetric_set_result = geo_symdiff)

```

Approach 2: Run multiple functions in one step

```

# Create GeoDataFrame on a table containing Geospatial data.
geoDF = GeoDataFrame("sample_shapes")

# ColumnExpression a.k.a GeoDataFrameColumn object.
geocol1 = geoDF.geom_col1

# This can be achieved using GeoDataFrame.assign() and
GeoDataFrameColumn methods.
geoDF.assign(intersect_result = geocol1.intersects(geoDF.geom_col2),
             equals_result = geocol1.equals(geoDF.geom_col2),
             set_diff_result = geocol1.difference(geoDF.geom_col2),
             symmetric_set_result
= geocol1.symmetric_difference(geoDF.geom_col2),
             )

```

Note:

- User can use 'geoDF.geometry' property to invoke GeoDataFrameColumn method instead of 'geoDF.geom_col1', if geometry points to 'geom_col1'.
- User can pass column name as string, that is, 'geom_col2', instead of GeoDataFrameColumn object of the column.
- In Approach 2, a variable 'geocol1' is created. Creation of the variable is optional. User can directly use 'geoDF.geom_col1' instead of 'geo_col1' in assign() function call.

Use Case 3: Filtering using Geospatial Functions

User can run Geospatial functions to filter the data.

In the following example, there are two tables: 'sample_cities' and 'sample_streets'. Both contain a geometry column 'cityShape' and 'streetShape' respectively. The following steps return the street name and city name, if 'cityShape' contains a 'streetShape'.

```
# Create GeoDataFrame on tables containing Geospatial data.
cities = GeoDataFrame("sample_cities")
streets = GeoDataFrame("sample_streets")

# First we must perform cross join between the two GeoDataFrames.
city_streets = cities.join(streets, how="cross")

# Apply slice filter and invoke Geospatial function contains in the filter and
use the same as predicate.
city_streets[city_streets.cityShape.contains(city_streets.streetShape) ==
1].select("streetName", "cityName")

# Slice filtering with Geometry Type object passed for comparison.
# Create a LineString object.
line_object = LineString([(2,2), (3,2), (4,1)])
streets[streets.streetShape == line_object]
```

Note:

- User can use 'city_streets.geometry' property to invoke GeoDataFrameColumn method instead of 'city_streets.cityShape', if geometry points to 'cityShape'.
- User can pass column name as string, that is, 'streetShape' instead of GeoDataFrameColumn object of the column.

GeoDataFrameColumn Properties

The following tables list properties of teradataml GeoDataFrameColumn. For more details and examples, see [Teradata Package for Python Function Reference](#).

Properties inherited from teradataml DataFrameColumn

| Property | Purpose | Return |
|----------|--|-------------------------|
| name | Get the name of the GeoDataFrameColumn. | string |
| type | Get the Teradata type of the GeoDataFrameColumn. | teradatasqlalchemy type |

Properties specific to Geospatial Data (All Geometry Types)

| Property | Purpose | Return |
|----------|--|--|
| boundary | Return the boundary of the Geometry value. | GeoDataFrameColumn with result column containing Geometry values |

| Property | Purpose | Return |
|-------------|---|---|
| centroid | Return the mathematical centroid of an ST_Polygon or ST_MultiPolygon value. | GeoDataFrameColumn with result column containing Geometry values |
| convex_hull | Return the convex hull of the Geometry value. | GeoDataFrameColumn with result column containing Geometry values |
| coord_dim | Return the coordinate dimension of a geometry. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the input geometry is 1D • 2, if the input geometry is 2D • 3, if the input geometry is 3D |
| dimension | Return the dimension of the Geometry type. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 0, for a 0-dimensional geometry • 1, for a 1D geometry • 2, for a 2D geometry • -1, if the input geometry is empty |
| geom_type | Return the Geometry type of the Geometry value. | GeoDataFrameColumn Resultant column contains any of the following strings: <ul style="list-style-type: none"> • 'ST_Point' • 'ST_LineString' • 'ST_Polygon' • 'ST_MultiPoint' • 'ST_MultiLineString' • 'ST_MultiPolygon' • 'ST_GeomCollection' • 'GeoSequence' |
| is_3D | Test if a Geometry value has Z coordinate value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the Geometry contains Z coordinates • 0, if the Geometry does not contain Z coordinates |
| is_empty | Test if a Geometry value corresponds to the empty set. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometry is empty • 0, if the geometry is not empty |
| is_simple | Test if a Geometry value has no anomalous geometric points, such as self intersection tangency. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometry is simple, with no anomalous points • 0, if the geometry is not simple |
| is_valid | Test if a Geometry value is well-formed. | GeoDataFrameColumn Resultant column contains: |

| Property | Purpose | Return |
|----------|--|--|
| | | <ul style="list-style-type: none"> • 1, if the geometry is valid • 0, if the geometry is not valid |
| max_x | Return the maximum X coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| max_y | Return the maximum Y coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| max_z | Return the maximum Z coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| min_x | Return the minimum X coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| min_y | Return the minimum Y coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| min_z | Return the minimum Z coordinate of a Geometry value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| srid | Get the spatial reference system identifier of the Geometry value. | GeoDataFrameColumn |

Properties for Point Geometry

| Property | Purpose | Return |
|----------|--|--------------------|
| x | Get the X coordinate of an ST_Point value. | GeoDataFrameColumn |
| y | Get the Y coordinate of an ST_Point value. | GeoDataFrameColumn |
| z | Get the Z coordinate of an ST_Point value. | GeoDataFrameColumn |

Properties for LineString Geometry

| Property | Purpose | Return |
|--------------|--|---|
| is_closed_3D | Tests whether a 3D LineString or 3D MultiLineString is closed, taking into account the Z coordinates in the calculation. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the 3D LineString or 3D MultiLineString is closed. • 0, if the 3D LineString or 3D MultiLineString is not closed or is empty. |

| Property | Purpose | Return |
|-----------|--|--|
| is_closed | Tests if a Geometry type that represents an ST_LineString, GeoSequence, or ST_MultiLineString value is closed. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the ST_LineString, GeoSequence, or ST_LineString components of an ST_MultiLineString are closed. • 0, if the ST_LineString, GeoSequence, or ST_LineString components of an ST_MultiLineString are not closed, or if the input geometry is empty. |
| is_ring | Tests if a Geometry type that represents an ST_LineString or a GeoSequence value is a ring. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the ST_LineString or GeoSequence value is simple (has no anomalous geometric points, such as self intersection tangency) and is closed (the start point of the ST_LineString value is equal to the end point) • 0, in all other cases. |

Properties for Polygon Geometry

| Property | Purpose | Return |
|-----------|---|--|
| area | Return the area measurement of an ST_Polygon or ST_MultiPolygon. For ST_MultiPolygon, returns the sum of the area measurements of the component polygons. | GeoDataFrameColumn |
| exterior | Get the exterior ring of a Geometry type that represents an ST_Polygon value. | GeoDataFrameColumn with result column containing ST_LineString Geometry values |
| perimeter | Return the boundary length of an ST_Polygon, or the sum of the boundary lengths of the component polygons of an ST_MultiPolygon. | GeoDataFrameColumn |

GeoDataFrameColumn Methods

The following tables list methods of teradataml GeoDataFrameColumn. For more details and examples, see [Teradata Package for Python Function Reference](#).

Methods specific to Geospatial Data (All Geometry Types)

| Method | Method Signature | Purpose | Return |
|----------|-----------------------------|---|--|
| buffer | buffer(self, distance) | Return all points whose distance from a Geometry value is less than or equal to a specified distance. | teradataml GeoDataFrameColumn |
| contains | contains(self, geom_column) | Test if a Geometry value spatially contains another Geometry value. | GeoDataFrameColumn Resultant column contains: |

| Method | Method Signature | Purpose | Return |
|--------------|---------------------------------|--|---|
| | | | <ul style="list-style-type: none"> • 1, if the input geometry contains other geometry • 0, if the input geometry does not contain other geometry |
| crosses | crosses(self, geom_column) | Test if a Geometry value spatially crosses another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries cross • 0, if the geometries do not cross |
| difference | difference(self, geom_column) | Return a Geometry value that represents the point set difference of two Geometry values. | teradataml GeoDataFrameColumn |
| disjoint | disjoint(self, geom_column) | Test if a Geometry value is spatially disjoint from another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries are spatially disjoint • 0, if the geometries are not spatially disjoint |
| distance | distance(self, geom_column) | Return the distance between two Geometry values. | teradataml GeoDataFrameColumn |
| distance_3D | distance_3D(self, geom_column) | Return the distance between two Geometry values. Function considers Z coordinate values, if they exist in the geometries passed to it. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • a NULL, if the Geometry is an empty set. • 0, if the two geometries intersect. • a float in all other cases. The distance units are those of the two input geometries. |
| envelope | envelope(self) | Return the bounding rectangle for the Geometry value. | GeoDataFrameColumn with result column containing Geometry values |
| geom_equals | geom_equals(self, geom_column) | Test if a Geometry value is spatially equal to another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries are spatially equal • 0, if the geometries are not spatially equal |
| intersection | intersection(self, geom_column) | Return a Geometry type where the value represents the point set intersection of two Geometry values. | teradataml GeoDataFrameColumn |

| Method | Method Signature | Purpose | Return |
|------------|-------------------------------------|--|---|
| intersects | intersects(self, geom_column) | Test if a Geometry value spatially intersects another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries intersect • 0, if the geometries do not intersect |
| make_2D | make_2D(self, validate=False) | Converts a 3D Geometry value to a 2D Geometry value, and optionally validates that the 2D geometry is well-formed. This method strips the z coordinate from 3D geometries. | teradataml GeoDataFrameColumn |
| mbb | mbb(self) | Return the 3D minimum bounding box (MBB) that encloses a 3D Geometry. | teradataml GeoDataFrameColumn |
| mbr | mbr(self) | Return the minimum bounding rectangle (MBR) of a Geometry value. | teradataml GeoDataFrameColumn |
| overlaps | overlaps(self, geom_column) | Test if a Geometry value spatially overlaps another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries overlap • 0, if the geometries do not overlap |
| relates | relates(self, geom_column, amatrix) | Test if a Geometry value is spatially related to another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the spatial relationship between the geometries corresponds to one of the acceptable values represented by "amatrix" • 0, if the spatial relationship between the geometries does not correspond to one of the acceptable values represented by "amatrix" |
| set_srid | set_srid(self, srid) | Set the spatial reference system identifier of the Geometry value. | GeoDataFrameColumn Resultant column contains the Geometry with the spatial reference system identifier set to the specified spatial reference system. |
| simplify | simplify(self, tolerance) | Simplify a geometry by removing points that would | teradataml GeoDataFrameColumn |

| Method | Method Signature | Purpose | Return |
|-----------------|--|--|---|
| | | fall within a specified distance tolerance. | |
| sym_difference | sym_difference(self, geom_column) | Return a Geometry value that represents the point set symmetric difference of two Geometry values. | teradataml GeoDataFrameColumn |
| to_binary | to_binary(self) | Return the well-known binary (WKB) representation of a Geometry value. | GeoDataFrameColumn with result column containing BLOB value |
| to_text | to_text(self) | Return the well-known text (WKT) representation of a Geometry value. | GeoDataFrameColumn with result column containing CLOB value |
| touches | touches(self, geom_column) | Test if a Geometry value spatially touches another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometries touch • 0, if the geometries do not touch |
| transform | transform(self, to_wkt_srs, from_wkt_srs, to_srsid=-12345) | Return a Geometry value transformed to the specified spatial reference system. | teradataml GeoDataFrameColumn |
| union | union(self, geom_column) | Return a Geometry value that represents the point set union of two Geometry values. | teradataml GeoDataFrameColumn |
| within | within(self, geom_column) | Test if a Geometry value is spatially within another Geometry value. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the geometry is within other geometry • 0, if the geometry is not within other geometry |
| wkb_geom_to_sql | wkb_geom_to_sql(self, column) | Return a specified Geometry value. | teradataml GeoDataFrameColumn |
| wkt_geom_to_sql | wkt_geom_to_sql(self, column) | Return a specified Geometry value. | teradataml GeoDataFrameColumn |

Methods for Point Geometry

| Method | Method Signature | Purpose | Return |
|------------------|--|--|-------------------------------|
| spherical_buffer | spherical_buffer(self, distance, radius=6371000.0) | Return an MBR that represents the minimum and maximum latitude and | teradataml GeoDataFrameColumn |

| Method | Method Signature | Purpose | Return |
|---------------------|--|--|----------------------------------|
| | | longitude of all points within a given distance from a point. The earth is modeled as a sphere. | |
| spherical_distance | spherical_distance(self, geom_column) | Return the spherical distance between two spherical coordinates on the planet using the Haversine Formula. Both coordinates must be specified as ST_Point values. | teradataml GeoDataFrameColumn |
| spheroidal_buffer | spheroidal_buffer(self, distance, semimajor=6378137.0, invflattening=298.257223563) | Return an MBR that represents the minimum and maximum latitude and longitude of all points within a given distance from the point. The earth is modeled as a spheroid. | teradataml GeoDataFrameColumn |
| spheroidal_distance | spheroidal_distance(self, geom_column, semimajor=6378137.0, invflattening=298.257223563) | Return the distance, in meters, between two spherical coordinates. | teradataml GeoDataFrameColumn |
| set_x | set_x(self, xcoord) | Set the X coordinate of an ST_Point value. | teradataml GeoDataFrameColumn |
| set_y | set_y(self, ycoord) | Set the Y coordinate of an ST_Point value. | teradataml GeoDataFrameColumn |
| set_z | set_z(self, zcoord) | Set the Z coordinate of an ST_Point value. | teradataml GeoDataFrameColumn |

Methods for LineString Geometry

| Method | Method Signature | Purpose | Return |
|-----------|------------------|--|--|
| end_point | end_point(self) | Return the end point of an ST_LineString or GeoSequence value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| length | length(self) | Return the length measurement of a Geometry type that represents an ST_LineString, GeoSequence, or ST_MultiLineString value. | teradataml GeoDataFrameColumn |
| length_3D | length_3D(self) | Return the length of a 3D LineString or MultiLineString, | GeoDataFrameColumn |

| Method | Method Signature | Purpose | Return |
|------------------------|--|---|--|
| | | taking into account the Z coordinates in the calculation. | Resultant column contains a 0, if the LineString or MultiLineString is an empty set. |
| line_interpolate_point | line_interpolate_point(self, proportion) | Return a point interpolated along a Geometry type that represents an ST_LineString or GeoSequence value, given a proportional distance along that line. | GeoDataFrameColumn with result column containing ST_Point Geometry values |
| num_points | num_points(self) | Return the number of points in a Geometry type that represents an ST_LineString or GeoSequence value. | GeoDataFrameColumn Resultant column contains a NULL, if the Geometry is an empty set. |
| point | point(self, position) | Return the specified point from a Geometry type that represents an ST_LineString or GeoSequence value. | teradataml GeoDataFrameColumn |
| start_point | start_point(self) | Return the start point of an ST_LineString or GeoSequence value. Note: This method returns a point having a Z coordinate if the input geometry is three-dimensional. | teradataml GeoDataFrameColumn |

Methods for Polygon Geometry

| Method | Method Signature | Purpose | Return |
|-------------------|---------------------------|--|--|
| interiors | interiors(self, position) | Return the specified interior ring of a Geometry type that represents an ST_Polygon value. | GeoDataFrameColumn with result column containing ST_LineString Geometry values |
| num_interior_ring | num_interior_ring(self) | Return the number of interior rings of a Geometry type that represents an ST_Polygon value. | teradataml GeoDataFrameColumn |
| point_on_surface | point_on_surface(self) | Return a point that is guaranteed to spatially intersect an ST_Polygon, or at least one of the component polygons of an ST_MultiPolygon. | teradataml GeoDataFrameColumn |

Methods for GeometryCollection Geometry

| Method | Method Signature | Purpose | Return |
|--------------------|---------------------------------------|---|----------------------------------|
| geom_ component | geom_ component(self, position) | Return the geometry of one component member of a composite geometry type (ST_GeomCollection, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon). The element to be returned is specified by position with the collection. | teradataml GeoDataFrameColumn |
| num_ geometry | num_ geometry(self) | Return the number of distinct geometries that constitute a composite geometry type (ST_GeomCollection, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon). | teradataml GeoDataFrameColumn |

Methods for GeoSequence Geometry

| Method | Method Signature | Purpose | Return |
|--------------------------|---|---|----------------------------------|
| clip | clip(self, start_ timestamp, end_timestamp) | Return a GeoSequence type containing the subset of points and associated data that lie between the two timestamp input arguments. | teradataml GeoDataFrameColumn |
| get_final_ timestamp | get_ final_timestamp(self) | Return the TimeStamp of the last point of a GeoSequence. | teradataml GeoDataFrameColumn |
| get_init_ timestamp | get_ init_timestamp(self) | Return the TimeStamp of the first point of a GeoSequence. | teradataml GeoDataFrameColumn |
| get_link | get_link(self, index) | Get the link ID of a specified point in a GeoSequence. | teradataml GeoDataFrameColumn |
| get_user_ field | get_user_field(self, field_index, index) | Return the user field specified by "field_index" for the point specified by "index" for a GeoSequence type. | teradataml GeoDataFrameColumn |
| get_user_ field_count | get_user_ field_count(self) | Return the number of user fields associated with each point of a GeoSequence. | teradataml GeoDataFrameColumn |
| point_ heading | point_ heading(self, index) | Return the heading for the specified point of a GeoSequence. The value is calculated as the angle (in degrees) between a vertical line and the line segment from the specified point to the next point clockwise from the North. | teradataml GeoDataFrameColumn |
| set_link | set_link(self, index, link_id) | Set the link ID of a specified point in a GeoSequence. | teradataml GeoDataFrameColumn |

| Method | Method Signature | Purpose | Return |
|--------|---|--|----------------------------------|
| speed | speed(self, index=None, begin_index=None, end_index=None) | Return the approximate speed at a specified point (speed(index INTEGER)) or between two points (speed(iBegin INTEGER, iEnd INTEGER)) for a GeoSequence type. | teradataml GeoDataFrameColumn |

Filtering Functions and Methods

| Method | Method Signature | Purpose | Return |
|----------------|-----------------------------------|---|---|
| intersects_mbb | intersects_mbb(self, geom_column) | Test whether a 3D geometry spatially intersects a specified MBB. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the input 3D geometry intersects the MBB argument. • 0, if the input 3D geometry does not intersect the MBB argument. • Return an error if the input geometry is not 3D (does not have a z coordinate). |
| mbb_filter | mbb_filter(self, geom_column) | Test whether the MBBs of two 3D geometries spatially intersect. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the MBB of the input 3D geometry intersects the MBB of the geometry passed to the method. • 0, if the MBB of the input 3D geometry does not intersect the MBB of the geometry passed to the method. • Return an error if either geometry is not 3D (does not have a z coordinate). |
| mbr_filter | mbr_filter(self, geom_column) | Test whether the MBRs of two 2D geometries spatially intersect. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the MBR of the input geometry intersects the MBR of the geometry passed to the method. • 0, if the MBR of the input geometry does not intersect the MBR of the geometry passed to the method. |
| within_mbb | within_mbb(self, geom_column) | Test whether a 3D geometry is spatially within the bounds of a specified MBB. | GeoDataFrameColumn Resultant column contains: <ul style="list-style-type: none"> • 1, if the input 3D geometry is spatially within MBB argument. • 0, if the input 3D geometry is not spatially within the MBB argument. • Return an error if the input geometry is not 3D (does not have a z coordinate). |

Using Teradata Vantage Analytic Functions with Teradata Package for Python

Teradata Package for Python provides a Python interface to all of the analytic functions in Vantage: ML Engine analytic functions and Analytics Database in-database analytic functions.

The `teradataml` analytics package includes two subpackages:

- `teradataml.analytics.mle`
- `teradataml.analytics.sqlc`

All `teradataml` analytic functions are in either of these two subpackages.

Teradata recommends importing the desired analytic function in one of the following two ways:

- For best practice, import from the `teradataml` package.

For example:

```
# import DecisionTreePredict from the teradataml package (recommended)
from teradataml import DecisionTreePredict
```

- To choose the engine where the analytic function is used, import from the desired subpackage.

For example:

```
# import DecisionTreePredict from the teradataml.analytics.sqlc package
from teradataml.analytics.sqlc import DecisionTreePredict
```

Note:

Older versions of `teradataml` placed all analytic functions in a single package (`teradataml.analytics`), with each analytic function in its own module.

From release 16.20.00.01, the analytic functions are in either of the two subpackages, with each analytic function in its own module.

Instead of using the old modules in the `teradataml.analytics` package, Teradata recommends importing the desired analytic function in one of the ways listed in the previous examples.

Instantiating an analytic function from the old module in the `teradataml.analytics` package gives a `DeprecationWarning`.

For example:

```
>>> from teradataml.analytics.KMeans import KMeans
>>> KMeans(*args, **kw)
```

DeprecationWarning:

The "teradataml.analytics.KMeans.KMeans" class has moved to a new package in version 16.20.00.02.

Import from the `teradataml` package, `teradataml.analytics` package, or directly from the `teradataml.analytics.mle.KMeans` module.

See the `teradataml 16.20.00.02 User Guide` for more information.

This section shows examples of how the Teradata Package for Python enables users to run a selection of these analytic functions through a Python interface.

In this section, assume you have the connection to the Analytics Database as user "tdapUser", where the target database is "tdapUserDB".

See [Testing Connection to Vantage](#) for details on how to set up a connection and import the `teradataml` module.

Usage Notes

Note:

- Analytics Database functions and UAF functions are available only if underlying Vantage, which user is connected to, supports those.
 - You should import these functions only after establishing the connection to Vantage.
Importing these functions before context creation raises import error.
 - You can check whether Analytics Database functions and UAF functions are available by running the function `display_analytic_functions()`.
 - If these functions are available for the corresponding Vantage version, then `display_analytic_functions()` displays those functions.
 - Otherwise, these functions are not displayed in `display_analytic_functions()` output.
-

Column Specification in Analytics Database Analytic Functions

Column Specification

Some analytic functions in the Analytics Database provide arguments for selecting or performing operations on multiple columns.

teradataml allows user to specify columns in the following ways:

- Pass a single column name.
For example, `column_arg = "col1"`
- Pass multiple columns (specific columns only).
For example, `column_arg = ["col1", "col3", "col8"]`
- Pass multiple columns using `DataFrame.columns` and slice filtering.
For example, `column_arg = list(set(df.columns[2:10]) - set(df.columns[5:7]))`
- Pass multiple column as a column range.
For example, `column_arg = "<column_range>"`

Specifying Column Range `column_range`

Column range can be specified in the following ways:

- Without column exclusion.
Syntax: `"start_column:end_column"`

Note:

Must be passed as string.

- With column exclusion.

Syntax: ["start_column:end_column", "-exclude_column1", ...]

Note:

Must be passed as a list of strings.

The *start_column* and *end_column* can be:

- Column names.
For example, "column1:column2".
- Nonnegative integers that represent the indexes of columns in the table. The first column has index 0.
For example, "0:4" specifies the first five columns in the table.
- Empty.
For example,
 - ":4" or ":columnD" specifies all columns up to and including the column with index 4 or columnD.
 - "4:" or "columnD:" specifies the column with index 4 (or columnD) and all columns after it.
 - ":" specifies all columns in the table.

The *exclude_column* is a column in the specified range, represented by either its name or its index.

For example, ["0:99", "-[50]", "-column10"] specifies the columns with indexes from 0 to 99, except the column with index 50 and column10.

Note:

Column ranges cannot overlap, and cannot include any specified column.

When handling column name containing range separators:

- Column can be enclosed in double quotes.
For example, if DataFrame has columns :columnA, columnB: and :column:C:, columns can be selected by enclosing in double quotes:
 - "\" :columnA\" " or '":columnA''
 - "\" columnB:\" " or '"columnB: ''
 - "\" :column:C:\" " or '":column:C: ''
- You can always use column range and exclusion by index instead of column names.

Column Range Examples

Assume "insect_sprays" DataFrame has columns groupA, :groupB, groupC:CC, groupD:, groupE:EE:EEE and :groupF:, the following examples show how to select range of columns for the *group_columns* argument in the ANOVA function.

Example 1: Specify range index of columns.

```
ANOVA(data=insect_sprays,
      group_columns="2:5",
      alpha = 0.025)
```

Example 2: Specify range index of columns with first column empty.

```
ANOVA(data=insect_sprays,
      group_columns=":5",
      alpha=0.025)
```

Example 3: Specify range of columns by their names. Column has separator, hence enclose it in double quotes.

```
ANOVA(data=insect_sprays,
      group_columns= '"groupC:CC":"groupE:EE:EEE"',
      alpha = 0.05)
```

Example 4: Specify range of columns by their names with second column empty. Column has separator, hence enclose it in double quotes.

```
ANOVA(data=insect_sprays,
      group_columns= '"groupE:EE:EEE":',
      alpha = 0.05)
```

Example 5: Specify range of columns for all columns. Column has separator, hence enclose it in double quotes.

```
ANOVA(data=insect_sprays,
      group_columns=[':', "-\":groupB\\"", # or [':', "-[2]"]
      alpha = 0.05)
```

load_example_data() Function

The `load_example_data()` is a helper function that loads the sample datasets.

Teradata Package for Python offers various APIs and each API provides some examples. To test these examples, users need the sample datasets loaded in Vantage.

The `load_example_data()` function can only be used in a restricted way. Function arguments only accept predetermined values as shown in the given examples in this User Guide and the [Teradata Package for Python Function Reference](#).

- *function_name*

This required argument contains the prefix name of the example JSON file to be used to load data.

Note:

You must specify the *function_name* values as specified in the example sections of corresponding teradataml APIs. If any other string is passed as prefix input, an error will be raised as 'prefix_str_example.json' file not found. This *_example.json file contains the schema information for the tables that can be loaded using this JSON file.

- *table_name*

This required argument specifies the name(s) of the table to be created in the database.

Note:

Table names provided here must have an equivalent datafile (CSV) present at teradataml/data. Schema information for the same must also be present in <function_name>_example.json as shown in 'function_name' argument description.

Note:

- The function creates a new table in Vantage with the name specified in the *table_name* argument. You must manually drop the table if required.
- If a table with the name provided for *table_name* argument already exists, this function skips creation and loading of the dataset.
- Database used for loading table depends on the following conditions:
 - If the configuration option **temp_table_database** is set, then the tables are loaded in the database specified in this option.
 - If the configuration option **temp_table_database** is not set and the *temp_database_name* argument is used while creating context, then the tables are loaded in the database specified in the *temp_database_name* argument.
 - If none of them are specified, then the tables are created in the connecting users' default database or the connecting database.

Example 1: When connection is created without *temp_database_name*, table is loaded in users' default database

```
>>> from teradataml import *

>>> con = create_context(host = 'tdhost', username='tduser', password
= 'tdpassword')

>>> load_example_data("pack", "ville_temperature")
```

```
# Create a teradataml DataFrame.
>>> df = DataFrame("ville_temperature")
```

Example 2: When connection is created with *temp_database_name*, table is loaded in that database

This examples shows when connection is created with the *temp_database_name* argument, then tables are loaded in the database specified in the *temp_database_name* argument.

This example also demonstrates loading multiple tables in a single function call.

```
>>> con = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', temp_database_name = "temp_db")

>>> load_example_data("GLM", ["admissions_train", "housing_train"])

# Create teradataml DataFrames.
>>> admission_train = DataFrame(in_schema("temp_db", "admissions_train"))
>>> housing_train = DataFrame(in_schema("temp_db", "housing_train"))
```

Example 3: When connection is created with *temp_database_name* and the configuration option *temp_table_database* is set

This example shows when connection is created with *temp_database_name* and the configuration option **temp_table_database** is also specified, the table is created in the database specified in the configuration option **temp_table_database**.

```
>>> from teradataml import *

>>> con = create_context(host = 'tdhost', username='tduser', password =
'tdpassword', temp_database_name = "temp_db")

>>> configure.temp_table_database = "temp_db1"

>>> load_example_data("pack", "ville_temperature")

# Create a teradataml DataFrame.
>>> df = DataFrame(in_schema("temp_db1", "ville_temperature"))
```

Use Cases

Text Analysis with teradataml Package

This example investigates a log of vehicle complaints that have been categorized as crash-related or not crash-related. Use this log to build a Naïve Bayes Text Classifier model, and then apply the model to a new log data to predict if the complaint is associated with a crash.

This example shows the steps to build a Naïve Bayes Text Classifier model and then apply the model to the new log data.

1. Import the required modules and load the example datasets.

```
from teradataml import NaiveBayesTextClassifierTrainer
from teradataml import NaiveBayesTextClassifierPredict
from teradataml import TextParser
```

```
from teradataml import load_example_data
from teradataml.dataframe.dataframe import DataFrame
```

```
# Load the data to run the example.
load_example_data("TextTokenizer", "complaints")
```

2. Create a teradataml DataFrame from the training dataset.

```
complaints = DataFrame.from_table("complaints")
```

3. Create tokens from the training dataset.

```
text_tokenizer_out = TextParser(data=complaints,
                                text_column="text_data",
                                remove_stopwords=True,
                                accumulate=["doc_id", "category"])
```

4. Create a teradataml DataFrame "tddf_nbayes_tokens" consisting of the tokens from the training dataset, ignoring the case (*lower()*).

```
tddf_nbayes_tokens = text_tokenizer_out.result.assign(drop_columns = True,
                                                    doc_id
= text_tokenizer_out.result.doc_id,
                                                    token
= text_tokenizer_out.result.token.str.lower(),
                                                    category
= text_tokenizer_out.result.category)
```

5. Train a new Naïve Bayes Text Classifier based on the teradataml DataFrame from the training dataset, using the NaiveBayesTextClassifierTrainer function from teradataml package.

```

nb_textclassifier_model = NaiveBayesTextClassifierTrainer(data
= tddf_nbayes_tokens,

data_partition_column = "category",

token_column

= "token",

doc_category_column

= "category")

```

6. Get the test data or creates a sample test data.

```

import pandas as pd
nbayes_test = {"doc_id": range(1, 11),
"text_data":
["ELECTRICAL CONTROL MODULE IS SHORTENING OUT, CAUSING THE VEHICLE TO STALL.
ENGINE WILL BECOME TOTALLY INOPERATIVE. CONSUMER HAD TO CHANGE ALTERNATOR/
BATTERY AND STARTER, AND MODULE REPLACED 4 TIMES, BUT DEFECT STILL OCCURRING
CANNOT DETERMINE WHAT IS CAUSING THE PROBLEM.",
"ABS BRAKES FAIL TO OPERATE PROPERLY, AND AIR BAGS FAILED TO DEPLOY DURING A
CRASH AT APPROX. 28 MPH IMPACT. MANUFACTURER NOTIFIED.",
"WHILE DRIVING AT 60 MPH GAS PEDAL GOT STUCK DUE TO THE RUBBER THAT IS
AROUND THE GAS PEDAL.",
"THERE IS A KNOCKING NOISE COMING FROM THE CATALYTIC CONVERTER, AND THE
VEHICLE IS STALLING. ALSO, HAS PROBLEM WITH THE STEERING.",
"CONSUMER WAS MAKING A TURN, DRIVING AT APPROX 5-10 MPH WHEN CONSUMER HIT
ANOTHER VEHICLE. UPON IMPACT, DUAL AIRBAGS DID NOT DEPLOY. ALL DAMAGE WAS
DONE FROM ENGINE TO TRANSMISSION, TO THE FRONT OF VEHICLE, AND THE VEHICLE
CONSIDERED A TOTAL LOSS.",
"WHEEL BEARING AND HUBS CRACKED, CAUSING THE METAL TO GRIND WHEN MAKING A
RIGHT TURN. ALSO WHEN APPLYING THE BRAKES, PEDAL GOES TO THE FLOOR, CAUSE
UNKNOWN. WAS ADVISED BY MIDAS NOT TO DRIVE VEHICLE- WHEEL COULD COME OFF.",
"DRIVING ABOUT 5-10 MPH, THE VEHICLE HAD A LOW FRONTAL IMPACT IN WHICH THE
OTHER VEHICLE HAD NO DAMAGES. UPON IMPACT, DRIVER'S AND THE PASSENGER'S
AIR BAGS DID NOT DEPLOY, RESULTING IN INJURIES. PLEASE PROVIDE FURTHER
INFORMATION AND VIN#.",
"THE AIR BAG WARNING LIGHT HAS COME ON, INDICATING AIRBAGS ARE INOPERATIVE.
THEY WERE FIXED ONE AT THE TIME, BUT PROBLEM HAS REOCCURRED.",
"CONSUMER WAS DRIVING WEST WHEN THE OTHER CAR WAS GOING EAST. THE OTHER
CAR TURNED IN FRONT OF CONSUMER'S VEHICLE, CONSUMER HIT OTHER VEHICLE AND
STARTED TO SPIN AROUND, COULDN'T STOP, RESULTING IN A CRASH. UPON IMPACT,
AIRBAGS DIDN'T DEPLOY.",
"WHILE DRIVING ABOUT 65 MPH AND THE TRANSMISSION MADE A STRANGE NOISE, AND
THE LEFT FRONT AXLE LOCKED UP. THE DEALER HAS REPAIRED THE VEHICLE."]}

```

```
nbayes_test = pd.DataFrame(nbayes_test)
from teradataml import copy_to_sql
copy_to_sql(nbayes_test, table_name="nbayes_test")
```

Next, apply the model to the test data.

7. Create a teradataml DataFrame from the test dataset.

```
nbayes_test = DataFrame.from_table("nbayes_test")
```

8. Create tokens from the test dataset.

```
text_tokenizer_out_2 = TextParser(data=nbayes_test,
                                   text_column="text_data",
                                   remove_stopwords=True,
                                   accumulate="doc_id")
```

9. Create a teradataml DataFrame "tddf_nbayes_tokens_test" with the tokens from the test dataset, ignoring the case (*lower()*).

```
tddf_nbayes_tokens_test = text_tokenizer_out_2.result.assign(drop_columns
= True,
                                                             doc_id
= text_tokenizer_out_2.result.doc_id,
                                                             token
= text_tokenizer_out_2.result.token.str.lower() )
```

10. Predict the categories ('crash' or 'no crash') by applying the Naïve Bayes Text Classifier model to the teradataml DataFrame from the test dataset, using the NaiveBayesTextClassifierPredict function.

```
nb_textclassifier_pred = NaiveBayesTextClassifierPredict(newdata
= tddf_nbayes_tokens_test,
                                                         object
= nb_textclassifier_model.result,
newdata_partition_column = "doc_id",
                                                         input_token_column
= "token",
                                                         doc_id_columns
= "doc_id" )
```

11. Inspect the results.

```
nb_textclassifier_pred
```


Using ClassificationEvaluator to Evaluate the Classification

This example performs prediction on a model and evaluates the model, then generates the statistics for the classification.

1. Set up the environment.
 - a. Import required libraries.

```
import tempfile
```

```
import getpass
```

```
from teradataml import DataFrame, load_example_data, create_context
```

- b. Create the connection to database.

```
con = create_context(host=getpass.getpass("Hostname: "),
                    username=getpass.getpass("Username: "),
                    password=getpass.getpass("Password: "))
```

- c. Load example data and create required dataframes.

```
load_example_data("textparser", ["complaints", "stop_words"])
```

```
complaints = DataFrame.from_table("complaints")
```

```
stop_words = DataFrame.from_table("stop_words")
```

2. Train the model and create classifier for crash or nocrash.

- a. Check the list of available analytic functions.

```
display_analytic_functions()
```

- b. Import functions TextParser, NaiveBayesTextClassifierTrainer, NaiveBayesTextClassifierPredict.

```
from teradataml import TextParser,
NaiveBayesTextClassifierTrainer, NaiveBayesTextClassifierPredict
```

- c. Tokenize the "text_column" and accumulate result by "doc_id" and "category".

```
complaints_tokenized = TextParser(data=complaints,
                                  text_column="text_data",
                                  object=stop_words,
                                  remove_stopwords=True,
                                  accumulate=["doc_id", "category"])
```

- d. Calculate the conditional probabilities for token-category pairs.

```
NaiveBayesTextClassifierTrainer_out =
NaiveBayesTextClassifierTrainer(data=complaints_tokenized.result,

token_column="token",

doc_category_column="category")
```

- e. Print the result DataFrames.

```
print(NaiveBayesTextClassifierTrainer_out.result)
print(NaiveBayesTextClassifierTrainer_out.model_data)
```

- f. Score the data using NaiveBayesTextClassifierPredict() on model generated by NaiveBayesTextClassifier() where model_type is "MULTINOMIAL".

```
nbt_predict_out = NaiveBayesTextClassifierPredict(object
= NaiveBayesTextClassifierTrainer_out.model_data,
                                                    newdata
= complaints_tokenized.result,
                                                    input_token_column
= 'token',
                                                    accumulate="category",
                                                    doc_id_columns
= 'doc_id')
```

- g. Print the result DataFrame.

```
print(nbt_predict_out.result)
```

3. Convert prediction column and category column to same DataType.

```
from teradataml import ConvertTo

predicted_data = ConvertTo(data = nbt_predict_out.result,
                           target_columns = ["category", "prediction"],
                           target_datatype
= ["VARCHAR(charlen=20, charset=UNICODE, casespecific=NO)"])
```

4. Evaluate classifier.

- a. Import function ClassificationEvaluator.

```
from teradataml import ClassificationEvaluator
```

- b. Evaluate classification.

```

ClassificationEvaluator_obj =
ClassificationEvaluator(data=predicted_data.result,

observation_column='category',

prediction_column='prediction',

labels=['no_crash', 'crash'])

```

- c. Print the result DataFrames.

```

print(ClassificationEvaluator_obj.result)
print(ClassificationEvaluator_obj.output_data)

```

Using GLM Model with teradataml Package

This example shows the steps to build a Generalized Linear model (GLM) and then apply the model to the new testing admissions data. The data set contains two classes, where one class represents the successful admission while the other represents no admission.

Note:

This example uses Vantage Analytic Library (VAL) functions. You must make sure VAL functions are installed in Vantage before using VAL functions.

1. Import required libraries.

```

from teradataml import GLM
from teradataml import TDGLMPredict
from teradataml.dataframe.dataframe import DataFrame

```

2. Create training data.

- a. If the input table (admissions_train) does not exist already, create the table and load the dataset into the table.

```
load_example_data("dataframe", "admissions_train")
```

- b. Create a teradataml DataFrame for the training dataset from "admissions_train" table.

```
admissions_train = DataFrame.from_table("admissions_train")
```

3. Convert categorical columns in the input data table into numeric columns using OneHotEncoder() and Transform() functions from VAL, as GLM() function supports only numeric columns.

- a. Import required libraries.

```
from teradataml import valib, OneHotEncoder, Retain
```

- b. Configure VAL install location.

```
configure.val_install_location = "VAL"
```

- c. Define encoders for categorical columns.

```
masters_code = OneHotEncoder(values=["yes", "no"],
                              columns="masters",
                              out_columns="masters")
stats_code = OneHotEncoder(values=["Advanced", "Novice"],
                            columns="stats",
                            out_columns="stats")
programming_code = OneHotEncoder(values=["Advanced",
                                         "Novice", "Beginner"],
                                  columns="programming",
                                  out_columns="programming")
```

- d. Retain numerical columns.

```
retain = Retain(columns=["admitted", "gpa"])
```

- e. Transform categorical columns to numeric columns.

```
all_numeruc_admissions_train = valib.Transform(data=admissions_train,
                                                one_hot_encode=[masters_code, stats_code, programming_code],
                                                retain=retain)
```

4. Train a new Generalized Linear Model (GLM) based on the teradataml DataFrame from the training dataset, using the train function - GLM() function from teradataml package.

```
glm_train = GLM(formula="admitted ~ gpa + yes_masters + no_masters +
Advanced_stats + Novice_stats + Advanced_programming + Novice_programming
+ Beginner_programming",
                 data=all_numeruc_admissions_train.result,
                 learning_rate="INVTIME",
                 momentum=0.80)
```

Next, apply the model to the test data using TDGLMPredict() function.

5. Create test data.

- a. If the input table (admissions_test) does not exist already, create the table and load the dataset into the table.

```
load_example_data("GLMPredict", "admissions_test")
```

- b. Create a teradataml DataFrame for the test dataset from "admissions_test" table.

```
admissions_test = DataFrame.from_table("admissions_test")
```

6. Convert categorical columns in test input data table into numeric columns using OneHotEncoder() and Transform() functions from VAL, as GLM() function has created the model on "all_numeric_admissions_train" table having only numeric columns.

- a. Import required libraries.

```
from teradataml import valib, OneHotEncoder, Retain
```

- b. Configure VAL install location.

```
configure.val_install_location = "VAL"
```

- c. Define encoders for categorical columns.

```
masters_code = OneHotEncoder(values=["yes", "no"],
                              columns="masters",
                              out_columns="masters")
stats_code = OneHotEncoder(values=["Advanced", "Novice"],
                            columns="stats",
                            out_columns="stats")
programming_code = OneHotEncoder(values=["Advanced",
                                         "Novice", "Beginner"],
                                  columns="programming",
                                  out_columns="programming")
```

- d. Retain numerical columns.

```
retain = Retain(columns=["admitted", "gpa"])
```

- e. Transform categorical columns to numeric columns.

```
all_numeric_admissions_test= valib.Transform(data=admissions_test,
                                              one_hot_encode=[masters_code, stats_code, programming_code],
                                              retain=retain)
```

7. Predict the admission status by applying the Generalized Linear Model (GLM) to the teradataml DataFrame from the test dataset, using the TDGLMPredict() function and output of the train function.

```
tdglmpredict_out = TDGLMPredict(object= glm_train.result,
                                 newdata= all_numeric_admissions_test.result,
                                 id_column="id")
```

8. Inspect the results.

```
tdglmpredict_out.result
```

Using Decision Forest Model with teradataml Package

This section uses iris data with three classes. The dataset contains 150 samples, each with four features describing flower properties, and a fifth column indicates the flower species.

In this example, you build a Decision Forest model based on the training dataset and apply the model to the test dataset to evaluate the performance of the model.

1. Import the required modules.

```
from teradataml import DecisionForest
from teradataml import DecisionForestPredict
from teradataml import load_example_data
from teradataml.dataframe.dataframe import DataFrame
```

2. If the input table "iris_input" does not already exist, create it and load the dataset.

```
load_example_data("byom", "iris_input")
```

3. Create a teradataml DataFrame from the loaded dataset.
 - a. Create a teradataml DataFrame "iris_input" consisting the tokens from the training dataset.

```
iris_input = DataFrame("iris_input")
```

- b. Create two samples of input data: sample 1 has 80% of the total rows for training the model ("iris_train"), and sample 2 has 20% of the total rows for testing the model ("iris_test"). First, sample the "iris_input" dataframe.

```
iris_sample = iris_input.sample(frac=[0.8, 0.2])
```

- c. Create train dataset from sample 1 by filtering on "sampleid" and drop "sampleid" column as it is not required for training model.

```
iris_train = iris_sample[iris_sample.sampleid == "1"].drop("sampleid",
axis = 1)
```

- d. Create test dataset from sample 2 by filtering on "sampleid" and drop "sampleid" column as it is not required for scoring.

```
iris_test = iris_sample[iris_sample.sampleid == "2"].drop("sampleid",
axis = 1)
```

4. Train a new Decision Forest model based on the teradataml DataFrame "iris_train" from the training dataset, using the DecisionForest function from teradataml package.
This can be done with or without using the *formula* argument.

Example 1: Train the decision forest Classification model using input teradataml dataframe and provided the *formula* argument.

```
formula = "species ~ sepal_length + sepal_width + petal_length + petal_width"

# Train the Decision Forest model.
rft_model = DecisionForest(data=iris_train,
                           formula = formula,
                           tree_type="classification",
                           ntree=50,
                           tree_size=100,
                           nodesize=1,
                           variance=0.0,
                           max_depth=12,
                           maxnum_categorical=20,
                           mtry=3,
                           mtry_seed=100,
                           seed=100)
```

Example 2: Train the same decision forest Classification model (rft_model) without using the *formula* argument.

```
rft_model = DecisionForest(data=iris_train,
                           input_columns=["sepal_length", "sepal_width",
                                           "petal_length", "petal_width"],
                           response_column="species",
                           tree_type="classification",
                           ntree=50,
                           tree_size=100,
                           nodesize=1,
                           variance=0.0,
                           max_depth=12,
                           maxnum_categorical=20,
                           mtry=3,
                           mtry_seed=100,
                           seed=100)
```

Once the model is created, you can apply the model to the test dataset.

5. Predict the iris species by applying the Decision Forest model to the teradataml DataFrame "iris_test" from the test dataset, using the DecisionForestPredict function.

```
decision_forest_predict_out = DecisionForestPredict(object = rft_model,
                                                    newdata = iris_test,
                                                    id_column = "id",
                                                    detailed = False,
```

```
terms = ["species"]
)
```

6. Inspect the results.

```
decision_forest_predict_out.result
```

Clustering Using KMeans with teradataml Package

This example uses the US Arrests data of 50 samples containing statistics for arrests made per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973, along with the percentage of the population living in urban areas.

Example here shows how to use KMeans clustering function.

1. Import the required modules.

```
from teradataml import KMeans, KMeansPredict
from teradataml.dataframe.dataframe import DataFrame
from teradataml.data.load_example_data import load_example_data
import matplotlib.pyplot as plt
```

2. If the input table "kmeans_us_arrests_data" does not already exist, create the table and load the datasets into the table.

```
load_example_data("kmeans", "kmeans_us_arrests_data")
```

3. Create a teradataml DataFrame from "kmeans_us_arrests_data".

```
## Creating TeradataML dataframes
df_train = DataFrame('kmeans_us_arrests_data')
```

```
# Print train data to see how the sample train data looks like
print("\nHead(10) of Train data:")
print(df_train.head(10))
```

4. In the training dataset, the features are 'sno', 'state', 'murder', 'assault', 'urban_pop' and 'rape'. And 'sno' to 'state' is a one-to-one mapping. For training, drop off the repeated feature 'state'.

```
# A dictionary to get 'sno' to 'state' mapping. Required for plotting.
df1 = df_train.select(['sno', 'state'])
sno_to_state = dict(df1.to_pandas()['state'])
print(sno_to_state)
```

```
# No need of 'state' columns, instead we have 'sno' column for the same
df_train = df_train.drop(['state'], axis=1)
colnames = ["sno", "murder", "assault", "urban_pop", "rape"]
```


5. Apply KMeans algorithm to generate two clusters and inspect the outputs.

a. Apply KMeans algorithm.

```
## Train the KMeans model with 2 clusters.
kMeans_model = KMeans(id_column="sno",
                      target_columns=['murder', 'assault',
                                     'urban_pop', 'rape'],
                      data=df_train,
                      num_init=10,
                      num_clusters=2)
```

```
# Print the KMeans model training results
print(kMeans_model.result)
```

b. 'model_data' dataframe presents overall summary of the trained KMeans model.

```
print(kMeans_model.model_data)
```

c. Use KMeansPredict function to perform predictions using trained KMeans model.

```
# KMeansPredict function performs prediction on the dataset using trained
KMeans model.
kMeans_output = KMeansPredict(object=kMeans_model.result,
                              data=df_train)
```

d. 'result' dataframe presents the 'sno' representing state and the corresponding cluster id for all the samples.

```
kMeans_output.result.head(30)
```

6. Quick Analysis of clustering output by plotting clusters based on features.

```
## Inner join of clustered_output to actual dataset df_train We shall use
the data from df1 to plot.
df1 = df_train.join(kMeans_output.clustered_output, how='inner', on=['sno'],
                    lsuffix='t1', rsuffix='t2')
```

```
print("\nInner join of clustered_output to actual dataset df_train:")
print(df1)
```

a. Plot clusters based on the two features 'urban_pop' and 'murder'.

```
# Selecting only the necessary features for plot.
df3 = df1.select(['t1_sno', 'urban_pop', 'murder', 'td_clusterid_kmeans'])
```

```
# Since there is no plotting possible for teradataml DataFrame, we are
converting it to
```

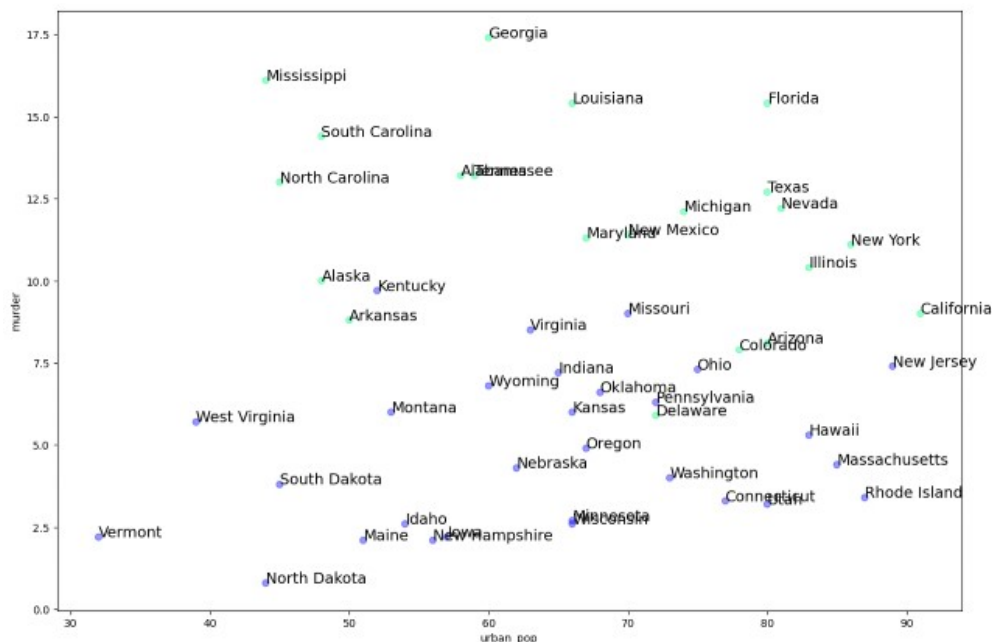
```
# pandas dataframe and then to numpy_array 'numpy_df' to use matplotlib
library of python.
pandas_df = df3.to_pandas()
numpy_df = pandas_df.values
```

```
# Setting figure display size.
plt.rcParams['figure.figsize'] = [15, 10]
```

```
# Coloring based on 'td_clusterid_kmeans'.
plt.scatter(numpy_df[:,1], numpy_df[:,2], c=numpy_df[:,3],
            cmap='winter_r', alpha=0.4)
for ind, value in enumerate(numpy_df[:, 0]):
    # sno_to_state is used hear to get state names.
    plt.text(numpy_df[ind,1], numpy_df[ind,2],
            sno_to_state[int(value)], fontsize=14)
```

```
plt.xlabel('urban_pop')
plt.ylabel('murder')
plt.show()
```

After running these commands, the following plot shows:



- b. Plot the clusters based on the two features 'rape' and 'murder'.

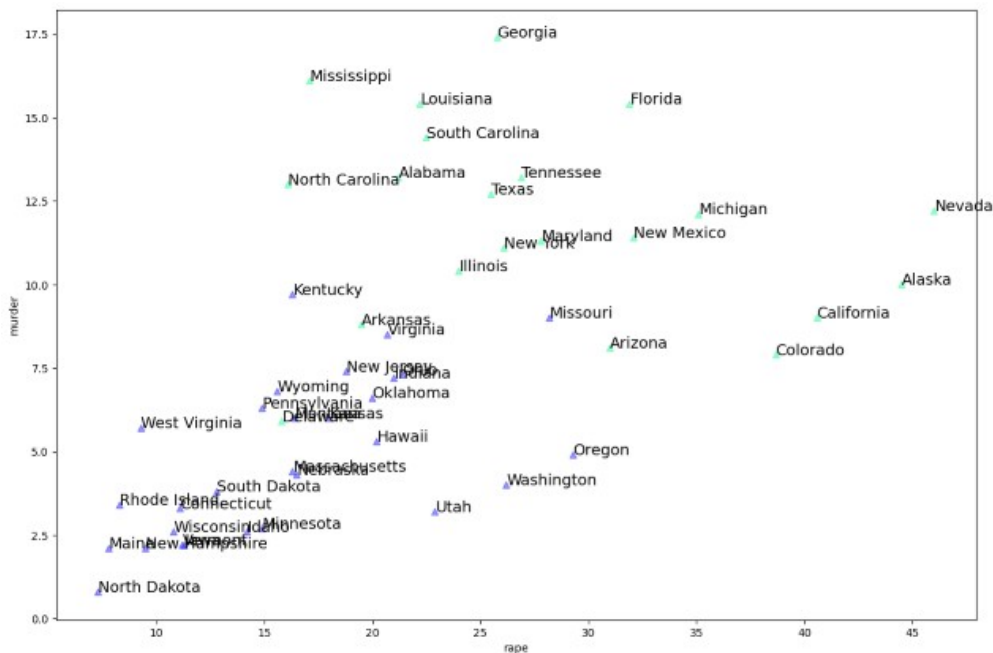
```
# Selecting only the necessary features for plot.
df3 = df1.select(['t1_sno', 'rape', 'murder', 'td_clusterid_kmeans'])
```

```
# Since there is no plotting possible for teradataml DataFrame, we are
# converting it to
# pandas dataframe and then to numpy_array 'numpy_df' to use matplotlib
# library of python.
pandas_df = df3.to_pandas()
numpy_df = pandas_df.values

# Coloring based on 'td_clusterid_kmeans'.
plt.scatter(numpy_df[:,1], numpy_df[:,2], c=numpy_df[:,3],
            cmap='winter_r', marker='^', alpha=0.4)
for ind, value in enumerate(numpy_df[:, 0]):
    # sno_to_state is used here to get state names.
    plt.text(numpy_df[ind,1], numpy_df[ind,2],
             sno_to_state[int(value)], fontsize=14)

plt.xlabel('rape')
plt.ylabel('murder')
plt.show()
```

After running these command, the following plot shows:



Churn Analysis Using Sessionize and NPath with Teradata Package for Python

This example uses the retail dataset of product purchase and returns to perform churn analysis.

1. Import the required modules.

```
from teradataml.analytics.sql.Sessionize import Sessionize
from teradataml.analytics.sql.NPath import NPath
from teradataml.dataframe.dataframe import DataFrame, in_schema
from teradataml.data.load_example_data import load_example_data
```

2. Load the retail churn data.

```
load_example_data("sessionize", "retail_churn_table")
```

3. Create the DataFrame on the loaded dataset 'retail_churn_table'.

```
df1 = DataFrame(in_schema('alice', 'retail_churn_table'))
# in_schema(databaseName, tableName) - Assuming table has been created in
# 'alice' database.
print("***** Let's take a peek at the data *****")
print(df1)
print("\n***** Retails Churn Table Column Types *****")
print(df1.dtypes)
print("\n***** Row Count and column count for the retail data *****")
print(df1.shape)
```

4. Sessionize.

The Sessionize function maps each click in a session to a unique session identifier. A session is defined as a sequence of clicks by one user that are separated by at most n seconds.

- a. Example 1: Call the Sessionize function – NO CHURN CUSTOMER.

The non-churn group (denoted by the churn flag value of 'N'). Filter rows where churn_flag = 'N'.

```
input_retail_data = df1[df1.churn_flag == 'N']
print("***** No churn data *****")
print(input_retail_data)
```

```
# Sessionize the non-churn data
daily_session_nochurn = Sessionize(data = input_retail_data,
                                   time_out = 86400.0,
                                   data_order_column = 'datestamp',
                                   data_partition_column = 'customer_id',
                                   time_column = 'datestamp')
```

```
# Data checking
# SELECT * FROM xxxx ORDER BY 2 ASC WHERE customer_id=1531;
# Here xxxx is the table/view created from daily_session_nochurn.result.
res = daily_session_nochurn.result
res = res[res.customer_id == 1531].sort(['datestamp'],ascending = True)
print(res)
```

- b. Example 2: Call the Sessionize function – CHURN CUSTOMER.

The churn group (denoted by the churn flag value of 'Y'). Filter rows where churn_flag = 'Y'.

```
input_retail_data2 = df1[df1.churn_flag == 'Y']
print("***** Churn data *****")
print(input_retail_data2)
```

```
# Sessionize the churn data
daily_session_churn = Sessionize(data = input_retail_data2,
                                time_out = 86400.0,
                                data_order_column = 'datestamp',
                                data_partition_column = 'customer_id',
                                time_column = 'datestamp')
```

```
# Data checking
# SELECT * FROM xxxx ORDER BY 2 ASC WHERE customer_id=64497;
# Here xxxx is the table/view created from daily_session_churn.result.
res2 = daily_session_churn.result
res2 = res2[res2.customer_id == 64497].sort(['datestamp'], ascending=True)
res2
print(res2)
```

5. Use NPath function on the outcome of Sessionize to perform Pathing.

Pathing is the process of discovering a sequence of antecedent actions that occur prior to a specific event of interest. For example, a buyer could browse a site, call a company, read online reviews, test out the product, and then purchase.

Pathing discovers the most salient patterns across a group of individuals or entities based on which further actions are considered.

- a. Call the NPath function – NO CHURN CUSTOMERS.

```
npath_nochurn = NPath(data1 = daily_session_nochurn.result,
                      mode = "NONOVERLAPPING",
                      pattern = 'E*.C',
                      symbols = ["EVENT = 'Purchase' AS C", "EVENT <> 'Purchase'
AS E"],
                      result = ["FIRST(customer_id OF ANY(E,C)) AS customer_id",
                                "FIRST(datestamp OF ANY(E,C)) AS DS_START",
```

```

        "LAST(datestamp OF ANY(E,C)) AS DS_END",
        "COUNT(* OF E) AS EVENT_CNT",
        "ACCUMULATE(EVENT OF ANY(E,C)) AS PATH"],
    data1_partition_column = ['customer_id', 'SESSIONID'],
    data1_order_column = 'datestamp')

```

```

# Shape of the resultant dataframe
npath_nochurn.result.shape

```

```

# Print result
npath_nochurn.result.to_pandas().head(10)

```

```

# Path for a particular customer, 1895
res3 = npath_nochurn.result
res3 = res3[res3.customer_id == 1895]
res3.to_pandas()

```

- b. Call the NPath function – CHURN CUSTOMERS.

```

npath_churn = NPath(data1 = daily_session_churn.result,
                    mode = "NONOVERLAPPING",
                    pattern = 'E*.C',
                    symbols = ["EVENT = 'Product Return' AS C",
                              "EVENT <> 'Product Return' AS E"],
                    result = ["FIRST(customer_id OF ANY(E,C)) AS customer_id",
                              "FIRST(datestamp OF ANY(E,C)) AS DS_START",
                              "LAST(datestamp OF ANY(E,C)) AS DS_END",
                              "COUNT(* OF E) AS EVENT_CNT",
                              "ACCUMULATE(EVENT OF ANY(E,C)) AS PATH"],
                    data1_partition_column = ['customer_id', 'SESSIONID'],
                    data1_order_column = 'datestamp')

```

```

# Shape of the resultant dataframe
npath_churn.result.shape

```

```

# Print result
npath_churn.result.to_pandas().head(10)

```

```

# Path for a particular customer, 64115
res4 = npath_churn.result
res4 = res4[res4.customer_id == 64115]
res4.to_pandas()

```

Using DataFrame.map_partition() Function for GLM Model Fitting and Scoring Functions

This example shows how to fit one or multiple GLM models to the housing data, and then use the models to price the houses in the test data using DataFrame.map_partition() function.

1. Set up

a. Load example data.

The required training and test datasets are included in the teradataml package, and you can use the [load_example_data\(\) Function](#) to load the example data.

```
>>> load_example_data("GLMPredict", ["housing_test", "housing_train"])
```

b. Create input DataFrames.

```
>>> train = DataFrame('housing_train')
```

```
>>> test = DataFrame('housing_test')
```

2. Model fitting

a. Define the user function to fit multiple models to the partitions in housing train dataset using the statsmodels functions.

```
>>> # Define the function that we want to use to fit multiple GLM models, one
    # for each home style.
    >>> def glm_fit(rows):
        """
        DESCRIPTION:
            Function that accepts a iterator on a pandas DataFrame
            (TextFileObject) created using
            'chunk_size' with pandas.read_csv(), and fits a GLM model to it.
            The underlying data is the housing data with 12 independent
            variable (including the home style)
            and one dependent variable (price).

        RETURNS:
            A numpy.ndarray object with two elements:
            * The homestyle value (type: str)
            * The GLM model that was fit to the corresponding data, which is
            serialized using pickle
            and base64 encoded. We use decode() to make sure it is of type
            str, and not bytes.
        """
        # Read the entire partition/group of rows in a pandas DataFrame - pdf.
        data = rows.read()

        # Add the 'intercept' column along with the features.
        data['intercept'] = 1.0

        # Function does not process the partition if there are no rows here.
        if data.shape[0] > 0:
            # Fit the model using R-style formula to specify categorical
            variables as well.
            # Use 'disp=0' to prevent stderr output.
            model = smf.glm('price ~ C(recroom) + lotsize + stories + garagepl
+ C(gashw) + '
                                ' bedrooms + C(driveway) + C(airco) + C(homestyle)
+ bathrms +'
```

```

        ' C(fullbase) + C(prefarea)',
family=sm.families.Gaussian(), data=data).fit(dis=0)

        # Serialize and base64 encode the model in preparation to output it.
        modelSer = b64encode(dumps(model))

        # The user function can either return a value of supported type
        # (numpy array, Pandas Series, or Pandas DataFrame),
        # or just print it to find it's way to the output.
        # Here we return it as a numpy ndarray object.

        # Note that we use decode for the serialized model so that it is
        # represented in the ascii form (which is what base64
encoding does),
        # instead of bytes.
        return asarray([data.loc[0]['homestyle'], modelSer.decode('ascii')])

```

- b. Use the defined function "glm_fit()" to fit the model on group of the housing data where the grouping is done by the 'homestyle'.
 - a. Apply the "glm_fit" function defined in the previous step to create a model for every 'homestyle' in the training dataset. Specify the output column names and their types with the *returns* argument since the output is not similar to the input.

```

>>> model =
train.map_partition(glm_fit, data_partition_column='homestyle',

                    returns=OrderedDict([('homestyle',
train.homestyle.type),('model', CLOB())]))

```

- b. Print the model to show the model table has been created successfully.

```

>>> print(model.head())

                model
homestyle
Eclectic      gANjc3RhdHNtb2RlbHMuZ2VubW9kLmdlbmVyYWxpemVkX2...
Classic       gANjc3RhdHNtb2RlbHMuZ2VubW9kLmdlbmVyYWxpemVkX2...
bungalow     gANjc3RhdHNtb2RlbHMuZ2VubW9kLmdlbmVyYWxpemVkX2...

```

3. Scoring

- a. Use window functions to assign row numbers to each subset of data corresponding to a particular 'homestyle'. This is to extend the table to add the model corresponding to the 'homestyle' as the last column value for the first row in the partition, making it easier for the scoring function to read the model and then score the input records based on it.
- a. Create row number column ('row_id') in the 'test' DataFrame.

```

>>> test_with_row_num = test.assign(row_id =
func.row_number().over(partition_by=test.homestyle.expression,
order_by=test.sn.expression.desc()))

```


- b. Join it with the model created based on the value of 'homestyle'.

```
>>> temp = test_with_row_num.join(model, on
= [(test_with_row_num.homestyle == model.homestyle)],
rsuffix='r', lsuffix='l')
```

- c. Set the model column to NULL when row_id is not 1.

```
>>> temp = temp.assign(modeldata = case([(temp.row_id == 1,
literal_column(temp.model.name))], else_ = None))
```

- d. Drop the extraneous columns created in the processing.

```
>>> temp = temp.assign(homestyle
= temp.l_homestyle).drop('l_homestyle',
axis=1).drop('r_homestyle',axis=1).drop('model', axis=1)
```

- e. Reorder the columns to have the housing data columns positioned first, followed by the row_id and modeldata.

```
>>> new_test = temp.select(test.columns + ['row_id', 'modeldata'])
```

- b. Define the user function that will score the test data to predict the prices based on the features.

```
>>> DELIMITER = '\t'
>>> QUOTECHAR = None
```

```
>>> def glm_score(rows):
    """
    DESCRIPTION:
        Function that accepts a iterator on a pandas DataFrame
        (TextFileObject) created using
        'chunk_size' with pandas.read_csv(), and scores it based on the
        model found in the data.
        The underlying data is the housing data with 12 independent
        variable (including the home style)
        and one dependent variable (price).

        The function chooses to output the values itself, rather than
        returning objects of supported type.

    RETURNS:
        None.
    """
    model = None

    for chunk in rows:
        # Process data only if there is any, i.e. only when the chunk read
        has any rows.
        if chunk.shape[0] > 0:
            if model is None:
                # Read the model once (it is found only once)
                per partition.
                model = loads(b64decode(chunk.loc[0].iloc[-1]))

            # Exclude the row_id and modeldata columns from the scoring
            dataset as they are no longer required.
```

```

        chunk = chunk.iloc[:, :-2]

        # For prediction, exclude the first two column ('sn' - not
        relevant, and 'price' - the dependent variable).
        prediction = model.predict(chunk.iloc[:, 2:])

        # Concat the chunk with the prediction column (Pandas Series)
        to form a DataFrame.
        outdf = concat([chunk,
        prediction], axis=1)

        # We just cannot return this DataFrame yet as more chunks need
        to be processed.
        # In such scenarios, there are two options:
        # 1. print the output here, or
        # 2. keep concatenating the results of each chunk to create
        a final resultant pandas DataFrame to return.
        # In this example, use option #1 here.

        for _, row in outdf.iterrows():
            if QUOTECHAR is not None:
                # A NULL value should not be enclosed in quotes.
                # The CSV module has no support for such output with
                writer, and hence the custom formatting.
                values = ['' if isna(s) else "{}{}".format(QUOTECHAR, str(s), QUOTECHAR) for s in row]
            else:
                values = ['' if isna(s) else str(s) for s in row]
            print(DELIMITER.join(values), file=sys.stdout)

```

- c. Perform the actual scoring by calling the `map_partition()` method on the test data.

- a. Specify the output of the function. It has one more column than the input.

```

>>> returns = OrderedDict([(col.name, col.type) for col in
test._metaexpr.c] + [('prediction', FLOAT())])

```

- b. Scoring using `map_partition()`, using the `data_order_column` argument to order by the 'row_id' column so that the model is read before any data that need to be scored.

```

>>> prediction = new_test.map_partition(glm_score,
                                       returns=returns,

                                       data_partition_column='homestyle',

                                       data_order_column='row_id')

```

- c. Print the scoring result.

```

>>> print(prediction.head())

```

| sn | price | lotsize | bedrooms | garagepl | prefarea | homestyle | prediction | recroom | fullbase | gashw | airco |
|-----|---------|---------|----------|----------|----------|--------------|------------|---------|----------|-------|-------|
| 469 | 55000.0 | 2176.0 | 2 | yes | Eclectic | 64597.746106 | yes | no | no | no | no |
| 301 | 55000.0 | 4080.0 | 2 | no | Eclectic | 54979.762152 | no | no | no | no | no |
| 463 | 49000.0 | 2610.0 | 3 | yes | Classic | 46515.461314 | no | yes | no | no | no |
| 177 | 70000.0 | 5400.0 | 4 | no | Eclectic | 63607.229642 | no | no | no | no | no |
| 38 | 67000.0 | 5170.0 | 3 | no | Eclectic | 78029.766193 | yes | no | no | no | yes |
| 13 | 27000.0 | 1700.0 | 3 | no | Classic | 39588.073581 | yes | no | no | no | no |
| 255 | 61000.0 | 4360.0 | 4 | no | Eclectic | 61320.393435 | yes | no | no | no | no |
| 53 | 68000.0 | 9166.0 | 2 | | | | yes | yes | no | yes | yes |

| | | | | | | | | | | | |
|-----|-----|---------|--------|----------|--------------|---|-----|-----|-----|----|----|
| 2 | 364 | 72000.0 | no | Eclectic | 76977.937496 | 2 | yes | yes | yes | no | no |
| 0 | 0 | 10700.0 | no | Eclectic | 80761.658291 | 1 | yes | no | no | no | no |
| 459 | 0 | 44555.0 | 2398.0 | Classic | 42921.671929 | 1 | yes | no | no | no | no |

4. Model generated on client, and scored on Analytics Database.

This step shows how a model generated locally on the client machine can be used to score data on Vantage.

- a. First, create a sklearn GLM model, and serialize and base64-encode it. Fit the model on the entire training housing data using the sklearn package on the client machine .

- a. Import the pandas module.

```
>>> import pandas as pd
```

- b. Read the housing_train.csv file (shipped with the teradataml package) into a pandas DataFrame.

```
>>> with open('<path on local machine to the housing data>\
housing_train.csv', 'r') as f:
    housing_train = pd.read_csv(f)
```

- c. Encode the categorical columns.

```
>>> replace_dict = {'driveway': {'yes': 1, 'no': 0}, 'recroom':
{'yes': 1, 'no': 0}, 'fullbase': {'yes': 1, 'no': 0}, 'gashw': {'yes':
1, 'no': 0}, 'airco': {'yes': 1, 'no': 0}, 'prefarea': {'yes': 1,
'no': 0}, 'homestyle': {'Classic': 1, 'Eclectic': 2, 'bungalow': 3}}
```

- d. Replace the values *inplace*.

```
>>> housing_train.replace(replace_dict, inplace=True)
```

- e. Fit the GLM model.

```
>>> model = LogisticRegression(max_iter=5000, solver='lbfgs',
multi_class='auto').fit(housing_train.iloc[:,2:], housing_train.price)
```

- f. Serialize and base64-encode the GLM model.

```
>>> modelSer = b64encode(dumps(model)).decode('ascii')
```

- b. Define a user function which accepts the model and the dataset to use it for scoring with map_partition().

```
>>> def glm_score_local_model(rows, model):
    """
    DESCRIPTION:
        Function that accepts a iterator on a pandas DataFrame
        (TextFileObject) created using
        'chunk_size' with pandas.read_csv(), and scores it based on the
        model passed to the function as
        the second argument.
```

```

    The underlying data is the housing data with 12 independent
    variable (including the home style)
    and one dependent variable (price).

    The function concatenates the result of all chunk scoring
    operation into a final pandas DataFrame to return.

    RETURNS:
    """ pandas DataFrame.
    # Decode and deserialize the model.
    model = loads(b64decode(model))
    result_df = None
    for chunk in rows:
        # Process data only when the chunk read has any rows.
        if chunk.shape[0] > 0:

            # Perform the encoding for the categorical columns.
            chunk.replace(replace_dict, inplace=True)
            # For prediction, exclude the first two column ('sn' - not
relevant, and 'price' - the dependent variable).
            prediction = pd.Series(model.predict(chunk.iloc[:,2:]))

            # Concat the chunk with the prediction column (Pandas Series)
to form a DataFrame.
            outdf = concat([chunk, prediction], axis=1)

            # We just cannot return this DataFrame yet as more chunks need
to be processed.
            # In such scenarios, there are two options:
            # 1. print the output here, or
            # 2. keep concatenating the results of each chunk to create
a final resultant Pandas DataFrame to return.

            # In this example, use option #2 here.
            if result_df is None:
                result_df = outdf
            else:
                result_df = concat([result_df, outdf], axis=0)

    # Return the result pandas DataFrame.
    return result_df

```

- c. Call the `map_partition()` method for the test data to score the data and predict the prices.
 - a. Specify the output of the function. It has one more column than the input.

```

>>> # Note that here the output of the function is going to have one
more column than the input,
>>> # and we must specify the same.
>>> returns = OrderedDict([(col.name, col.type) for col in
test._metaexpr.c] + [('prediction', FLOAT())])

```

- b. Scoring using `map_partition()`, using the `data_order_column` argument to order by the 'row_id' column so that the model is read before any data that need to be scored.

```

>>> prediction = test.map_partition(lambda rows:
glm_score_local_model(rows, modelSer),

```

```
returns=returns,
data_partition_column='homestyle')
```

- c. Print the scoring result.

```
>>> print(prediction.head())
      price  lotsize bedrooms  bathrms  stories driveway recroom fullbase gashw airco  garagepl
prefarea homestyle prediction
sn
25  42000.0   4960.0         2         1         1         1         0         0         0         0
0      0         1   50000.0
53  68000.0   9166.0         2         1         1         1         0         1         0         1
2      0         2   70000.0
111 43000.0   5076.0         3         1         1         0         0         0         0         0
0      0         1   50000.0
117 93000.0   3760.0         3         1         2         1         0         0         1         0
2      0         2   62000.0
140 43000.0   3750.0         3         1         2         1         0         0         0         0
0      0         1   50000.0
142 40000.0   2650.0         3         1         2         1         0         1         0         0
1      0         1   48000.0
157 60000.0   2953.0         3         1         2         1         0         1         0         1
0      0         2   52000.0
161 63900.0   3162.0         3         1         2         1         0         0         0         1
1      0         2   52000.0
176 57500.0   3630.0         3         2         2         1         0         0         1         0
2      0         2   60000.0
177 70000.0   5400.0         4         1         2         1         0         0         0         0
0      0         2   60000.0
```

Using PMMLPredict to Score using Externally Trained Models

This example uses the iris_input dataset and performs a prediction on each row of the input table using a model previously trained in PMML and then loaded into the database.

1. Set up the environment.
 - a. Import required libraries.

```
import tempfile
```

```
import getpass
```

```
from teradataml import PMMLPredict, DataFrame, load_example_data,
create_context, db_drop_table, remove_context, save_byom, delete_byom,
retrieve_byom, list_byom
```

```
from teradataml.options.configure import configure
```

- b. Create the connection to database.

```
con = create_context(host=getpass.getpass("Hostname: "),
                    username=getpass.getpass("Username: "),
                    password=getpass.getpass("Password: "))
```

- c. Load example data.

```
load_example_data("byom", "iris_input")
```

```
iris_input = DataFrame("iris_input")
```

2. Create train dataset and test dataset.

a. Create two samples of input data.

This step creates two samples of input data: sample 1 has 80% of total rows and sample 2 has 20% of total rows.

```
iris_sample = iris_input.sample(frac=[0.8, 0.2])
```

```
iris_sample
```

b. Create train dataset.

This step creates train dataset from sample 1 by filtering on "sampleid" and drop "sampleid" column as it is not required for training model.

```
iris_train = iris_sample[iris_sample.sampleid == "1"].drop("sampleid",
axis = 1)
```

```
iris_train
```

c. Create test dataset.

This step creates test dataset from sample 2 by filtering on "sampleid" and drop "sampleid" column as it is not required for scoring.

```
iris_test = iris_sample[iris_sample.sampleid == "2"].drop("sampleid",
axis = 1)
```

```
iris_test
```

3. Train the Random Forest model and perform the Prediction using PMMLPredict().

a. Import required libraries.

```
import numpy as np
```

```
from sklearn import tree
```

```
from nyoka import skl_to_pmml
```

```
from sklearn.pipeline import Pipeline
```

```
from sklearn_pandas import DataFrameMapper
```

```
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.ensemble import RandomForestClassifier
```

- b. Prepare dataset to create a Random Forest model.

```
traid_pd = iris_train.to_pandas()
features = traid_pd.columns.drop('species')
target = 'species'
```

- c. Generate the Random Forest model.

```
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
```

```
rf_pipe_obj = Pipeline([
    ("mapping", DataFrameMapper([
        (['sepal_length', 'sepal_width'], StandardScaler()),
        (['petal_length', 'petal_width'], imputer)
    ])),
    ("rfc", RandomForestClassifier(n_estimators = 100))
])
```

```
rf_pipe_obj.fit(traid_pd[features], traid_pd[target])
```

- d. Save the model in PMML format.

```
temp_dir = tempfile.TemporaryDirectory()
```

```
model_file_path = f"{temp_dir.name}/iris_rf_class_model.pmml"
```

```
skl_to_pmml(rf_pipe_obj, features, target, model_file_path)
```

- e. Save the model in Vantage.

```
save_byom("pmml_random_forest_iris", model_file_path, "byom_models")
```

- f. List the model from Vantage.

```
list_byom("byom_models")
```

- g. Retrieve the model from Vantage.

```
modeldata = retrieve_byom("pmml_random_forest_iris", "byom_models")
```

- h. Score the test data using PMMLPredict function with the retrieved model.

```
result = PMMLPredict(
    modeldata = modeldata,
    newdata = iris_test,
```

```
accumulate = ['id', 'sepal_length', 'petal_length'],
overwrite_cached_models = '*',
)
```

- i. Print the equivalent SQL query and Score result.

```
print(result.show_query())
```

```
result.result
```

4. Clean up.

```
# Delete the model from table "byom_models", using the model
id 'pmml_random_forest_iris'.
delete_byom("pmml_random_forest_iris", "byom_models")
```

```
# Drop models table.
db_drop_table("byom_models")
```

```
# Drop input data tables.
db_drop_table("iris_input")
```

```
# One must run remove_context() to close the connection and garbage collect
internally generated objects.
remove_context()
```

Using H2OPredict to Score using Externally Trained Models

This example uses the iris_input dataset and performs a prediction on each row of the input table using a model previously trained in H2O and then loaded into the database.

1. Set up the environment.
 - a. Import required libraries.

```
import tempfile
```

```
import getpass
```

```
from terdataml import H2OPredict, DataFrame, load_example_data,
create_context, db_drop_table, remove_context, save_byom, delete_byom,
retrieve_byom, list_byom
```

```
from terdataml.options.configure import configure
```

- b. Create the connection to database.


```
con = create_context(host=getpass.getpass("Hostname: "),
                    username=getpass.getpass("Username: "),
                    password=getpass.getpass("Password: "))
```

- c. Load example data.

```
load_example_data("byom", "iris_input")
```

```
iris_input = DataFrame("iris_input")
```

2. Create train dataset and test dataset.

- a. Create two samples of input data.

This step creates two samples of input data: sample 1 has 80% of total rows and sample 2 has 20% of total rows.

```
iris_sample = iris_input.sample(frac=[0.8, 0.2])
```

```
iris_sample
```

- b. Create train dataset.

This step creates train dataset from sample 1 by filtering on "sampleid" and dropping "sampleid" column as it is not required for training model.

```
iris_train = iris_sample[iris_sample.sampleid == "1"].drop("sampleid",
axis = 1)
```

```
iris_train
```

- c. Create test dataset.

This step creates test dataset from sample 2 by filtering on "sampleid" and dropping "sampleid" column as it is not required for scoring.

```
iris_test = iris_sample[iris_sample.sampleid == "2"].drop("sampleid",
axis = 1)
```

```
iris_test
```

3. Train the Gradient Boosting Machine model and perform the Prediction using H2OPredict().

- a. Import required libraries.

```
import h2o
```

```
from h2o.estimators import H2OGradientBoostingEstimator
```

- b. Prepare dataset to create a Gradient Boosting Machine model.

Converting teradataml DataFrame to pandas DataFrame, since H2OFrame accepts pandas DataFrame.

```
h2o.init()

iris_train_pd = iris_train.to_pandas()
h2o_df = h2o.H2OFrame(iris_train_pd)
h2o_df
```

- c. Train the Gradient Boosting Machine model.

Add the code for training model.

```
h2o_df["species"] = h2o_df["species"].asfactor()
predictors = h2o_df.columns
response = "species"

gbm_model = H2OGradientBoostingEstimator(nfolds=5, seed=1111,
keep_cross_validation_predictions = True)

gbm_model.train(x=predictors, y=response, training_frame=h2o_df)
```

- d. Save the model to a file in MOJO format.

```
temp_dir = tempfile.TemporaryDirectory()
model_file_path = gbm_model.save_mojo(path=f"{temp_dir.name}", force=True)
```

- e. Save the model in Vantage.

```
save_byom(model_id="h2o_gbm_iris",
model_file=model_file_path, table_name="byom_models")
```

- f. List the model in Vantage.

```
list_byom("byom_models")
```

- g. Retrieve the model from Vantage.

```
model=retrieve_byom("h2o_gbm_iris", "byom_models")
```

- h. Set "configure.byom_install_location" to the database where BYOM functions are installed.

```
configure.byom_install_location =
getpass.getpass("byom_install_location: ")
```

- i. Score the test data using H2OPredict function with the retrieved model.

```
result = H2OPredict(newdata=iris_test,
                    newdata_partition_column='id',
                    newdata_order_column='id',
                    modeldata=model,
                    modeldata_order_column='model_id',
                    model_output_fields=['label', 'classProbabilities'],
                    accumulate=['id', 'sepal_length', 'petal_length'],
                    overwrite_cached_models='*',
                    enable_options='stageProbabilities',
                    model_type='OpenSource'
                )
```

- j. Print the equivalent SQL query and Score result.

```
print(result.show_query())
```

```
result.result
```

4. Clean up.

```
# Delete the saved Model.
delete_byom("h2o_gbm_iris", table_name="byom_models")
```

```
# Drop models table.
db_drop_table("byom_models")
```

```
# Drop input data tables.
db_drop_table("iris_input")
```

```
# One must run remove_context() to close the connection and garbage collect
internally generated objects.
remove_context()
```

Using ONNXPredict to Score using Externally Trained Models

This example uses the iris_input dataset and performs a prediction on each row of the input table using a model trained in ONNX format and then loaded into database.

1. Set up the environment.
 - a. Import required libraries.

```
import tempfile
```

```
import getpass
```

```
from terdataml import DataFrame, load_example_data,
create_context, db_drop_table, remove_context, save_byom, delete_byom,
retrieve_byom, list_byom
```

```
from terdataml.options.configure import configure
```

- b. Create the connection to database.

```
con = create_context(host=getpass.getpass("Hostname: "),
                    username=getpass.getpass("Username: "),
                    password=getpass.getpass("Password: "))
```

- c. Load example data.

```
load_example_data("byom", "iris_input")
```

```
iris_input = DataFrame("iris_input")
```

2. Create train dataset and test dataset.

- a. Create two samples of input data.

This step creates two samples of input data: sample 1 has 80% of total rows and sample 2 has 20% of total rows.

```
iris_sample = iris_input.sample(frac=[0.8, 0.2])
```

```
iris_sample
```

- b. Create train dataset.

This step creates train dataset from sample 1 by filtering on "sampleid" and dropping "sampleid" column as it is not required for training model.

```
iris_train = iris_sample[iris_sample.sampleid == "1"].drop("sampleid",
axis = 1)
```

```
iris_train
```

- c. Create test dataset.

This step creates test dataset from sample 2 by filtering on "sampleid" and dropping "sampleid" column as it is not required for scoring.

```
iris_test = iris_sample[iris_sample.sampleid == "2"].drop("sampleid",
axis = 1)
```

```
iris_test
```

3. Train the Random Forest model and perform the Prediction using ONNXPredict().

- a. Import required libraries.

```
from teradataml import ONNXPredict

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestClassifier
```

- b. Prepare dataset for training Random Forest model.

Convert teradataml dataframe to pandas dataframe.

```
train_pd = iris_train.to_pandas()
features = train_pd.columns.drop('species')
target = 'species'
```

- c. Generate Random Forest model.

```
rf_pipe_obj = Pipeline([
    ('scaler', StandardScaler()),
    ("rf", RandomForestClassifier(max_depth=5))
])
```

- d. Train the Random Forest model.

```
rf_pipe_obj.fit(train_pd[features], train_pd[target])
```

- e. Save the model in ONNX format.

- a. Create temporary file path to save the model.

```
temp_dir = tempfile.TemporaryDirectory()

model_file_path = f"{temp_dir.name}/iris_db_rf_model.onnx"
```

- b. Convert and save the Random Forest model in ONNX format.

```
onx = to_onnx(rf_pipe_obj, train_pd.iloc[:, :4].astype(np.float32))

with open(model_file_path, "wb") as f:
    f.write(onx.SerializeToString())
```

- f. Save the model in Vantage.

```
save_byom("onnx_rf_iris", model_file_path, "byom_models")
```

- g. List the model in Vantage.

```
list_byom("byom_models")
```

- h. Retrieve the model from table "byom_models", using the model id 'onnx_rf_iris'.

```
modeldata = retrieve_byom("onnx_rf_iris", "byom_models")
```

- i. Set "configure.byom_install_location" to the database where BYOM functions are installed.

```
configure.byom_install_location =  
getpass.getpass("byom_install_location: ")
```

- j. Perform prediction using ONNXPredict() function and the ONNX model stored in Vantage.

```
predict_output = ONNXPredict(  
    modeldata = modeldata,  
    newdata = iris_test,  
    accumulate = ['id',  
'sepal_length', 'petal_length'],  
    overwrite_cached_models = '*',  
    model_output_fields = "output_label"  
)
```

- k. Print the equivalent SQL query and Score result.

```
print(result.show_query())
```

```
result.result
```

4. Clean up.

- a. Delete the model from table "byom_models".

```
delete_byom("onnx_rf_iris", "byom_models")
```

- b. Delete models table.

```
db_drop_table("byom_models")
```

- c. Drop input data table.

```
db_drop_table("iris_input")
```

- d. Remove context.

```
remove_context()
```

Using Open Analytics to Score using Externally Trained Models using Apply

This example uses Open Analytics to score using externally trained models using Apply.

Note:

This example works only on VantageCloud Lake.

1. Set up the environment.
 - a. Import required libraries.

```
from terdataml import create_context, remove_context, list_base_envs,
list_user_envs, create_env, remove_env, get_env, DataFrame, copy_to_sql,
Apply, configure, read_csv, set_config_params
```

```
from terdataml.options.display import display
```

```
import pandas as pd, getpass, os
```

```
from collections import OrderedDict
```

```
from terdatasqlalchemy.types import BIGINT, VARCHAR, INTEGER, FLOAT
```

- b. Set Authentication token and UES URL.

```
set_config_params(ues_url=getpass.getpass("UES URL: "),
                  auth_token=getpass.getpass("JWT Token: "))
```

- c. Create the connection.

```
con = create_context(host=getpass.getpass("Hostname: "),
                    username=getpass.getpass("Username: "),
                    password=getpass.getpass("Password: "))
```

Note:

You can use the same JWT token instead of password to create a context. See [create_context](#) for more details.

2. Generate model.
 - a. Import required libraries.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
```

- b. Read the data from the scikit-learn package.

```
iris = load_iris()
X, y = iris.data, iris.target
```

- c. Train a model with Random Forests.

```
X_train, X_test, y_train, y_test = train_test_split(X, y)
clr = RandomForestClassifier()
clr.fit(X_train, y_train)
```

- d. Convert the model into ONNX format. Generate ONNX model file "rf_iris.onnx".

```
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

initial_type = [('float_input', FloatTensorType([None, 4]))]
onx = convert_sklearn(clr, initial_types = initial_type)
with open("rf_iris.onnx", "wb") as f:
    f.write(onx.SerializeToString())

print("RF model trained and saved in 'rf_iris.onnx'.")
```

3. Load test data into VantageCloud Lake and create teradataml dataframe for the input table.

```
dfIn = pd.DataFrame(X_test, columns=["sepal_length", "sepal_width",
    "petal_length", "petal_width"])
copy_to_sql(dfIn, table_name = 'onnx_test_table_dataset', if_exists
    = 'replace')
```

```
onnx_test_data = DataFrame.from_table("onnx_test_table_dataset")
onnx_test_data.head(n=5)
```

4. Create a python file to score the model.

Create a file with the name 'sklearn_onnx_scoring.py' in local client with following code.

```
# Train a model.
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
import csv
import sys
```



```

# Read input data from stdin into a dataframe.
_reader = csv.DictReader(sys.stdin.readlines(), fieldnames
= ["sepal_length", "sepal_width", "petal_length", "petal_width"])
data=pd.DataFrame(_reader, columns
= ["sepal_length", "sepal_width", "petal_length", "petal_width"])

# For AMPs that receive no data, exit the script instance gracefully.
if data.empty:
    sys.exit()

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y)
clr = RandomForestClassifier()
clr.fit(X_train, y_train)

# Compute the prediction with ONNX Runtime
import onnxruntime as rt
import numpy
sess = rt.InferenceSession("rf_iris.onnx")
input_name = sess.get_inputs()[0].name
label_name = sess.get_outputs()[0].name
pred_onx = sess.run([label_name], {input_name:
data.values.astype(numpy.float32)})[0]

listToStr = ' '.join([str(elem) for elem in pred_onx])

print(listToStr)

```

5. Create Environment and install the corresponding files in the environment.

a. List the base Python environments.

```
list_base_envs()
```

Assume a new Python environment is needed.

b. Create a new Python user environment for Python 3.8.13.

Function `create_env()` will return an object of 'UserEnv'.

```

demo_env = create_env(env_name = 'oaf_usecase_2c_env',
                      base_env = 'python_3.8.13',
                      desc = 'OAF Demo Use Case 2c Environment')

```

c. Verify the new environment has been created.

```
list_user_envs()
```

- d. Install necessary Python add-ons synchronously, for use by the script in the user environment using an object 'demo_env' of class "UserEnv".

```
demo_env.install_lib(["skl2onnx", "sklearn", "onnxruntime", "pandas"])
```

- e. Verify the Python libraries have been installed correctly.

```
demo_env.libs
```

- f. Install the model file and Python file to score the data inside VantageCloud Lake.

```
demo_env.install_file(file_path = 'rf_iris.onnx', replace = True)
demo_env.install_file(file_path = 'sklearn_onnx_scoring.py', replace
= True)
```

- g. Verify the files have been installed correctly.

```
demo_env.files
```

6. Score the data inside VantageCloud Lake.

- a. Use Apply to create an object for the Random Forest based prediction.

```
applyRF_obj = Apply(data = onnx_test_data,
                    apply_command = 'python3 sklearn_onnx_scoring.py',
                    returns = {"Predicted_Class_RF": VARCHAR(200)},
                    env_name = demo_env
                    )
```

- b. Run the Python script inside the remote user environment.

```
applyRF_obj.execute_script()
```

Note:

You can display the underlying SQL by setting 'display.print_sqlmr_query = True'.

7. Remove the environment and disconnect from VantageCloud Lake.

- a. After scoring the data, remove the environment.

```
remove_env('oaf_usecase_2c_env')
```

- b. Verify the specified environment has been removed.

```
list_user_envs()
```

- c. Disconnect from VantageCloud Lake.

```
remove_context()
```

Teradata Package for Python Function Reference

The Teradata Package for Python, `teradataml`, includes functions for data management, exploration, preparation and analytics in Vantage.

See [Teradata Package for Python Function Reference](https://docs.teradata.com/) (*Teradata Package for Python Function Reference*, B700-4008) at <https://docs.teradata.com/> for detailed description and usage examples of the `teradataml` functions.

Teradata Package for Python Limitations and Considerations

Teradata Package for Python Limitations

This section provides information on limitations of the Teradata Package for Python.

DataFrame Creation on Primary Time Index Tables is Partially Supported

DataFrame creation on Primary Time Index (PTI) tables is partially supported. DataFrame is created when the underlying PTI table is not created with 'timebucket' duration.

Example

Create a PTI table without 'timebucket_duration', using copy_to_sql and then create a DataFrame on it.

```
>>> load_example_data("sessionize", "sessionize_table")
>>> df3 = DataFrame('sessionize_table')
>>> copy_to_sql(df3, "test_copyto_pti",
                timecode_column='clicktime',
                columns_list='event')
```

```
>>> DataFrame("test_copyto_pti")
      TD_TIMECODE  partition_id  adid  productid
event
click  2009-03-19 16:43:26.000000    1199      1      1001
click  2009-07-04 09:18:17.000000    1231      1      1001
click  2009-07-04 09:18:17.000000    1231      1      1001
click  2009-07-16 11:18:16.000000    1039      4      1001
click  2009-07-16 11:18:16.000000    1039      4      1001
click  2009-07-24 04:18:10.000000    1167      2      1001
view   2009-02-09 15:17:59.000000    1263      4      1001
view   2009-03-09 21:17:59.000000    1199      2      1001
view   2009-03-09 21:17:59.000000    1199      2      1001
view   2009-03-13 17:17:59.000000    1071      4      1001
```

Example

Create a PTI table with 'timebucket_duration', DataFrame creation on the PTI table fails.

```
>>> load_example_data("sessionize", "sessionize_table")
>>> df3 = DataFrame('sessionize_table')
>>> copy_to_sql(df3, "test_copyto_pti1",
                timecode_column='clicktime',
```

```

        columns_list='event',
        timebucket_duration="HOURS(2)")

>>> DataFrame("test_copyto_pti1")
/anaconda3/lib/python3.6/site-packages/sqlalchemy/engine/reflection.py:888:
SAWarning: index key 'TD_TIMEBUCKET' was not located in columns for
table 'test_copyto_pti1'
"columns for table '%s'" % (flavor, c, table_name)
Traceback (most recent call last):
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/dataframe/
dataframe.py", line 163, in __init__
    self.metaexpr = self._get_metaexpr()
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/dataframe/
dataframe.py", line 395, in _get_metaexpr
    return _MetaExpression(t, column_order = self.columns)
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/dataframe/
sql.py", line 164, in __init__
    self._t = _SQLTableExpression(table, **kw)
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/dataframe/
sql.py", line 460, in __init__
    raise ValueError('Reflected column names do not match those in DataFrame.columns')
ValueError: Reflected column names do not match those in DataFrame.columns

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/dataframe/
dataframe.py", line 171, in __init__
    raise TeradataMLException(Messages.get_message(MessageCodes.TDMLDF_CREATE_FAIL),
MessageCodes.TDMLDF_CREATE_FAIL) from err
teradataml.common.exceptions.TeradataMLException: [Teradata][teradataml](TDML_2010)
Failed to create Teradata DataFrame.

```

copy_to_sql and to_sql Functions Limitation

UTF-8 and Unicode support

UTF-8 and Unicode characters are not supported by the `copy_to_sql` function.

To work around this limitation, convert UTF-8 and Unicode characters to ASCII and then push the data into the `teradataml` DataFrame.

Multithreading and Concurrent Usage Limitations

The Teradata Package for Python does not support multithreading and concurrent usage of the following operations:

- Analytic functions for model creations
For example, calling an analytic function, such as `glm`, `kmeans` and so forth, from multiple threads by passing different functional parameters to the same API in each thread.
- Analytic functions for prediction
For example, calling an analytic predict function, such as `glmpredict`, from multiple threads by passing different functional parameters to the same API in each thread.
- Transformations of data frames

For example, performing transformation on the same teradataml DataFrame from multiple threads.

print(dataframe.dtypes) Displays Column Types as String

When a teradataml DataFrame is created from a table with the following column data types, the `print(dataframe.dtypes)` displays the column types as String:

- Interval (Year, Month, Day, Hour)
- Period (Date, Time)
- CLOB

Error Calling teradataml Analytic Functions

When Connected to Teradata Vantage with Analytics Database Only

If you connect to Vantage without a ML Engine installed and request to run teradataml analytic functions that execute on ML Engine, the system will return an error.

For example:

```
TeradataMLException: [Teradata][teradataml](TDML_2102) Failed to execute SQL:
teradatasql.OperationalError: [Version 16.20.0.32] [Session 76702] [Teradata
Database] [Error 3707]
Syntax error, expected something like ')' between the word 'ConfusionMatrix'
and '('.'")'
```

The Teradata Package for Python is fully featured when connected to Vantage with Analytics Database and ML Engine.

When connecting to Vantage with Analytics Database only, the only analytic functions available to the teradataml users are the ones that execute on the Analytics Database.

To invoke and use teradataml analytic functions available on the ML Engine, your Vantage system must consist of both Analytics Database and ML Engine.

Using Function with New Arguments on Vantage1.0

Following functions are updated by adding new arguments.

| Analytic Functions | Newly added arguments (Only supported on Vantage 1.1 or later) |
|--------------------|---|
| AdaBoost | categorical_encoding |
| DecisionForest | categorical_encoding |
| DecisionTree | categorical_encoding |
| KNN | 'accumulate' and 'output_prob' |
| RandomSample | setid_as_first_column |

| Analytic Functions | Newly added arguments (Only supported on Vantage 1.1 or later) |
|--------------------|---|
| VarMax | 'order_p', 'order_d', 'order_q', 'seasonal_order_p', 'seasonal_order_d', 'seasonal_order_q' |

Note:

These arguments are supported only on Vantage 1.1 or later. If used with Vantage 1.0, the following error will show:

```
[Teradata Database] [Error 4382] Argument {argument-name} is not defined in the function mapping definition
```

Using Sampling Function on Vantage1.0

Using Sampling function on Vantage 1.0 will result in following error. This function is now only supported on Vantage1.1 or later.

```
[Teradata Database] [Error 4381] Only one ON clause with either undefined or empty correlation name can be mapped to ANY in function mapping definition.
```

select() Method Requires Unique Column Names

The select method does not handle identical column names being passed as arguments.

For example:

```
>>> df = DataFrame('table1')
>>> df.select(['col1', 'col1'])
```

To work around, use the assign method to provide an alias.

```
# c1 and c2 both refer to the col1 column
>>> df.assign(c1 = df.col1, c2 = df.col1)
```

Limited Missing Value Support

NaN and +/- Inf values can arise in floating point calculations. They are rendered when a DataFrame is evaluated.

```
>>> df
      value
row_id
```



```

2          -inf
1           inf
3          NaN

>>> df.dtypes
row_id      str
value      float

```

NaN and +/- Inf values are not supported as *missing values*. Particularly, there is no support to reference these values in the Analytics Database. Only the NULL value is supported as a *missing value*, in which case they are usually rendered as None. Floating point columns with NULL values can be rendered as NaN. In this case, NaN is recognized as a *missing value*.

```

>>> df[df.value.isna() == True]

      value
row_id
3       None

>>> df[df.value.isna() == False]

      value
row_id
2       -inf
1        inf

```

DataFrames Become Invalid When a Connection Context is Recreated

On recreating a connection context using the `create_context` command, most of the operations on the DataFrames created using an earlier connection context fail.

The users receive a warning when the `create_context` command is rerun.

For example:

```

>>> create_context(host = "myhostname", username="myusername", password
= "mypassword")
>>> df = DataFrame('table1')

>>> create_context(host = "myhostname", username="myusername", password
= "mypassword")

```

```
UserWarning: [Teradata][teradataml](TDML_2002) Overwriting an existing context
associated with Teradata connection. Most of the operations on any teradataml
DataFrames created before this will not work.
```

```
>>> df.select(['model','phrase_id'])
TeradataMLException: [Teradata][teradataml](TDML_2103) Internal Error: Non-zero
status returned from AED.
```

To work around this limitation, recreate the DataFrames with the new connection context.

For example, recreate the 'df' DataFrame after the new connection context is created in the previous example.

```
>>> df = DataFrame('table1')
>>> df.select(['model','phrase_id'])
   phrase_id
model
1           1
1           2
1           2
1           1
1           1
1           1
(6, 3)
```

Error Using responses Argument in MLE Function NaïveBayesTextClassifierPredict

The 'responses' argument in the ML Engine NaïveBayesTextClassifierPredict function (teradataml.mle.NaiveBayesTextClassifierPredict) is not supported in this release. Using this argument will result in an error.

For example:

```
>>> nbt_predict_out = NaiveBayesTextClassifierPredict(object = nbt_out,
                                                    newdata =
complaints_tokens_test,
                                                    input_token_column =
'token',
                                                    doc_id_columns = 'doc_id',
                                                    model_type = "Bernoulli",
                                                    model_token_column =
'token',
```

```

'category',
'prob',
'doc_id',

responses=['crash','no_crash'])

model_category_column =
model_prob_column =
newdata_partition_column =

[Teradata Database] [Error 4382] Argument Responses is not defined in the function
mapping definition.
```

copy_to API When Loading Data into PTI Table

When the copy_to API fails with the Database error message “4358 Time Series: TD_TIMECODE|TD_SEQNO out of range for table”, you must check the permitted TD_TIMECODE and TD_SEQNO range using the following:

```

con = get_connection().connection
cur = con.cursor()
td_cur = cur.execute("exec DBC.TD_TIMESERIES_RANGE('MyDB.<table_name>')")
td_cur.fetchall()
```

Where:

- *MyDB* is the schema name
- *<table_name>* is the name of the table to be created

And correct the data or the arguments like "seq.max" to suit the data.

Note:

When the data insertion fails, the table is already created.

If a table is to be created with the same table name after the error is handled, the existing table must be dropped using the argument "if_exists = replace".

Teradata Package for Python Considerations

This section provides information on topics that impact the use of the Teradata Package for Python.

Dynamic Imports for Analytics Database, BYOM, ReadNOS, WriteNOS and UAF Functions

teradataml generates the following functions dynamically based on the Vantage database it is connected to.

- BYOM functions
- Analytics Database functions
- ReadNOS and WriteNOS functions
- UAF functions

If the database does not support any of these functions, then teradataml do not expose these functions. Hence all of these functions are available only after establishing a connection to Vantage.

You should import these functions after context creation (see [create_context](#)). These functions may raise error or may not work properly if imported before establishing a connection to Vantage.

Example 1: Import the function before context creation

An error is raised.

```
>>> from teradataml import CategoricalSummary
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'CategoricalSummary'
```

Example 2: Import the function after establishing connection to Vantage

Function is successfully imported.

```
>>> from teradataml import create_context
>>> create_context(host=host, user = user, password=password)
Engine(teradatasql://:***@host/?USER=user)

>>> from teradataml import CategoricalSummary
>>>
```

User Permissions in Vantage

To operate and interact with Vantage with the Teradata Package for Python, the user must have a series of permissions granted. Otherwise, executing ML Engine analytic functions can result in SQL errors on the Python client that stem from inadequate database user permissions.

A database user must be granted in advance the following permissions by the database administrator before using the teradataml package.

- GRANT EXECUTE FUNCTION ON SYSLIB TO user;
- GRANT CONNECT THROUGH proxyuser TO PERMANENT user WITHOUT ROLE;
- GRANT SELECT ON TD_SERVER_DB.coprocessor TO user;
- GRANT INSERT ON TD_SERVER_DB.coprocessor TO user;
- GRANT EXECUTE FUNCTION ON TD_SERVER_DB.coprocessor TO user;
- GRANT CREATE SERVER ON TD_SERVER_DB TO user;
- GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.QGEXECUTEFOREIGNQUERY TO user;
- GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.QGINITIATOREXPORT TO user;
- GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.QGINITIATORIMPORT TO user;
- GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.QGREMOTEEXPORT TO user;
- GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.QGREMOTEIMPORT TO user;
- GRANT CTCONTROL ON user TO proxy_user.

Note:

The proxyuser is a suitable database proxy user for the analytic functions as determined by the database administrator.

Users must be granted the SELECT privilege to the following Data Dictionary views:

- DBC.DatabasesV
- DBC.TablesV
- DBC.ColumnsV
- DBC.UsersV
- DBC.Indices
- DBC.DBCInfoV

teradataml requires that the user has certain permissions on the user's default database or the initial default database specified using the *database* argument, or the temporary database when specified using *temp_database_name*.

These permissions allow the user to:

- Create tables and views to save results of teradataml analytic functions;
- Create views in the background for results of DataFrame APIs such as 'assign()', 'filter()', and so on, whenever the result for these APIs are accessed using a 'print()';
- Create view in the background on the query passed to the 'DataFrame.from_query()' API.

It is expected that the user has the required permissions to create these objects in the database that will be used.

For views based on Vantage analytic functions, additional permissions may be required, which can be granted using:

GRANT EXECUTE FUNCTION ON SYSLIB ... WITH GRANT OPTION

For example:

A user named ALICE connects to a non-default database named TOM to run the NamedEntityFinder function.

```
>>> # Load example data.
>>> load_example_data("namedentityfinder",
['assortedtext_input', 'name_Find_configure'])

>>> # Create teradataml DataFrame objects.
>>> nameFind_configure = DataFrame.from_table("name_Find_configure")
>>> assortedtext_input = DataFrame.from_table("assortedtext_input")

>>> # Find entities using a configuration table containing model items.
>>> NamedEntityFinder_out = NamedEntityFinder(newdata = assortedtext_input,
                                              configure_table_data =
nameFind_configure,
                                              text_column = 'content',
                                              accumulate = ['id', 'source'],
                                              entity_column = 'entity',
                                              model = 'all',
                                              show_entity_context = 0,
                                              newdata_sequence_column = 'id',
                                              configure_table_data_sequence_column='model_file')

teradata.sql.OperationalError: [Version 17.0.0.2] [Session 37706] [Teradata
Database] [Error 3523] An owner referenced by user does not have EXECUTE FUNCTION
WITH GRANT OPTION access to SYSLIB.NamedEntityFinder.
```

In order for this to work, the database TOM needs a certain permission, which can be granted using:

```
GRANT EXECUTE FUNCTION ON SYSLIB TO TOM WITH GRANT OPTION;
```

To let the function create the output views, or for the user to access such created views, additional permissions may be required depending on which database is used and which object the view being created is based on, and can be granted using:

```
GRANT SELECT ... WITH GRANT OPTION
```

For Example:

A user named ALICE connects to a non-default database named TOM to run the NamedEntityFinder function.

```
>>> # Connect to the default database to load the example dataset.
>>> con = create_context(host="myhostname",
username="myusername", password="mypassword")
```

```

>>> load_example_data("namedentityfinder",
['assortedtext_input', 'name_Find_configure'])

>>> # Reconnect to make sure all writes here on happen to the database
specified using 'temp_database_name'.
>>> remove_context()
>>> con = create_context(host="myhostname", username="myusername",
password="mypassword", temp_database_name="tom")

>>> # Create teradataml DataFrame objects.
>>> nameFind_configure = DataFrame.from_table("name_Find_configure")
>>> assortedtext_input = DataFrame.from_table("assortedtext_input")

>>> # Find entities using a configuration table containing model items.
>>> NamedEntityFinder_out = NamedEntityFinder(newdata = assortedtext_input,
                                              configure_table_data =
nameFind_configure,
                                              text_column = 'content',
                                              accumulate = ['id', 'source'],
                                              entity_column = 'entity',
                                              model = 'all',
                                              show_entity_context = 0,
                                              newdata_sequence_column = 'id',
                                              configure_table_data_sequence_column='model_file')

teradata.sql.OperationalError: [Version 17.0.0.2] [Session 37708] [Teradata
Database] [Error 3523] An owner referenced by user does not have SELECT WITH GRANT
OPTION access to alice.assortedtext_input.

```

In order for this to work, the following permission must be granted:

```

GRANT SELECT ON ALICE.assortedtext_input TO TOM WITH GRANT OPTION;

GRANT SELECT ON ALICE.name_Find_configure TO TOM WITH GRANT OPTION;

```

Persistence of Tables Created by the teradataml Package in Vantage

When users run a ML Engine analytic function, results are stored as tables in the database that is specified in the Vantage connection.

However, not all of these resulting tables may be persistent (in permanent storage) in the connection database. Specifically, tables that store models produced by analytic functions are non-persistent work tables (temporary tables).

The difference is that tables in permanent storage persist across different sessions, whereas temporary tables are automatically dropped at the end of a session.

Therefore, if the user establishes a Vantage connection in Python and calls an analytic function that creates an analytic model table in the database, when the user eliminates the connection, the database session will be terminated and the model table will be automatically dropped from the database.

To preserve a non-persistent model table created by `teradataml`, use the `copy_to` function with the model as a table object input to the function, before disconnecting from the session where the model table was created.

Filtering with Boolean Column Requires a Comparison

The Analytics Database does not support the Boolean data type. So the values '0' and '1' are used instead. To be consistent with pandas, '0' maps to 'False' and '1' maps to 'True'. This implies that the Boolean column in `teradataml` is numeric and behaves like a numeric column.

For example:

```
import teradataml
>>> df = df.assign(drop_columns = True,
                    Name= df.Name,
                    is_setosa = df.Name.str.contains('Setosa'))
```

```
>>> df
      Name is_setosa
0  Iris-versicolor      0
1   Iris-virginica      0
2  Iris-versicolor      0
3   Iris-virginica      0
4   Iris-virginica      0
5     Iris-setosa      1
6   Iris-virginica      0
7  Iris-versicolor      0
8     Iris-setosa      1
9  Iris-versicolor      0
```

```
>>> df[df.is_setosa == 1]
      Name is_setosa
0  Iris-setosa      1
1  Iris-setosa      1
2  Iris-setosa      1
```



```

3  Iris-setosa      1
4  Iris-setosa      1
5  Iris-setosa      1
6  Iris-setosa      1
7  Iris-setosa      1
8  Iris-setosa      1
9  Iris-setosa      1

```

Using Boolean literals 'True' and 'False' when comparing is supported.

For example:

```
>>> df[df.is_setosa == True]
```

```

      Name is_setosa
0  Iris-setosa      1
1  Iris-setosa      1
2  Iris-setosa      1
3  Iris-setosa      1
4  Iris-setosa      1
5  Iris-setosa      1
6  Iris-setosa      1
7  Iris-setosa      1
8  Iris-setosa      1
9  Iris-setosa      1

```

```
>>> df[df.is_setosa == False]
```

```

      Name is_setosa
0  Iris-versicolor      0
1  Iris-virginica      0
2  Iris-versicolor      0
3  Iris-virginica      0
4  Iris-virginica      0
5  Iris-virginica      0
6  Iris-versicolor      0
7  Iris-versicolor      0
8  Iris-virginica      0
9  Iris-versicolor      0

```

A comparison operator is needed whenever using a Boolean column, or else an error is thrown.

For example:

```
>>> df[df.is_setosa]
```

```
OperationalError: [Version 16.20.0.38] [Session 1961] [Teradata Database]
[Error 3707] Syntax error, expected something like a 'SUCCEEDS' keyword or a
'MEETS' keyword or a 'PRECEDES' keyword or an 'IN' keyword or a 'CONTAINS'
keyword between the word 'is_setosa' and ';'.
```

String Comparisons may be Case Insensitive

Comparing string literals when filtering with the teradataml DataFrame is not necessarily case sensitive.

All character data, except for CLOBs, accessed in the execution of a Teradata SQL statement has an attribute of CASESPECIFIC or NOT CASESPECIFIC, either by default or by explicit designation.

Character string comparisons use this attribute to determine whether the comparison is case blind or case specific. Case specificity does not apply to CLOBs.

Note:

For more information, see the Character String Comparisons section in the *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

For example:

```
>>> df.head(5)
```

| | SepalLength | SepalWidth | PetalLength | PetalWidth | Name |
|---|-------------|------------|-------------|------------|-------------|
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |

```
>>> df[df['Name'] == 'iris-SETOSA'].head(5)
```

| | SepalLength | SepalWidth | PetalLength | PetalWidth | Name |
|---|-------------|------------|-------------|------------|-------------|
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |

A workaround is to use the `str.contains` method with `case = True`.

```
>>> has_SETOSA = df['Name'].str.contains('iris-SETOSA', case = True)
>>> df[has_SETOSA == True]

Empty DataFrame
Columns: [SepalLength, SepalWidth, PetalLength, PetalWidth, Name]
Index: []
```

Teradata Package for Python Issues with VIEW Creation

SQL versus teradataml Differences

The main difference between SQL and teradataml is that teradataml internally creates temporary objects like views, and in order to do so the user needs explicit permissions whereas while running plain SQL none of these temporary objects are created.

teradataml requires that the user has certain permissions on the user's default database or the initial default database specified using the `database` argument, or the temporary database when specified using `temp_database_name`.

These permissions allow the user to do the following:

- Create tables and views to save results of teradataml analytic functions.
- Create views in the background for results of DataFrame APIs such as `assign()`, `filter()`, and so on, whenever the result for these APIs are accessed using a `print()`.
- Create view in the background on the query passed to the `DataFrame.from_query()` API.

It is expected that the user has the correct permissions to create these objects in the database that will be used. The access to the views created may also require issuing additional `GRANT SELECT ... WITH GRANT OPTION` permission depending on which database is used and which object the view being created is based on.

Note:

- User created in Teradata, by default has `CREATE VIEW` and `CREATE TABLE` permissions in own database.
 - Users created should have `PERM` space available to create tables.
 - `VIEW` creation does not require `PERM` space.
-

Issue while Creating a terdataml DataFrame from a Table in Non-Default Database

Description

terdataml user may encounter an error when the user tries to create a terdataml DataFrame from a table in a non-default database as "[Error 3524] The user does not have CREATE VIEW access to database [user_db]."

Cause

User has revoked CREATE VIEW for each user on the user's database.

Solution

Grant CREATE VIEW permission to the user who will work with zero PERM space.

```
GRANT CREATE VIEW ON <userid> TO <userid>
```

User can use configure command to set the userid:

```
>>> from terdataml import configure
>>> configure.temp_view_database = "<userid>"
```

Issue of not Having PERM SPACE to Create Temporary Tables

Description

terdataml user may encounter the limitation of not having PERM SPACE to create temporary tables. "[Error 3524] The user does not have CREATE VIEW access to database [user_db]."

Cause

User does not have PERM SPACE or not allowed to create temporary tables on the database, or both.

Solution

User can use configure command set the database or userid where they are allowed to create temporary tables.

```
>>> from terdataml import configure
>>> configure.temp_table_database = "<database> or <userid>"
```

Issue while Accessing a View in other Schema

Description

If a user created the context with the same default database, and tried to access the view in the other schema. The user will get an error as - "An owner referenced by user does not have SELECT WITH GRANT OPTION access to [non_default_database].[any_view]"

Cause

User created context with the default database.

Solution

Grant select to the USERNAME on the database where temporary objects are being created.

```
GRANT SELECT ON USERNAME TO <database> WITH GRANT OPTION;
```

teradataml allows user to specify different databases for creating tables and views. You can specify these databases using configuration options:

```
>>> from teradataml import configure
>>> configure.temp_table_database = "<database>"
>>> configure.temp_view_database = "<database>"
```

Use these options in cases when user has permissions to create tables in one database and view in other database.

For more details, see [temp_table_database and temp_view_database](#).

CREATE VIEW and SELECT WITH [non_default_database].[any_view] Issue Solution

Create the setup

- Create context using admin "dbc" user.

```
>>> from teradataml import *

>>> create_context(host="myhostname", username="dbc", password="dbc")
```

- Create required users and grant the required permissions: two users with PERM space and one user with no PERM space.

```
>>> get_connection().execute("create user alice_views as PERM = 100000000 ,
PASSWORD = \"alice\"")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD692630>
```

```
>>> get_connection().execute("create user user_with_perm as PERM =
100000000 , PASSWORD = \"alice\"")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD692A20>
```

```
>>> get_connection().execute("create user user_no_perm as PERM = 0 ,
PASSWORD = \"alice\"")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD692D68>
```

- Grant SELECT permission to users 'user_with_perm' and 'user_no_perm' on table 'titanic' in database 'alice'.

```
>>> get_connection().execute("grant select on alice.titanic
to user_no_perm;")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD6B37B8>
```

```
>>> get_connection().execute("grant select on alice.titanic
to user_with_perm;")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD6B3780>
```

- Grant CREATE VIEW permission to users 'user_with_perm' and 'user_no_perm' on database 'alice_view'.

```
>>> get_connection().execute("grant create view on alice_views
to user_no_perm;")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD6B3908>
```

```
>>> get_connection().execute("grant create view on alice_views
to user_with_perm;")
<sqlalchemy.engine.cursor.LegacyCursorResult object at 0x0000021EAD6B38D0>
```

Example 1: Case explaining user with no PERM space

- Create context using 'user_no_perm' user that does not have any perm space and pass "temp_database_name" as 'alice_view', so that internal objects will be created in 'alice_views'.

```
>>> create_context(host="myhostname", username="user_no_perm",
password="mypassword", temp_database_name="alice_views")
C:\Users\workspace\pyTeradata\teradataml\context\context.py:404:
UserWarning: [Teradata][teradataml](TDML_2002) Overwriting an existing
context associated with Teradata Vantage C
onnection. Most of the operations on any teradataml DataFrames created
before this will not work.
  warnings.warn(Messages.get_message(MessageCodes.OVERWRITE_CONTEXT))
```

- Create DataFrame using DataFrame.from_query(), this will result in error "An owner referenced by user does not have any WITH GRANT OPTION access to alice.titanic."

```
>>> df = DataFrame.from_query("select * from alice.titanic")
Traceback (most recent call last):
  File "C:\Users\.conda\envs\teradataml\lib\site-
packages\sqlalchemy\engine\base.py", line 1800, in _execute_context
    cursor, statement, parameters, context
  File "C:\Users\.conda\envs\teradataml\lib\site-
packages\sqlalchemy\engine\default.py", line 717, in do_execute
    cursor.execute(statement, parameters)
  File "C:\Users\.conda\envs\teradataml\lib\site-
packages\teradatasql\__init__.py", line 686, in execute
    self.executemany(sOperation, None, ignoreErrors)
  File "C:\Users\.conda\envs\teradataml\lib\site-
packages\teradatasql\__init__.py", line 933, in executemany
    raise OperationalError (sErr)
teradatasql.OperationalError: [Version 17.10.0.16] [Session 3829] [Teradata
Database] [Error 3523] An owner referenced by user does not have any WITH
GRANT OPTION access to alice.titanic.
  at gosqldriver/teradatasql.formatError ErrorUtil.go:88
  at gosqldriver/teradatasql.
(*teradataConnection).formatDatabaseError ErrorUtil.go:216
  at gosqldriver/teradatasql.
(*teradataConnection).makeChainedDatabaseError ErrorUtil.go:232
  at gosqldriver/teradatasql.
(*teradataConnection).processErrorParcel TeradataConnection.go:803
  at gosqldriver/teradatasql.
(*TeradataRows).processResponseBundle TeradataRows.go:2229
  at gosqldriver/teradatasql.
(*TeradataRows).executeSQLRequest TeradataRows.go:814
  at gosqldriver/teradatasql.newTeradataRows TeradataRows.go:673
  at gosqldriver/teradatasql.
(*teradataStatement).QueryContext TeradataStatement.go:122
  at gosqldriver/teradatasql.
(*teradataConnection).QueryContext TeradataConnection.go:1304
  at database/sql.ctxDriverQuery ctxutil.go:48
  at database/sql.(*DB).queryDC.func1 sql.go:1759
  at database/sql.withLock sql.go:3437
  at database/sql.(*DB).queryDC sql.go:1754
  at database/sql.(*Conn).QueryContext sql.go:2013
  at main.goCreateRows goside.go:666
  at _cgoexp_7cdf4597d74c_goCreateRows _cgo_gotypes.go:340
  at runtime.cgocallbackg1 cgocall.go:314
```

```
at runtime.cgocallbackg cgocall.go:233
at runtime.cgocallback asm_amd64.s:971
at runtime.goexit asm_amd64.s:1571
```

- Using 'configure' option from teradataml.
 - Set the 'temp_table_database' to 'alice_views' so that internal tables will be created in the same.

```
>>> configure.temp_table_database = "alice_views"
```

- Set the 'temp_view_database' to 'user_no_perm' so that internal views will be created in the user without perm space.

```
>>> configure.temp_view_database = "user_no_perm"
```

- Create DataFrame using DataFrame.from_query(), this will work fine.

```
>>> df = DataFrame.from_query("select * from alice.titanic")
```

```
>>> df
parch  SURVIVED  pclass  fare  cabin  embarked  name  sex  age  sibsp
PASSENGER  ticket
244      0      3      S      Maenpaa, Mr. Matti Alexanteri  male  22.0  0  0  STON/O
2. 3101275  7.1250  None
101      0      3      S      Petranec, Miss. Matilda  female  28.0  0
0      349245  7.8958  None
570      1      3      S      Jonsson, Mr. Carl  male  32.0  0
0      350417  7.8542  None
835      0      3      S      Allum, Mr. Owen George  male  18.0  0
0      2223  8.3000  None
692      1      3      S      Karun, Miss. Manca  female  4.0  0
1      349256  13.4167  None
284      1      3      C      Dorking, Mr. Edward Arthur  male  19.0  0  0
A/5. 10482  8.0500  None  S
427      1      2  Clarke, Mrs. Charles V (Ada Maria Winfield)  female  28.0  1
0      2003  26.0000  None  S
305      0      3      Williams, Mr. Howard Hugh "Harry"  male  NaN  0  0
A/5 2466  8.0500  None  S
530      0      2      Hocking, Mr. Richard George  male  23.0  2
1      29104  11.5000  None  S
265      0      3      Henry, Miss. Delia  female  NaN  0
0      382649  7.7500  None  Q
```

- Sample few rows from DataFrame.

```
>>> df.sample(1)
SURVIVED  pclass  name  sex  age  sibsp  parch  ticket  fare  cabin
embarked  sampleid
PASSENGER
203      0      3  Johanson, Mr. Jakob Alfred  male  34  0  0  3101264  6.4958  None
S      1
```

Example 2: Case explaining user with PERM space

- Create context using 'user_with_perm' user that have perm space and pass "temp_database_name" as 'alice_view', so that internal objects will be created in 'alice_views'.

```
>>> create_context(host="myhostname", username="user_with_perm",
password="mypassword", temp_database_name="alice_views")
C:\Users\workspace\pyTeradata\teradataml\context\context.py:404:
UserWarning: [Teradata][teradataml](TDML_2002) Overwriting an existing
context associated with Teradata Vantage C
onnection. Most of the operations on any teradataml DataFrames created
```


before this will not work.

```
warnings.warn(Messages.get_message(MessageCodes.OVERWRITE_CONTEXT))
```

- Create DataFrame using `DataFrame.from_query()`, this will result in error "The user does not have CREATE VIEW access to database user_no_perm."

```
>>> df = DataFrame.from_query("select * from alice.titanic sample 1")
Traceback (most recent call last):
  File "C:\Users\workspace\pyTeradata\teradataml\dataframe\dataframe.py",
line 180, in __init__
    UtilFuncs._create_view(self._table_name, self._query)
  File "C:\Users\workspace\pyTeradata\teradataml\common\utils.py", line 671,
in _create_view
    UtilFuncs._execute_ddl_statement(crt_view)
  File "C:\Users\workspace\pyTeradata\teradataml\common\utils.py", line 528,
in _execute_ddl_statement
    cursor.execute(ddl_statement)
  File "C:\Users\.conda\envs\teradatamla\lib\site-
packages\teradatasql\__init__.py", line 686, in execute
    self.executemany(sOperation, None, ignoreErrors)
  File "C:\Users\.conda\envs\teradatamla\lib\site-
packages\teradatasql\__init__.py", line 933, in executemany
    raise OperationalError (sErr)
teradatasql.OperationalError: [Version 17.10.0.16] [Session 3833] [Teradata
Database] [Error 3524] The user does not have CREATE VIEW access to
database user_no_perm.
  at gosqldriver/teradatasql.formatError ErrorUtil.go:88

  at gosqldriver/teradatasql.
(*teradataConnection).formatDatabaseError ErrorUtil.go:216
  at gosqldriver/teradatasql.
(*teradataConnection).makeChainedDatabaseError ErrorUtil.go:232
  at gosqldriver/teradatasql.
(*teradataConnection).processErrorParcel TeradataConnection.go:803
  at gosqldriver/teradatasql.
(*TeradataRows).processResponseBundle TeradataRows.go:2229
  at gosqldriver/teradatasql.
(*TeradataRows).executeSQLRequest TeradataRows.go:814
  at gosqldriver/teradatasql.newTeradataRows TeradataRows.go:673
  at gosqldriver/teradatasql.
(*teradataStatement).QueryContext TeradataStatement.go:122
  at gosqldriver/teradatasql.
(*teradataConnection).QueryContext TeradataConnection.go:1304
  at database/sql.ctxDriverQuery ctxutil.go:48
```

```

at database/sql.(*DB).queryDC.func1 sql.go:1759
at database/sql.withLock sql.go:3437
at database/sql.(*DB).queryDC sql.go:1754
at database/sql.(*Conn).QueryContext sql.go:2013
at main.goCreateRows goside.go:666
at _cgoexp_7cdf4597d74c.goCreateRows _cgo_gotypes.go:340
at runtime.cgocallbackg1 cgocall.go:314
at runtime.cgocallbackg cgocall.go:233
at runtime.cgocallback asm_amd64.s:971
at runtime.goexit asm_amd64.s:1571

```

- Using 'configure' option from teradataml.
 - Set the 'temp_table_database' to 'alice_views' so that internal tables will be created in the same.

```
>>> configure.temp_table_database = "alice_views"
```

- Set the 'temp_view_database' to 'user_no_perm' so that internal views will be created in the user without perm space.

```
>>> configure.temp_view_database = "user_no_perm"
```

- Create DataFrame using DataFrame.from_query(), this will result in error "An owner referenced by user does not have any WITH GRANT OPTION access to alice.titanic."

```

>>> df = DataFrame.from_query("select * from alice.titanic sample 1")
Traceback (most recent call last):
  File "C:\Users\.conda\envs\teradataml\lib\site-packages\sqlalchemy\engine\base.py", line 1800, in _execute_context
    cursor, statement, parameters, context
  File "C:\Users\.conda\envs\teradataml\lib\site-packages\sqlalchemy\engine\default.py", line 717, in do_execute
    cursor.execute(statement, parameters)
  File "C:\Users\.conda\envs\teradataml\lib\site-packages\teradatasql\__init__.py", line 686, in execute
    self.executemany(sOperation, None, ignoreErrors)
  File "C:\Users\.conda\envs\teradataml\lib\site-packages\teradatasql\__init__.py", line 933, in executemany
    raise OperationalError(sErr)
teradatasql.OperationalError: [Version 17.10.0.16] [Session 3833] [Teradata Database] [Error 3523] An owner referenced by user does not have any WITH GRANT OPTION access to alice.titanic.
  at gosqldriver/teradatasql.formatError ErrorUtil.go:88
  at gosqldriver/teradatasql.
(*teradataConnection).formatDatabaseError ErrorUtil.go:216
  at gosqldriver/teradatasql.

```

```
(*teradataConnection).makeChainedDatabaseError ErrorUtil.go:232
  at gosqldriver/teradatasql.
(*teradataConnection).processErrorParcel TeradataConnection.go:803
  at gosqldriver/teradatasql.
(*TeradataRows).processResponseBundle TeradataRows.go:2229
  at gosqldriver/teradatasql.
(*TeradataRows).executeSQLRequest TeradataRows.go:814
  at gosqldriver/teradatasql.newTeradataRows TeradataRows.go:673
  at gosqldriver/teradatasql.
(*teradataStatement).QueryContext TeradataStatement.go:122
  at gosqldriver/teradatasql.
(*teradataConnection).QueryContext TeradataConnection.go:1304
  at database/sql.ctxDriverQuery ctxutil.go:48
  at database/sql.(*DB).queryDC.func1 sql.go:1759
  at database/sql.withLock sql.go:3437
  at database/sql.(*DB).queryDC sql.go:1754
  at database/sql.(*Conn).QueryContext sql.go:2013
  at main.goCreateRows goside.go:666
  at _cgoexp_7cdf4597d74c.goCreateRows _cgo_gotypes.go:340
  at runtime.cgocallbackg1 cgocall.go:314
  at runtime.cgocallbackg cgocall.go:233
  at runtime.cgocallback asm_amd64.s:971
  at runtime.goexit asm_amd64.s:1571
```

- Using 'configure' option from teradataml.
 - Set the 'temp_view_database' to 'user_with_perm' so that internal views will be created in the same.

```
>>> configure.temp_view_database = "user_with_perm"
```

- Create DataFrame using DataFrame.from_query(), this will work fine.

```
>>> df = DataFrame.from_query("select * from alice.titanic sample 1")
```

```
>>> df
  PASSENGER  SURVIVED  pclass      name  sex  age  sibsp  parch  ticket  fare  cabin  embarked
0         811         0        3  Alexander, Mr. William  male  26      0      0    3474   7.8875   None        S
```

Using teradataml with Native Object Store

Native Object Store (NOS) is a new capability included with Teradata Vantage that makes it easy for users to explore datasets that have been stored in JSON, comma-separated values (CSV), or Parquet format, located on external object stores like AWS S3 and Azure Blob Storage, using standard SQL. Because the data is being processed and analyzed inside the database, Native Object Store can take advantage of the scalability, performance, and workload management that is part of the Vantage platform. No special object storage-side compute infrastructure is required to use Native Object Store. Simply create a Native Object Store table definition within the Analytics Database, point it at any AWS-S3 or Azure Blob Storage bucket that you are authorized to access, and within minutes, you can explore the data located in that bucket using all the analytics functionality of Vantage. As a result, Native Object Store is ideally suited for data scientists, analysts, and other business users that want to use object stores on an ad-hoc basis to store interim results, prior versions, and other data sets as part of their analytics workflow.

You can use `teradataml` to explore this external data made available on Vantage via Native Object Store capability. This section shows how to use `teradataml` to explore data stored on external objects with the help of Native Object Store. The first step is to create a foreign table. The foreign table allows the external data to be easily referenced within the Analytics Database and makes the data available in a structured relational format which could include complex data types. Once the data is in a relational format, either persistently or virtually, it can be aggregated or joined to other relational tables.

Refer to the Native Object Store documentation for more details about NOS.

The following sections show different ways to explore foreign table data in Vantage via `teradataml`, based on the actual format of the data. You may refer to the examples in the [Teradata Vantage™ - Native Object Store Getting Started Guide](#).

Data in JSON or CSV Format

Foreign table created in JSON or comma-separated values (CSV) format usually contains two columns:

- Location
- Payload

User can create a `teradataml` `DataFrame` on a foreign table using `"DataFrame()"` or `"DataFrame.from_table()"`, the same way to create a `teradataml` `DataFrame` on a regular table. With the created `DataFrame`, user can easily access the data in these columns and process the data using `teradataml` `DataFrame` API or other Python packages.

The following sections show how to create `teradataml` `DataFrames` on NOS foreign tables.

Create `teradataml` `DataFrame` on NOS foreign table in JSON data format

Assume that the following foreign table has been created on JSON data in Amazon S3 bucket:

```
CREATE MULTISET FOREIGN TABLE riverflow ,FALLBACK ,
  EXTERNAL SECURITY DEFINER TRUSTED AUTH_OBJECT ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    Payload JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE)
USING
(
  LOCATION ('/s3/s3.amazonaws.com/td-usgs/DATA/')
  PATHPATTERN ('$data/$siteno/$year/$month/$day')
);
```

- Create a dataframe on foreign table.

```
# Create dataframe on foreign table which contains data in JSON format on S3
>>> riverflow = DataFrame.from_table("riverflow")
```

- Display the columns in the DataFrame.

```
# As seen from create table statement, table has two columns 'Location'
and 'Payload'
>>> riverflow.columns

['Location', 'Payload']
```

- Check the types of the columns.

```
# Let's check their types
>>> riverflow.dtypes

Location      str
Payload       str
```

- Print the content of the table.

```
# Let's print the DataFrame.
# Equivalent SQL:
#     SELECT * FROM riverflow
>>> riverflow.head().to_pandas()
```

| | Location | Payload |
|---|---|--------------------------|
| 0 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |
| | "datetime": "2018-06-27..." | |
| 1 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |
| | "datetime": "2018-06-27..." | |
| 2 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |
| | "datetime": "2018-06-27..." | |
| 3 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |
| | "datetime": "2018-06-27..." | |
| 4 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |
| | "datetime": "2018-06-27..." | |
| 5 | /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... | { "site_no": "09380000", |

```

"datetime":"2018-06-27...
6 /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... { "site_no":"09380000",
"datetime":"2018-06-27...
7 /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... { "site_no":"09380000",
"datetime":"2018-06-27...
8 /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... { "site_no":"09380000",
"datetime":"2018-06-27...
9 /S3/s3.amazonaws.com/td-usgs/DATA/09380000/201... { "site_no":"09380000", "datetime":"2018-06-27...

```

Create teradataml DataFrame on NOS foreign table in CSV data format

Assume that the following foreign table has been created on CSV data in Amazon S3 bucket:

```

CREATE FOREIGN TABLE riverflowcsv
, EXTERNAL SECURITY DEFINER TRUSTED AUTH_OBJECT
(
  Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
  PAYLOAD DATASET INLINE LENGTH 64000 STORAGE FORMAT CSV)
USING
(
  LOCATION('/s3/s3.amazonaws.com/td-usgs/CSVDATA/')
  PATHPATTERN ('$data/$siteno/$year/$month/$day')
);

```

- Create a dataframe on foreign table.

```

# Create dataframe on foreign table which contains data in CSV format on S3
>>> riverflow = DataFrame.from_table("riverflowcsv")

```

- Display the columns in the DataFrame.

```

# As seen from create table statement, table has two columns 'Location'
and 'Payload'
>>> riverflow.columns

```

```
['Location', 'Payload']
```

- Check the types of the columns.

```

# Let's check their types
>>> riverflow.dtypes

```

```

Location    str
Payload     str

```

- Print the content of the table.

```

# Let's print the DataFrame.
# Equivalent SQL:
#     SELECT * FROM riverflow
>>> riverflow.head().to_pandas()

```

| | Location | PAYLOAD |
|---|---|---------|
| 0 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 1 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 2 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 3 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 4 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 5 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 6 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 7 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 8 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |
| 9 | /S3/s3.amazonaws.com/td-usgs/CSVDATA/09380000/... | |
| | Temp,Flow,site_no,datetime,Conductance,Precipi... | |

How to access actual data and path variables

Though teradataml provides direct access to the data in foreign tables, actual data that resides in 'Payload' column and path variables for the foreign table can be accessed in one of the following ways:

- [Accessing Columns and Path Variables using DataFrame.from_query\(\)](#)
- [Accessing Columns and Path Variables by Creating a View on Foreign Table](#)
- [Accessing Columns and Path Variables by Creating a Table from Foreign Table](#)

Accessing Columns and Path Variables using DataFrame.from_query()

User can access actual columns and path variables using `DataFrame.from_query()` that pass a SELECT query with each column from JSON or CSV data projected from foreign table. Each column must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access actual columns and keys in the JSON or CSV data. It is up to the user on what must be selected in SELECT query passed to "`DataFrame.from_query()`": columns, attributes, keys from JSON or CSV data and path variables.

Example for JSON data

SELECT query with each column from JSON selected.

```
# Select query with each column from JSON selected. Each column is type casted
# to a valid type and
# then aliased to the required column name. Notice, we are selecting each
# attribute including path variables.
```

```
query = """SELECT CAST($path.$siteno AS CHAR(10)) TheSite,
CAST($path.$year AS CHAR(4)) TheYear,
CAST($path.$month AS CHAR(2)) TheMonth,
CAST($path.$day AS CHAR(2)) TheDay,
CAST(payload.site_no AS CHAR(8)) Site_no,
CAST(payload.Flow AS FLOAT) Flow,
```

```

CAST(payload.GageHeight AS FLOAT) GageHeight1,
CAST(payload.Precipitation AS FLOAT) Precipitation,
CAST(payload.Temp AS FLOAT) Temperature,
CAST(payload.Velocity AS FLOAT) Velocity,
CAST(payload.BatteryVoltage AS FLOAT) BatteryVoltage,
CAST(payload.GageHeight2 AS FLOAT) GageHeight2
FROM riverflow""

```

Create DataFrame from the query and display the head of the DataFrame.

```

>>> wrk1df = DataFrame.from_query(query)
>>> wrk1df.head().to_pandas()

```

| | TheSite | TheYear | TheMonth | TheDay | Site_no | Flow | GageHeight1 | Precipitation | Temperature | Velocity | BatteryVoltage |
|-------------|----------|---------|----------|--------|----------|----------|-------------|---------------|-------------|----------|----------------|
| GageHeight2 | 09380000 | | 2018 | 07 | 14 | 09380000 | 11400.0 | 8.99 | 0.0 | 11.4 | None |
| 0 | None | None | | | | | | | | | |
| 1 | 09380000 | 2018 | 07 | 27 | 09380000 | 11100.0 | 8.91 | 0.0 | 11.0 | None | None |
| 2 | 09380000 | 2018 | 07 | 25 | 09380000 | 18600.0 | 10.41 | 0.0 | 11.7 | None | None |
| 3 | 09380000 | 2018 | 06 | 29 | 09380000 | 16700.0 | 10.07 | 0.0 | 11.3 | None | None |
| 4 | 09380000 | 2018 | 07 | 03 | 09380000 | 18900.0 | 10.46 | 0.0 | 11.0 | None | None |
| 5 | 09380000 | 2018 | 07 | 13 | 09380000 | 11000.0 | 8.90 | 0.0 | 10.8 | None | None |
| 6 | 09380000 | 2018 | 07 | 15 | 09380000 | 11400.0 | 8.99 | 0.0 | 12.0 | None | None |
| 7 | 09380000 | 2018 | 07 | 18 | 09380000 | 11300.0 | 8.96 | 0.0 | 10.7 | None | None |
| 8 | 09380000 | 2018 | 06 | 30 | 09380000 | 0.0 | 0.00 | 0.0 | 11.6 | None | None |
| 9 | 09380000 | 2018 | 07 | 16 | 09380000 | 16700.0 | 10.07 | 0.0 | 11.0 | None | None |

Example for CSV data

SELECT query with each column from CSV selected.

```

# Select query with each column from CSV selected. Each column is type casted
to a valid type and
# then aliased to the required column name. Notice, we are selecting each
attribute including path variables.

```

```

query = """SELECT CAST($path.$sitenno AS CHAR(10)) TheSite,
CAST($path.$year AS CHAR(4)) TheYear,
CAST($path.$month AS CHAR(2)) TheMonth,
CAST($path.$day AS CHAR(2)) TheDay,
CAST(payload..site_no AS CHAR(8)) Site_no,
CAST(payload..Flow AS FLOAT) Flow,
CAST(payload..GageHeight AS FLOAT) GageHeight1,
CAST(payload..Precipitation AS FLOAT) Precipitation,
CAST(payload..Temp AS FLOAT) Temperature,
CAST(payload..Velocity AS FLOAT) Velocity,

```



```

        CAST(payload..BatteryVoltage AS FLOAT) BatteryVoltage,
        CAST(payload..GageHeight2 AS FLOAT) GageHeight2
FROM riverflowcsv""

```

Create DataFrame from the query and display the head of the DataFrame.

```

>>> wrk1df = DataFrame.from_query(query)
>>> wrk1df.head().to_pandas()

```

| | TheSite | TheYear | TheMonth | TheDay | Site_no | Flow | GageHeight1 | Precipitation | Temperature | Velocity | |
|----------------|----------|---------|----------|--------|---------|----------|-------------|---------------|-------------|----------|------|
| BatteryVoltage | 09380000 | | 2018 | 06 | 30 | 09380000 | 11700.0 | 9.04 | 0.0 | 10.0 | None |
| 0 | None | None | | | | | | | | | |
| 1 | 09380000 | | 2018 | 07 | 04 | 09380000 | 11400.0 | 8.99 | 0.0 | 11.9 | None |
| 2 | 09380000 | | 2018 | 07 | 11 | 09380000 | 18700.0 | 10.42 | 0.0 | 11.2 | None |
| 3 | 09380000 | | 2018 | 06 | 29 | 09380000 | 11600.0 | 9.02 | 0.0 | 10.2 | None |
| 4 | 09380000 | | 2018 | 07 | 06 | 09380000 | 16600.0 | 10.05 | 0.0 | 10.8 | None |
| 5 | 09380000 | | 2018 | 07 | 22 | 09380000 | 11500.0 | 9.00 | 0.0 | 10.9 | None |
| 6 | 09380000 | | 2018 | 07 | 11 | 09380000 | 15200.0 | 9.79 | 0.0 | 10.9 | None |
| 7 | 09380000 | | 2018 | 06 | 30 | 09380000 | 10000.0 | 8.65 | 0.0 | 11.7 | None |
| 8 | 09380000 | | 2018 | 07 | 15 | 09380000 | 11400.0 | 8.98 | 0.0 | 11.8 | None |
| 9 | 09380000 | | 2018 | 06 | 28 | 09380000 | 17000.0 | 10.12 | 0.0 | 11.6 | None |
| None | None | | | | | | | | | | |

Accessing Columns and Path Variables by Creating a View on Foreign Table

User can access actual columns and path variables by creating a view using a SELECT query with each column from JSON or CSV data projected from foreign table. Each column must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access actual columns and keys in the JSON or CSV data. It is up to the user on what must be selected in SELECT query passed to "DataFrame.from_query()": columns, attributes, keys from JSON or CSV data and path variables.

Example for JSON data

Create a view.

```

# While creating a view select each column is type casted to a valid type and
# then aliased to the required column name. Notice, we are selecting each
attribute including path variables.

# Following is the VIEW created at the backend:
"""
REPLACE VIEW riverflowview AS (
SELECT CAST($path.$sitenno AS CHAR(10)) TheSite,
        CAST($path.$year AS CHAR(4)) TheYear,
        CAST($path.$month AS CHAR(2)) TheMonth,

```

```

CAST($path.$day AS CHAR(2)) TheDay,
CAST(payload.site_no AS CHAR(8)) Site_no,
CAST(payload.Flow AS FLOAT) Flow,
CAST(payload.GageHeight AS FLOAT) GageHeight1,
CAST(payload.Precipitation AS FLOAT) Precipitation,
CAST(payload.Temp AS FLOAT) Temperature,
CAST(payload.Velocity AS FLOAT) Velocity,
CAST(payload.BatteryVoltage AS FLOAT) BatteryVoltage,
CAST(payload.GageHeight2 AS FLOAT) GageHeight2
FROM riverflow);
"""

```

Create a DataFrame on the view and display the head of the DataFrame.

```

# Create a DataFrame on a view.
>>> wrk2dfview = DataFrame("riverflowview")
>>> wrk2dfview.head().to_pandas()

```

| | Velocity | TheSite | TheYear | TheMonth | TheDay | Site_no | Flow | GageHeight1 | Precipitation | Temperature | | |
|---|----------|----------|---------|----------|--------|----------|---------|-------------|---------------|-------------|------|------|
| 0 | 09380000 | 09380000 | 2018 | 07 | 17 | 09380000 | 18300.0 | 10.36 | 0.0 | 12.0 | None | None |
| 1 | 09380000 | 09380000 | 2018 | 07 | 11 | 09380000 | 18000.0 | 10.31 | 0.0 | 11.2 | None | None |
| 2 | 09380000 | 09380000 | 2018 | 07 | 11 | 09380000 | 18500.0 | 10.40 | 0.0 | 11.5 | None | None |
| 3 | 09380000 | 09380000 | 2018 | 07 | 06 | 09380000 | 19000.0 | 10.47 | 0.0 | 11.5 | None | None |
| 4 | 09380000 | 09380000 | 2018 | 07 | 14 | 09380000 | 11500.0 | 9.01 | 0.0 | 10.3 | None | None |
| 5 | 09380000 | 09380000 | 2018 | 07 | 04 | 09380000 | 11400.0 | 8.98 | 0.0 | 11.9 | None | None |
| 6 | 09380000 | 09380000 | 2018 | 07 | 01 | 09380000 | 11100.0 | 8.92 | 0.0 | 10.7 | None | None |
| 7 | 09380000 | 09380000 | 2018 | 06 | 29 | 09380000 | 16700.0 | 10.07 | 0.0 | 11.3 | None | None |
| 8 | 09380000 | 09380000 | 2018 | 07 | 25 | 09380000 | 18100.0 | 10.33 | 0.0 | 11.9 | None | None |
| 9 | 09380000 | 09380000 | 2018 | 07 | 02 | 09380000 | 18800.0 | 10.44 | 0.0 | 11.4 | None | None |

Example for CSV data

Create a view.

```

# While creating a view select each column is type casted to a valid type and
# then aliased to the required column name. Notice, we are selecting each
attribute including path variables.

```

```

# Following is the VIEW created at the backend:
"""

```

```

REPLACE VIEW riverflowcsvview AS (
SELECT CAST($path.$siteno AS CHAR(10)) TheSite,
      CAST($path.$year AS CHAR(4)) TheYear,
      CAST($path.$month AS CHAR(2)) TheMonth,
      CAST($path.$day AS CHAR(2)) TheDay,
      CAST(payload..site_no AS CHAR(8)) Site_no,
      CAST(payload..Flow AS FLOAT) Flow,
      CAST(payload..GageHeight AS FLOAT) GageHeight1,
      CAST(payload..Precipitation AS FLOAT) Precipitation,

```

```

        CAST(payload..Temp AS FLOAT) Temperature,
        CAST(payload..Velocity AS FLOAT) Velocity,
        CAST(payload..BatteryVoltage AS FLOAT) BatteryVoltage,
        CAST(payload..GageHeight2 AS FLOAT) GageHeight2
FROM riverflowcsv);
"""

```

Create DataFrame on the view and display the head of the DataFrame.

```

# Create a DataFrame on a view.
>>> wrk2dfview = DataFrame("riverflowcsvview")
>>> wrk2dfview.head().to_pandas()

```

| | TheSite | TheYear | TheMonth | | TheDay | Site_no | Flow | GageHeight1 | Precipitation | Temperature | Velocity | |
|---|----------------|---------|-------------|----|----------|---------|-------|-------------|---------------|-------------|----------|------|
| | BatteryVoltage | | GageHeight2 | | | | | | | | | |
| 0 | 09380000 | 2018 | 07 | 18 | 09380000 | 18700.0 | 10.43 | 0.0 | 11.4 | None | None | None |
| 1 | 09380000 | 2018 | 06 | 29 | 09380000 | 16800.0 | 10.09 | 0.0 | 11.0 | None | None | None |
| 2 | 09380000 | 2018 | 07 | 06 | 09380000 | 11100.0 | 8.90 | 0.0 | 10.5 | None | None | None |
| 3 | 09380000 | 2018 | 07 | 10 | 09380000 | 15900.0 | 9.92 | 0.0 | 11.0 | None | None | None |
| 4 | 09380000 | 2018 | 07 | 12 | 09380000 | 18400.0 | 10.37 | 0.0 | 11.2 | None | None | None |
| 5 | 09380000 | 2018 | 07 | 12 | 09380000 | 11500.0 | 9.00 | 0.0 | 11.0 | None | None | None |
| 6 | 09380000 | 2018 | 06 | 28 | 09380000 | 11500.0 | 9.00 | 0.0 | 10.5 | None | None | None |
| 7 | 09380000 | 2018 | 07 | 12 | 09380000 | 11000.0 | 8.89 | 0.0 | 11.0 | None | None | None |
| 8 | 09380000 | 2018 | 07 | 10 | 09380000 | 15400.0 | 9.82 | 0.0 | 10.5 | None | None | None |
| 9 | 09380000 | 2018 | 07 | 26 | 09380000 | 18800.0 | 10.45 | 0.0 | 11.6 | None | None | None |

Accessing Columns and Path Variables by Creating a Table from Foreign Table

User can access actual columns and path variables by creating a regular table on a SELECT query (Create Table as SELECT, CTAS) with each column from JSON or CSV data selected from foreign table. Each column must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access actual columns and keys in the JSON or CSV data. It is up to the user on what must be selected in SELECT query passed to "DataFrame.from_query()": columns, attributes, keys from JSON or CSV data and path variables.

Example for JSON data

Create a regular table.

```

# Here is how we created a regular table from foreign table.

"""
CREATE TABLE riverflowprecip AS (
SELECT CAST($path.$year AS CHAR(4)) TheYear,
       CAST($path.$month AS CHAR(2)) TheMonth,
       CAST($path.$day AS CHAR(2)) TheDay,
       CAST(payload.site_no AS CHAR(8)) SiteNo,
       CAST(payload.Flow AS FLOAT) Flow,
       CAST(payload.GageHeight AS FLOAT) GageHeight

```

```

        FROM riverflow
        WHERE payload.Precipitation IS NOT NULL )
WITH DATA
PRIMARY INDEX(SiteNo)
"""

```

Create a DataFrame on the table and display the head of the DataFrame.

```

# Create a DataFrame on a table.
>>> wrk2dfview = DataFrame("riverflowprecip")
>>> wrk2dfview.head().to_pandas()

```

| TheMonth | TheDay | SiteNo | Flow | GageHeight |
|----------|--------|----------|------|------------|
| 2018 07 | 25 | 09400815 | | 0.00 0.00 |
| 2018 07 | 20 | 09400815 | | 0.00 0.42 |
| 2018 07 | 20 | 09400815 | | 0.00 0.00 |
| 2018 07 | 20 | 09400815 | | 0.00 0.41 |
| 2018 07 | 20 | 09400815 | | 0.27 0.54 |
| 2018 07 | 20 | 09400815 | | 0.19 0.51 |
| 2018 07 | 20 | 09400815 | | 0.05 0.45 |
| 2018 07 | 25 | 09400815 | | 0.00 0.36 |
| 2018 07 | 25 | 09400815 | | 0.00 0.37 |
| 2018 07 | 01 | 09400815 | | 0.00 -0.01 |

Example for CSV data

Create a regular table.

```

# Here is how we created a regular table from foreign table.

"""
CREATE TABLE riverflowcsvprecip AS (
SELECT CAST($path.$year AS CHAR(4)) TheYear,
       CAST($path.$month AS CHAR(2)) TheMonth,
       CAST($path.$day AS CHAR(2)) TheDay,
       CAST(payload..site_no AS CHAR(8)) SiteNo,
       CAST(payload..Flow AS FLOAT) Flow,
       CAST(payload..GageHeight AS FLOAT) GageHeight
FROM riverflowcsv
WHERE payload..Precipitation IS NOT NULL )
WITH DATA

```

```
PRIMARY INDEX(SiteNo)
"""
```

Create a DataFrame on the table and display the head of the DataFrame.

```
# Create a DataFrame on a table.
>>> wrk2dfview = DataFrame("riverflowcsvprecip")
>>> wrk2dfview.head().to_pandas()
```

| SiteNo | TheYear | TheMonth | TheDay | Flow | GageHeight |
|----------|---------|----------|--------|---------|------------|
| 09380000 | 2018 | 07 | 25 | 13900.0 | 9.52 |
| 09380000 | 2018 | 07 | 25 | 12800.0 | 9.30 |
| 09380000 | 2018 | 07 | 25 | 12500.0 | 9.24 |
| 09380000 | 2018 | 07 | 25 | 12300.0 | 9.18 |
| 09380000 | 2018 | 07 | 25 | 11600.0 | 9.04 |
| 09380000 | 2018 | 07 | 25 | 11500.0 | 9.00 |
| 09380000 | 2018 | 07 | 25 | 11800.0 | 9.08 |
| 09380000 | 2018 | 07 | 25 | 13500.0 | 9.44 |
| 09380000 | 2018 | 07 | 25 | 14300.0 | 9.61 |
| 09380000 | 2018 | 07 | 25 | 15500.0 | 9.85 |

Data in Parquet Format

Foreign table created in Parquet format usually contains the following columns:

- Location
- Several other user specified columns in Parquet format specified while creating foreign table

User can create a teradataml DataFrame on a foreign table using "DataFrame()" or "DataFrame.from_table()", the same way to create a teradataml DataFrame on a regular table. With the created DataFrame, user can easily access the data in these columns and process the data using teradataml DataFrame API or other Python packages.

How to access actual data and path variables

Unlike foreign tables on JSON and CSV format data, teradataml DataFrame on Parquet data provides direct access to the actual data in Parquet files, as described in [Accessing Foreign Table Created On Parquet Data](#).

Path variables from the foreign table can be accessed in one of the following ways:

- [Accessing Path Variables using DataFrame.from_query\(\)](#)
- [Accessing Path Variables by Creating a View on Foreign Table in Vantage](#)
- [Accessing Path Variables by Creating a Table from Foreign Table](#)

Accessing Foreign Table Created On Parquet Data

The following examples show how to create a teradataml DataFrame on this table and process this DataFrame with other teradataml API's.

Assume that the following foreign table has been created on Parquet data in Amazon S3 bucket:

```
CREATE MULTiset FOREIGN TABLE t11,FALLBACK,
EXTERNAL SECURITY INVOKER TRUSTED AUTH_OBJECT(
Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
a INTEGER,
b INTEGER,
c INTEGER,
d INTEGER,
e VARCHAR(10) CHARACTER SET UNICODE CASESPECIFIC,
f DATE FORMAT 'YYYY-MM-DD',
g Decimal(8, 3),
h FLOAT,
i BYTEINT,
j SMALLINT)
USING(
    LOCATION ('/s3/ceph2-s3.teradata.com/mk250104/csoj_files/t11.parquet')
    PATHPATTERN ('$Var1/$Var2/$var3/$Var4')
    STOREDAS ('PARQUET')
    MANIFEST ('FALSE')
)
NO PRIMARY INDEX
PARTITION BY COLUMN;
```

Create a dataframe on the foreign table

```
# Create dataframe on foreign table which contains data in PARQUET data
>>> t11 = DataFrame.from_table("t11")
```

Check Properties of the DataFrame

- Display the columns in the DataFrame.

```
# Let's take a peek at the columns in a DataFrame.
>>> t11.columns
```

```
['Location', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- Check the types of the columns.

```
# Let's check their types
>>> t1l.dtypes
```

```
Location          str
a                  int
b                  int
c                  int
d                  int
e                  str
f      datetime.date
g      decimal.Decimal
h                  float
i                  int
j                  int
```

- Keys

```
# Keys
>>> t1l.keys
```

```
<bound method DataFrame.keys of
0      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 7  70 107 1  CScs      1963-02-03 100.333 19.0
4 10
1      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 3  30 103 1  CScs      1961-06-05 300.333 19.0
4 10
2      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 10 100 110 1  CScs      1963-02-03 100.333 19.0
4 10
3      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 4   40 104 1  CScs      1961-06-05 100.333 19.0
4 10
4      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 8   80 108 1  CScs      1963-02-03 200.333 19.0
4 10
5      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 2   20 102 1  CScs      1961-06-05 200.333 19.0
4 10
6      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 1   10 101 1  CScs      1961-06-05 100.333 19.0
4 10
7      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 6   60 106 1  CScs      1963-02-03 300.333 19.0
4 10
8      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 9   90 109 1  CScs      1963-02-03 300.333 19.0
4 10
9      /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... 5   50 105 1  CScs      1961-06-05 200.333 19.0
4 10>
```

- Shape

```
# shape
>>> t1l.shape
```

```
(10, 11)
```

Explore Data

- Select some columns from the DataFrame.

```
>>> t1l.select(["a", "b", "e", "f", "i", "j"]).to_pandas()
```

| | a | b | e | f | i | j |
|---|----|-----|------|------------|---|----|
| 0 | 7 | 70 | CScs | 1963-02-03 | 4 | 10 |
| 1 | 3 | 30 | CScs | 1961-06-05 | 4 | 10 |
| 2 | 10 | 100 | CScs | 1963-02-03 | 4 | 10 |
| 3 | 4 | 40 | CScs | 1961-06-05 | 4 | 10 |
| 4 | 8 | 80 | CScs | 1963-02-03 | 4 | 10 |
| 5 | 2 | 20 | CScs | 1961-06-05 | 4 | 10 |
| 6 | 1 | 10 | CScs | 1961-06-05 | 4 | 10 |
| 7 | 6 | 60 | CScs | 1963-02-03 | 4 | 10 |
| 8 | 9 | 90 | CScs | 1963-02-03 | 4 | 10 |
| 9 | 5 | 50 | CScs | 1961-06-05 | 4 | 10 |

- Filter data.

```
>>> t1l[t1l.b == 70].to_pandas()
```

| | Location | a | b | c | d | e | f | g | h | i | j |
|------------|---|---------|------|---|----|---|---|----|-----|---|------|
| 0 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | 7 | 70 | 107 | 1 | CScs |
| 1963-02-03 | | 100.333 | 19.0 | 4 | 10 | | | | | | |

- Sample data.

```
>>> t1l.sample(5).to_pandas()
```

| | Location | a | b | c | d | e | f | g | h | i | j | sampleid |
|------------|---|---------|------|---|----|---|---|---|----|-----|---|----------|
| 0 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | | 6 | 60 | 106 | 1 | CScs |
| 1963-02-03 | | 300.333 | 19.0 | 4 | 10 | | | | | | | |
| 1 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | | 2 | 20 | 102 | 1 | CScs |
| 1961-06-05 | | 200.333 | 19.0 | 4 | 10 | | | | | | | |
| 2 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | | 4 | 40 | 104 | 1 | CScs |
| 1961-06-05 | | 100.333 | 19.0 | 4 | 10 | | | | | | | |
| 3 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | | 9 | 90 | 109 | 1 | CScs |
| 1963-02-03 | | 300.333 | 19.0 | 4 | 10 | | | | | | | |
| 4 | /S3/ceph2-s3.teradata.com/mk250104/csoj_files/... | | | | | | | 5 | 50 | 105 | 1 | CScs |
| 1961-06-05 | | 200.333 | 19.0 | 4 | 10 | | | | | | | |

Accessing Path Variables using DataFrame.from_query()

User can access actual column and path variables using DataFrame.from_query() that pass a SELECT query with each required column and path variable selected from foreign table. Each path variable must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access path variables along with Parquet data. It is up to the user on what must be selected in SELECT query passed to "DataFrame.from_query()": columns, attributes, keys from foreign table and path variables.

Example

SELECT query with each column and path variable selected.

```
# Select query with each column from PARQUET file selected. Each column is type
# casted to a valid type and
# then aliased to the required column name. Notice, we are selecting each
# attribute including path variables.
query = """SELECT CAST($path.$var1 AS CHAR(10)) Var1,
```



```

        CAST($path.$var2 AS CHAR(4)) Var2,
        CAST($path.$var3 AS CHAR(2)) var3,
        CAST($path.$var4 AS CHAR(2)) Var4,
        a, b, c, d, e, f, g, h, i, j
FROM t11""

```

Create DataFrame from the query and display the head of the DataFrame.

```

>>> wrk1df = DataFrame.from_query(query)
>>> wrk1df.head().to_pandas()

```

| | Var1 | Var2 | var3 | Var4 | a | b | c | d | e | f | g | h | i | j |
|---|------------|------|------|------|------|----|-----|-----|---|------|------------|---|---------|------|
| 0 | csoj_files | t11. | None | None | None | 10 | 100 | 110 | 1 | CScs | 1963-02-03 | | 100.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 1 | csoj_files | t11. | None | None | None | 8 | 80 | 108 | 1 | CScs | 1963-02-03 | | 200.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 2 | csoj_files | t11. | None | None | None | 2 | 20 | 102 | 1 | CScs | 1961-06-05 | | 200.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 3 | csoj_files | t11. | None | None | None | 1 | 10 | 101 | 1 | CScs | 1961-06-05 | | 100.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 4 | csoj_files | t11. | None | None | None | 9 | 90 | 109 | 1 | CScs | 1963-02-03 | | 300.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 5 | csoj_files | t11. | None | None | None | 5 | 50 | 105 | 1 | CScs | 1961-06-05 | | 200.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 6 | csoj_files | t11. | None | None | None | 6 | 60 | 106 | 1 | CScs | 1963-02-03 | | 300.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 7 | csoj_files | t11. | None | None | None | 4 | 40 | 104 | 1 | CScs | 1961-06-05 | | 100.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 8 | csoj_files | t11. | None | None | None | 3 | 30 | 103 | 1 | CScs | 1961-06-05 | | 300.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |
| 9 | csoj_files | t11. | None | None | None | 7 | 70 | 107 | 1 | CScs | 1963-02-03 | | 100.333 | 19.0 |
| 4 | 10 | | | | | | | | | | | | | |

Accessing Path Variables by Creating a View on Foreign Table in Vantage

User can access actual columns and path variables by creating a view on a SELECT query with each required column and path variable selected from a foreign table. Each path variable must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access path variables along with Parquet data. It is up to the user on what must be selected in SELECT query passed to "DataFrame.from_query()": columns, attributes, keys from foreign table and path variables.

Example

Create a view.

```

# While creating a view select each column is type casted to a valid type and
# then aliased to the required column name. Notice, we are selecting each
attribute including path variables.

# Following is the VIEW created at the backend:
"""
REPLACE VIEW t11view AS (
SELECT CAST($path.$var1 AS CHAR(10)) Var1,
        CAST($path.$var2 AS CHAR(4)) Var2,

```

```

        CAST($path.$var3 AS CHAR(2)) var3,
        CAST($path.$var4 AS CHAR(2)) Var4,
        a, b, c, d, e, f, g, h, i, j
FROM t1l);
"""

```

Create a DataFrame on the view and display the head of the DataFrame.

```

# Create a DataFrame on a view.
>>> wrk2dfview = DataFrame("t1lview")
>>> wrk2dfview.head().to_pandas()

```

| Var1 | Var2 | var3 | Var4 | a | b | c | d | e | f | g | h | i | j |
|------|------------|------|------|------|------|----|-----|-----|---|------|------------|---------|------|
| 0 | csoj_files | 10 | t1l. | None | None | 10 | 100 | 110 | 1 | CScs | 1963-02-03 | 100.333 | 19.0 |
| 1 | csoj_files | 10 | t1l. | None | None | 8 | 80 | 108 | 1 | CScs | 1963-02-03 | 200.333 | 19.0 |
| 2 | csoj_files | 10 | t1l. | None | None | 2 | 20 | 102 | 1 | CScs | 1961-06-05 | 200.333 | 19.0 |
| 3 | csoj_files | 10 | t1l. | None | None | 1 | 10 | 101 | 1 | CScs | 1961-06-05 | 100.333 | 19.0 |
| 4 | csoj_files | 10 | t1l. | None | None | 9 | 90 | 109 | 1 | CScs | 1963-02-03 | 300.333 | 19.0 |
| 5 | csoj_files | 10 | t1l. | None | None | 5 | 50 | 105 | 1 | CScs | 1961-06-05 | 200.333 | 19.0 |
| 6 | csoj_files | 10 | t1l. | None | None | 6 | 60 | 106 | 1 | CScs | 1963-02-03 | 300.333 | 19.0 |
| 7 | csoj_files | 10 | t1l. | None | None | 4 | 40 | 104 | 1 | CScs | 1961-06-05 | 100.333 | 19.0 |
| 8 | csoj_files | 10 | t1l. | None | None | 3 | 30 | 103 | 1 | CScs | 1961-06-05 | 300.333 | 19.0 |
| 9 | csoj_files | 10 | t1l. | None | None | 7 | 70 | 107 | 1 | CScs | 1963-02-03 | 100.333 | 19.0 |

Accessing Path Variables by Creating a Table from Foreign Table

User can access actual columns and path variables by creating a regular table on a SELECT query (Create Table as SELECT, CTAS) with each required column and path variables selected from the foreign table. Each path variable must be typecast to a valid type and then aliased to the appropriate column name. This allows user to access path variables along with Parquet data. It is up to the user on what must be selected in SELECT query passed to "DataFrame.from_query()": columns, attributes, keys from foreign table and path variables.

Example

Create a regular table.

```

# Let's see an example. Here is how we created a regular table from
foreign table.

"""
CREATE TABLE t1lctas AS (
SELECT CAST($path.$var1 AS CHAR(10)) Var1,
        CAST($path.$var2 AS CHAR(4)) Var2,
        CAST($path.$var3 AS CHAR(2)) var3,

```

```

        CAST($path.$var4 AS CHAR(2)) Var4,
        a, b, c, d, e, f, g, h, i, j
FROM t11)
WITH DATA
PRIMARY INDEX(a)
"""

```

Create a DataFrame on the table and display the head of the DataFrame.

```

# Create a DataFrame on a table.
>>> wrk2dfview = DataFrame("t11ctas")
>>> wrk2dfview.head().to_pandas()

```

| Var1 | Var2 | var3 | Var4 | b | c | d | e | f | g | h | i | j | |
|------|------------|------|------|------|------|-----|-----|---|------|------------|---------|------|---|
| a | | | | | | | | | | | | | |
| 3 | csoj_files | t11. | | None | None | 30 | 103 | 1 | CScs | 1961-06-05 | 300.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 5 | csoj_files | t11. | | None | None | 50 | 105 | 1 | CScs | 1961-06-05 | 200.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 6 | csoj_files | t11. | | None | None | 60 | 106 | 1 | CScs | 1963-02-03 | 300.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 7 | csoj_files | t11. | | None | None | 70 | 107 | 1 | CScs | 1963-02-03 | 100.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 9 | csoj_files | t11. | | None | None | 90 | 109 | 1 | CScs | 1963-02-03 | 300.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 10 | csoj_files | t11. | | None | None | 100 | 110 | 1 | CScs | 1963-02-03 | 100.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 8 | csoj_files | t11. | | None | None | 80 | 108 | 1 | CScs | 1963-02-03 | 200.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 4 | csoj_files | t11. | | None | None | 40 | 104 | 1 | CScs | 1961-06-05 | 100.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 2 | csoj_files | t11. | | None | None | 20 | 102 | 1 | CScs | 1961-06-05 | 200.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |
| 1 | csoj_files | t11. | | None | None | 10 | 101 | 1 | CScs | 1961-06-05 | 100.333 | 19.0 | 4 |
| 10 | | | | | | | | | | | | | |

teradataml Extension with SQLAlchemy

Teradata Vantage offers various SQL functions that allow a user to process data. This section describes how these common SQL functions can be used in teradataml with the help of SQLAlchemy.

You do not need to have thorough knowledge of SQLAlchemy to use the SQLAlchemy extension to teradataml. Refer to the examples provided in the demo notebooks which are part of the teradataml package. These demo notebooks can be found at <pkg_install_location>/teradataml/data/notebooks/sqlalchemy/.

Note:

teradataml 17.20.00.xx versions only support SQLAlchemy 1.4,
SQLAlchemy 2.0 is not yet supported.

Accessing Vantage SQL Functions

teradataml supports SQL functions with the SQLAlchemy extension. For lists of these functions, see [Supported SQL Functions](#).

To access these functions in teradataml, you only need to know the following SQLAlchemy statement, the rest of the functionality is managed by teradataml and Vantage.

```
from sqlalchemy import func
```

You can use this imported func module to call Vantage SQL functions and pass those to DataFrame.assign() function to process the data.

For more details about these functions, refer to *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example

In this example, you will calculate the standard average of column 'gpa' and 'admitted' of the "admissions_trains" dataset using the avg/average/ave function in Vantage.

The avg/average/ave function has the following syntax, and returns the arithmetic average of all values in *value_expression*.

```
Avg(value_expression)
```

- Create connection to Vantage and load example dataset.

```
# Get the connection to the Vantage using create_context()
>>> from teradataml import *
```

```
>>>td_context = create_context(host=getpass.getpass("Hostname:
"), username=getpass.getpass("Username: "),
password=getpass.getpass("Password: "))
```

```
# Load the example dataset.
load_example_data("GLM", ["admissions_train"])
```

- Prepare example dataset.

```
# Create the DataFrame on 'admissions_train' table
>>> admissions_train = DataFrame("admissions_train")
>>> admissions_train
```

| | masters | gpa | stats | programming | admitted |
|----|---------|------|----------|-------------|----------|
| id | | | | | |
| 13 | no | 4.00 | Advanced | Novice | 1 |
| 9 | no | 3.82 | Advanced | Advanced | 1 |
| 33 | no | 3.55 | Novice | Novice | 1 |
| 6 | yes | 3.50 | Beginner | Advanced | 1 |
| 7 | yes | 2.33 | Novice | Novice | 1 |
| 22 | yes | 3.46 | Novice | Beginner | 0 |
| 37 | no | 3.52 | Novice | Novice | 1 |
| 35 | no | 3.68 | Novice | Beginner | 1 |
| 27 | yes | 3.96 | Advanced | Advanced | 0 |
| 4 | yes | 3.50 | Beginner | Novice | 1 |

- Import func module from SQLAlchemy package.

```
# Import func from SQLAlchemy to use the same for executing
aggregate functions
>>> from sqlalchemy import func
```

- Use the func module to generate a function object "agg_func_".

```
>>> agg_func_ = func.avg(admissions_train.gpa.expression)
```

Where:

- *func*: sqlalchemy module that is imported.
- *avg*: the Vantage function name.

Note:

The function name is case insensitive.

You can pass the function name in lowercase (avg), uppercase (AVG) or in mixed cases (Avg).

- *admissions_train.gpa.expression*: an expression that is passed to the function, specifying the column to use for calculating average.

Where:

admissions_train: the teradataml DataFrame;

gpa: a column name of the teradataml DataFrame;

Together, *admissions_train.gpa* forms a ColumnExpression (teradataml DataFrame Column).

expression: a property of teradataml DataFrame column, which is passed as the required expression to this function object.

- View the type of the function object "agg_func_".

```
>>> type(agg_func_)
```

```
sqlalchemy.sql.functions.Function
```

- Pass the function object "agg_func_", with other function objects for calculating average value of the 'admitted' column, to DataFrame.assign() function.

```
>>> df = admissions_train.assign(True, avg_gpa_=agg_func_,
                                average_admitted_=func.average(admissions_train.admitted.expression),
                                ave_admitted_=func.ave(admissions_train.admitted.expression))
```

```
>>> print_variables(df, ["avg_gpa_", "average_admitted_", "ave_admitted_"])
```

Equivalent SQL: select ave(admitted) AS ave_admitted_, average(admitted) AS average_admitted_, avg(gpa) AS avg_gpa_ from "admissions_train"

```
***** DataFrame *****
  ave_admitted_  average_admitted_  avg_gpa_
0           0.65              0.65    3.54175
```

```
***** DataFrame.dtypes *****
ave_admitted_      float
average_admitted_  float
avg_gpa_           float
```

```
'avg_gpa_' Column Type: FLOAT
```

```
'average_admitted_' Column Type: FLOAT
'ave_admitted_' Column Type: FLOAT
```

Note:

Different function names "AVERAGE" and "AVE" that Vantage offers to calculate average are used along with 'avg'.

Note:

"print_variables()" is a private function used to display example outputs here. This function uses "DataFrame.show_query()" to print the query, "print(dataframe)" to display dataframe content, and "DataFrame.dtypes" to display DataFrame column types. It is not part of the teradataml package.

Supported SQL Functions

teradataml supports the following categories of SQL functions with SQLAlchemy extension:

- [Aggregate Functions](#)
- [Arithmetic, Hyperbolic and Trigonometric Functions](#)
- [Bit Byte Manipulation Functions](#)
- [Built-In Functions](#)
- [Hash Related Functions](#)
- [Regular Expression Functions](#)
- [String Functions](#)
- [Window Aggregate Functions](#)

Aggregate Functions

Note:

Examples for all aggregate functions can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Aggregate Functions using SQLAlchemy.ipynb.

Supported functions

| S/N | Function Name | Description |
|-----|---------------------|---|
| 1 | Avg / Average / Ave | Returns the arithmetic average of all values in value_expression. |

| S/N | Function Name | Description |
|-----|----------------|---|
| 2 | Corr | Returns the Sample Pearson product moment correlation coefficient of its arguments for all non-null data point pairs. |
| 3 | Count | Returns a column value that is the total number of qualified rows in value_expression. |
| 4 | Covar_pop | Returns the population covariance of its arguments for all non-null data point pairs. |
| 5 | Covar_samp | Returns the sample covariance of its arguments for all non-null data point pairs. |
| 6 | Kurtosis | Returns the kurtosis of the distribution of value_expression. |
| 7 | max / maximum | Returns a column value that is the maximum value for value_expression. |
| 8 | min / minimum | Returns a column value that is the minimum value for value_expression. |
| 9 | REGR_AVGX | Returns the mean of the independent_variable_expression for all non-null data pairs of the dependent and independent variable arguments. |
| 10 | REGR_AVGY | Returns the mean of the dependent_variable_expression for all non-null data pairs of the dependent and independent variable arguments. |
| 11 | REGR_Count | Returns the count of all non-null data pairs of the dependent and independent variable arguments. |
| 12 | REGR_Intercept | Returns the intercept of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments. |
| 13 | REGR_R2 | Returns the coefficient of determination for all non-null data pairs of the dependent and independent variable arguments. |
| 14 | REGR_SLOPE | Returns the slope of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments. |
| 15 | REGR_SXX | Returns the sum of the squares of the independent_variable_expression for all non-null data pairs of the dependent and independent variable arguments. |
| 16 | REGR_SXY | Returns the sum of the products of the independent_variable_expression and the dependent_variable_expression for all non-null data pairs of the dependent and independent variable arguments. |
| 17 | REGR_SYY | Returns the sum of the squares of the dependent_variable_expression for all non-null data pairs of the dependent and independent variable arguments. |
| 18 | Skew | Returns the skewness of the distribution of value_expression. |
| 19 | stddev_pop | Returns the population standard deviation for the non-null data points in value_expression. |
| 20 | stddev_samp | Returns the sample standard deviation for the non-null data points in value_expression. |
| 21 | sum | Returns a column value that is the arithmetic sum of value_expression. |
| 22 | var_pop | Returns the population variance for the data points in value_expression. |

| S/N | Function Name | Description |
|-----|---------------|--|
| 23 | var_samp | Returns the sample variance for the data points in value_expression. |

Note:

Grouping of columns is not allowed with DataFrame.assign() in teradataml 17.00.00.00. Always use 'drop_column = True' in DataFrame.assign().

Unsupported functions

- grouping
- pivot
- unpivot

Arithmetic, Hyperbolic and Trigonometric Functions**Note:**

Examples can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Arithmetic Functions Using SQLAlchemy.ipynb.

Supported functions

| S/N | Function Name | Description |
|-----------------------------|----------------|---|
| Arithmetic Functions | | |
| 1 | ABS | Computes the absolute value of an argument. |
| 2 | CASE_N | Evaluates a list of conditions and returns the position of the first condition that evaluates to TRUE, provided that no prior condition in the list evaluates to UNKNOWN. |
| 3 | CEIL / CEILING | Returns the smallest integer value that is not less than the input argument. |
| 4 | DEGREES | DEGREES takes a value specified in radians and converts it to degrees. |
| 5 | RADIANS | RADIANS takes a value specified in degrees and converts it to radians. |
| 6 | EXP | Raises e (the base of natural logarithms) to the power of the argument, where e = 2.71828182845905. |
| 7 | FLOOR | Returns the largest integer equal to or less than the input argument. |
| 8 | LN | Computes the natural logarithm of the argument. |
| 9 | LOG | Computes the base 10 logarithm of an argument. |
| 10 | MOD | Returns the remainder (modulus) of expr1 divided by expr2. |

| S/N | Function Name | Description |
|-------------------------|---------------|--|
| 11 | NULLIFZERO | Converts data from zero to null to avoid problems with division by zero. |
| 12 | POWER | Returns base_value raised to the power of exponent_value. |
| 13 | ROUND | Returns numeric_value rounded places_value places to the right or left of the decimal point. |
| 14 | SIGN | Returns the sign of numeric_value. |
| 16 | SQRT | Computes the square root of an argument. |
| 17 | TRUNC | Returns numeric_value truncated places_value places to the right or left of the decimal point. |
| 18 | WIDTH_BUCKET | Returns the number of the partition to which value_expression is assigned. |
| 19 | ZEROIFNULL | Converts data from null to 0 to avoid cases where a null result creates an error. |
| Hyperbolic Functions | | |
| 20 | COSH | Performs the hyperbolic or inverse hyperbolic function of an argument. |
| 21 | ACOSH | |
| 22 | SINH | |
| 23 | ASINH | |
| 24 | TANH | |
| 25 | ATANH | |
| Trigonometric Functions | | |
| 26 | SIN | Performs the trigonometric or inverse trigonometric function of an argument. |
| 27 | ASIN | |
| 28 | COS | |
| 29 | ACOS | |
| 30 | TAN | |
| 31 | ATAN | |
| 32 | ATAN2 | |

Unsupported functions

- RANDOM
- RANGE_N

Bit Byte Manipulation Functions

Note:

Examples for all bit byte manipulate functions can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Bit-Byte Manipulation Functions using SQLAlchemy.ipynb.

Supported functions

| S/N | Function Name | Description |
|-----|---------------|--|
| 1 | BITAND | Performs the logical AND operation on the corresponding bits from the two input arguments. |
| 2 | BITNOT | Performs a bitwise complement on the binary representation of the input argument. |
| 3 | BITOR | Performs the logical OR operation on the corresponding bits from the two input arguments. |
| 4 | BITXOR | Performs a bitwise XOR operation on the binary representation of the two input arguments. |
| 5 | COUNTSET | Returns the count of the binary bits within the target_arg expression that are either set to 1 or set to 0 depending on the target_value_arg value. |
| 6 | GETBIT | Returns the value of the bit specified by target_bit_arg from the target_arg byte expression. |
| 7 | ROTATELEFT | Returns an expression rotated to the left by the number of bits you specify, with the most significant bits wrapping around to the right. |
| 8 | ROTATERIGHT | Returns an expression rotated to the right by the number of bits you specify, with the least significant bits wrapping around to the left. |
| 9 | SETBIT | Sets the value of the bit specified by target_bit_arg to the value of target_value_arg in the target_arg byte expression. |
| 10 | SHIFTLEFT | Returns the expression target_arg shifted by the specified number of bits (num_bits_arg) to the left. The bits in the most significant positions are lost, and the bits in the least significant positions are filled with zeros. |
| 11 | SHIFTRIGHT | Returns the expression target_arg shifted by the specified number of bits (num_bits_arg) to the right. The bits in the least significant positions are lost, and the bits in the most significant positions are filled with zeros. |
| 12 | SUBBITSTR | Extracts a bit substring from the target_arg input expression based on the specified bit position. |
| 13 | TO_BYTE | Converts a numeric data type to the database server byte representation (byte value) of the input value. |

Built-In Functions

Note:

Examples for all built-in functions can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Built-in functions using SQLAlchemy.ipynb.

Supported functions

| S/N | Function Name | Description | Comment |
|-----|---------------------------|--------------------------------|--|
| 1 | CURRENT_DATE / CURDATE | Returns the current date. | AT LOCAL and AT TIME ZONE clauses are not supported. |
| 2 | CURRENT_TIME / CURTIME | Returns the current time. | AT LOCAL and AT TIME ZONE clauses are not supported. |
| 3 | CURRENT_TIMESTAMP | Returns the current timestamp. | AT LOCAL and AT TIME ZONE clauses are not supported. |

Hash Related Functions

Supported functions

| S/N | Function Name | Description | Comment |
|-----|---------------|---|---|
| 1 | HASHAMP | Finds the primary AMP corresponding to the hash bucket number specified in the expression and returns the AMP ID. If no hash bucket number is specified, HASHAMP returns one less than the maximum number of AMPs in the system. | <ul style="list-style-type: none"> Only expression, i.e., column can be passed to this function. MAP and COLOCATE USING clauses of the functions are not supported. |
| 2 | HASHBAKAMP | Finds the fallback AMP corresponding to the hash bucket number specified in the expression and returns the AMP ID. If no hash bucket is specified, HASHBAKAMP returns one less than the maximum number of fallback AMPs in the system. | <ul style="list-style-type: none"> Only expression, i.e., column can be passed to this function. MAP and COLOCATE USING clauses of the functions are not supported. |
| 3 | HASHBUCKET | Returns the hash bucket number that corresponds to a specified row hash value. | |

| S/N | Function Name | Description | Comment |
|-----|---------------|--|---|
| | | If no row hash value is specified, HASHBUCKET returns the highest hash bucket number. | |
| 4 | HASHROW | Returns the hexadecimal row hash value for an expression or sequence of expressions. If no expression is specified, HASHROW returns the maximum hash code value. | User can execute this either by passing expression or without any expression. |

Regular Expression Functions

Note:

Examples for all regular expression functions can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Regular Expressions Using SQLAlchemy.ipynb.

Examples in notebooks will help to understand on how values can be passed and these functions can be used in teradataml.

Supported functions

| S/N | Function Name | Description |
|-----|----------------|---|
| 1 | REGEXP_SUBSTR | Extracts a substring from source_string that matches a regular expression specified by regexp_string. |
| 2 | REGEXP_REPLACE | Replaces portions of source_string that match regexp_string with the replace_string. |
| 3 | REGEXP_INSTR | Searches source_string for a match to regexp_string. |
| 4 | REGEXP_SIMILAR | Compares source_string to regexp_string and returns integer value. |

Note:

Refer to SQL Documentation for more details on these functions and their signature.

Unsupported functions

- REGEXP_SPLIT_TO_TABLE

String Functions

Note:

Examples for all string functions can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage String Functions Using SQLAlchemy.ipynb.

Supported functions

| S/N | Function Name | Description | Comment |
|-----|---------------|--|---------|
| 1 | ASCII | Returns the decimal representation of the first character in string_expr as a NUMBER value. The decimal representation will reflect the character set of the input string. | |
| 2 | CHAR2HEXINT | Returns the hexadecimal representation for a character string. | |
| 3 | CHR | Returns the Latin ASCII character given a numeric code value. | |
| 4 | CONCAT | Concatenates string expressions. | |
| 5 | EDITDISTANCE | Returns the minimum number of edit operations (insertions, deletions, substitutions and transpositions) required to transform string1 into string2. | |
| 6 | INDEX | Returns the position in string_expression_1 where string_expression_2 starts. | |
| 7 | INITCAP | Modifies a string argument and returns the string with the first character in each word in uppercase and all other characters in lowercase. Words are delimited by white space or characters that are not alphanumeric. | |
| 8 | INSTR | Searches the source_string argument for occurrences of search_string. | |
| 9 | LEFT | Truncates in input string to a specified number of characters. The LEFT function can be called with the 'LEFT' or 'TD_LEFT' alias names. | |
| 10 | LENGTH | Returns the number of characters in the expression. | |
| 11 | LOCATE | Returns the position of the first occurrence of string_expr1 within string_expr2. The search for the first occurrence of string_expr1 begins with the first character position in string_expr2 unless the optional argument, n1, is specified. | |

| S/N | Function Name | Description | Comment |
|-----|--------------------|--|---|
| 12 | LOWER | Returns a character string identical to character_string_expression, except that all uppercase letters are replaced with their lowercase equivalents. | |
| 13 | LPAD | Returns the source_string padded to the left with the characters in fill_string so that the resulting string is length characters. | |
| 14 | LTRIM | Returns the argument expr1, with its left-most characters removed up to the first character that is not in the argument expr2. | |
| 15 | NGRAM | Returns the number of n-gram matches between string1 and string2. | |
| 16 | NVP | Extracts the value of a name-value pair where the name in the pair matches the name and the number of the occurrence specified. | |
| 17 | OREPLACE | Replaces every occurrence of search_string in the source_string with the replace_string. Use this function either to replace or remove portions of a string. | |
| 18 | OTRANSLATE | Returns source_string with every occurrence of each character in from_string replaced with the corresponding character in to_string. | |
| 19 | REVERSE | Reverses the input string. | |
| 20 | RIGHT | Starting from the end of the input string, a substring is created with the number of characters specified by the second parameter. The RIGHT function can be called with the 'RIGHT' or 'TD_RIGHT' alias names. | |
| 21 | RPAD | Returns the source_string padded to the right with the characters in fill_string so that the resulting string is length characters. | |
| 22 | RTRIM | Returns the argument expr1, with its right-most characters removed up to the first character that is not in the argument expr2. | |
| 23 | SOUNDEX | Returns a character string that represents the Soundex code for string_expression. | |
| 24 | STRING_CS | Returns a heuristically derived integer value that you can use to help determine which KANJI1-compatible client character set was used to encode string_expression. | |
| 25 | SUNSTRING / SUBSTR | Extracts a substring from a named string based on position. | Only Teradata Syntax is supported. ANSI |

| S/N | Function Name | Description | Comment |
|-----|---------------|---|--|
| | | | syntax is not supported. In SQL, both are supported. Check SQL functions documentation. |
| 26 | TRIM | Takes a character or byte string_expression argument, trims the specified pad characters or bytes, and returns the trimmed string. | Following clauses from SQL are not supported: <ul style="list-style-type: none"> • BOTH • Leading • Trailing • From • character_Set |
| 27 | UPPER / UCASE | Returns a character string identical to character_string_expression, except that all lowercase letters are replaced by their uppercase equivalents. | |

Unsupported functions

- CSV
- CSVLD
- POSITION
- STROK
- STRTOK_SPLIT_TO_TABLE
- TRANSLATE
- TRANSLATE_CHK
- VARGRAPHIC

Window Aggregate Functions

Syntax of over() function

Window aggregation offered by Vantage supports various clauses and can be used in teradataml with the help of SQLAlchemy.

For Window aggregates, one of the major tasks is to specify the window over which aggregate operation must be performed.

Specifying window in SQL is done by OVER clause, which has following syntax:


```
OVER([PARTITION BY ...] [ORDER BY ...] [RESET WHEN ...] [ROWS ... |
ROWS BETWEEN ...])
```

With SQLAlchemy, specifying window can be achieved by using the `over()` function, which has the following syntax:

```
over(partition_by = partition_expression, order_by = order_expression, rows =
(p, f))
```

Where:

- *partition_expression*: column expression to partition the data on.

For example: `df.column_name.expression`

- *order_expression*: column expression to sort the data.

Sorting can be done in either Ascending or Descending order with NULLS FIRST or NULLS LAST.

For example:

- Default sorting: `df.column_name.expression`
- Ascending Order: `df.column_name.expression.asc()`
- Descending Order: `df.column_name.expression.desc()`
- With NULLS FIRST - `df.column_name.expression.nullsfirst()`
- With NULLS LAST - `df.column_name.expression.nullslast()`
- Ascending Order with NULLS LAST - `df.column_name.expression.asc().nullslast()`
- *rows*: generates the syntax for 'ROWS BETWEEN'.

To perform windowed aggregate function over a window using ROWS and ROWS BETWEEN, you must use argument *rows* that accepts a tuple (p, f). *p* and *f* can accept the following values. Each value passed to *p* and *f* has different meaning or results in different syntax. SQL syntax will be generated based on these values.

- Negative Value: indicates a preceding value;
- Positive Value: indicates a following value;
- 0: for Current row;
- None: unbounded value.

Example

This section shows how to specify a window and use Window aggregate functions.

Note:

More examples and details can be found at: `<pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Window Aggregate Functions using SQLAlchemy.ipynb`.

In this example, you will return the first value in column 'gpa', over a window of 3 values preceding the current row and 1 value following the current row.

- Use the func module to generate a function object "FIRST_VALUE_".

```
# Example returns by id first gpa in the moving average group.
>>> FIRST_VALUE_ =
func.FIRST_VALUE(admissions_train.gpa.expression).over(order_by=admissions_train.id.expression, rows=(-3, 1))
```

- View the type of the function object "FIRST_VALUE_".

```
>>> type(FIRST_VALUE_)
```

```
sqlalchemy.sql.elements.Over
```

- Pass the function object "FIRST_VALUE_" to DataFrame.assign() function to return the first value in column 'gpa'.

```
>>> fv_gpa_df = admissions_train.assign(FIRST_VALUE_gpa=FIRST_VALUE_)

>>> print_variables(fv_gpa_df, "FIRST_VALUE_gpa")
```

Equivalent SQL: select id AS id, masters AS masters, gpa AS gpa, stats AS stats, programming AS programming, admitted AS admitted, FIRST_VALUE(gpa) OVER (ORDER BY id ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) AS "FIRST_VALUE_gpa" from "admissions_train"

```
***** DataFrame *****
  masters  gpa  stats programming  admitted  FIRST_VALUE_gpa
id
16      no  3.70  Advanced   Advanced         1           4.00
25      no  3.96  Advanced   Advanced         1           3.46
26     yes  3.57  Advanced   Advanced         1           3.59
1      yes  3.95  Beginner   Beginner         0           3.95
3       no  3.70   Novice    Beginner         1           3.95
34     yes  3.85  Advanced   Beginner         0           3.50
35      no  3.68   Novice    Beginner         1           3.46
36      no  3.00  Advanced   Novice         0           3.55
2      yes  3.76  Beginner   Beginner         0           3.95
24      no  1.87  Advanced   Novice         1           3.87
```

```
***** DataFrame.dtypes *****
```

```

id                int
masters           str
gpa               float
stats            str
programming       str
admitted          int
FIRST_VALUE_gpa   float

```

'FIRST_VALUE_gpa' Column Type: FLOAT

Supported functions

| S/N | Function Name | Description | Comment |
|-----|---------------|---|---------|
| 1 | CSUM | Returns the cumulative (or running) sum of a value expression for each row in a partition, assuming the rows in the partition are sorted by the sort_expression list. | |
| 2 | CUME_DIST | Calculates the cumulative distribution of a value in a group of values. | |
| 3 | DENSE_RANK | Returns an ordered ranking of rows based on the value_expression in the ORDER BY clause. | |
| 4 | FIRST_VALUE | Returns the first value in an ordered set of values. | |
| 5 | LAST_VALUE | Returns the last value in an ordered set of values. | |
| 6 | LAG | <p>Ordered analytic functions calculate an aggregate or non-aggregate value on a window of rows within a group of rows. The window of rows is defined by the Window Framing clause, also called the ROWS clause. Window sizes are based on the size specified in the ROWS clause. The group of rows is defined by the PARTITION BY clause of the Window function.</p> <p>The LAG function accesses data from the row preceding the current row at a specified offset value in a window group. If the offset value is outside the scope of the window, the user-specified default value is returned.</p> | |
| 7 | LEAD | <p>Ordered analytic functions calculate an aggregate or non-aggregate value on a window of rows within a group of rows. The window of rows is defined by the Window Framing clause, also called the ROWS clause. Window sizes are based on the size specified in the ROWS clause. The group of rows is defined by the PARTITION BY clause of the Window function.</p> <p>The LEAD function returns data from the row following the current row. If the offset value is outside the scope of the window, the user-specified default value is returned.</p> | |

| S/N | Function Name | Description | Comment |
|-----|-----------------------------------|---|--|
| 8 | MAVG | Computes the moving average of a value expression for each row in a partition using the specified value expression for the current row and the preceding width-1 rows. | |
| 9 | MDIFF | Returns the moving difference between the specified value expression for the current row and the preceding width rows for each row in the partition. | |
| 10 | MEDIAN | For numeric values, returns the middle value or an interpolated value that would be the middle value after the values are sorted. Nulls are ignored in the calculation. | Supported with drop_columns=True |
| 11 | MLINREG | Returns a predicted value for an expression based on a least squares moving linear regression of the previous width -1 (based on sort_expression) column values. | |
| 12 | MSUM | Computes the moving sum specified by a value expression for the current row and the preceding n-1 rows. This function is very similar to the MAVG function. | |
| 13 | PERCENT_RANK | Returns the relative rank of rows for a value_expression. | |
| 14 | PERCENTILE_CONT / PERCENTILE_DISC | Returns an interpolated value that falls within its value_expression with respect to its sort specification. | Supported with drop_columns=True |
| 15 | QUANTILE | Computes the quantile scores for the values in a group. | |
| 16 | RANK(ANSI) | Returns an ordered ranking of rows based on the value_expression in the ORDER BY clause. | Supported without "with ties" and "RESET WHEN" |
| 17 | RANK(Teradata) | Returns the rank (1 ... n) of all the rows in the group by the value of sort_expression list, with the same sort_expression values receiving the same rank. | |
| 18 | ROW_NUMBER | Returns the sequential row number, where the first row is number one, of the row within its window partition according to the window ordering of the window. | |

In addition to these functions, all 23 aggregate functions mentioned in the [Aggregate Functions](#) section are also supported.

To use these aggregate functions as Window aggregates, you must add a window to the same using `over()` function from SQLAlchemy, as show cased in the example notebook of Window Aggregates. Refer to the example for AVG function for details.

For Window Aggregate functions to work, you must specify a window on top of these aggregate functions, using SQLAlchemy. There are various types of Windows that you can specify:

- Cumulative / Expanding
- Moving / Rolling
- Remaining / Contracting
- Grouping

All of these types computation can be performed with teradataml.

Examples for the these can be found at: <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\Teradata Vantage Window Aggregate Functions using SQLAlchemy.ipynb.

Note:

The following Teradata SQL syntax are not supported by SQLAlchemy:

- RESET WHEN clause
- ROWS UNBOUNDED PRECEDING
- ROWS value PRECEDING
- ROWS CURRENT ROW

Note:

SQLAlchemy offers range over in Window aggregate, Teradata does not support the same.

For example, "RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING" is not supported.

Using SQLAlchemy Clause Element and Expression

teradataml is extended to work with different SQLAlchemy clause elements, expressions and functions. These can be used in teradataml "DataFrame.assign()" method or as filter expression in slice filter ([]) for teradataml DataFrame.

This section describes how the SQLAlchemy clause elements, expressions, and functions can be used with teradataml.

Using Basic SQLAlchemy ClauseElement and Expression in DataFrame.assign()

With the help of SQLAlchemy, in teradataml, user not only can run SQL functions, but also can run some other features that Vantage offers. Basically, user can use SQLAlchemy ClauseElements, representing select_expression clauses in query.

teradataml can use SQLAlchemy ClauseElements to do following:

- Build a CASE expression on teradataml DataFrame column and select the using DataFrame.assign().
- Cast a teradataml DataFrame column.
- Select distinct values from a teradataml DataFrame Column.

- Extract values from teradataml column, using EXTRACT function.

Refer to example notebook <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\teradataml assign() - Using Generic SQLAlchemy ClauseElements (Case, Cast, Extract, Distinct).ipynb to learn about these.

Using Basic SQLAlchemy ClauseElement and Expression for Filtering

Vantage offers various types of clauses in WHERE clause that allows users to filter the results in a table.

With the help of SQLAlchemy ClauseElements and Expression, teradataml can use the following SQL clauses to filter results in teradataml DataFrame:

- CASE expression
- BETWEEN ... AND ... clause
- IN clause
- NOT IN clause
- LIKE clause
- NOT LIKE clause
- ILIKE clause
- IS NULL
- IS NOT NULL

Refer to example notebook <pkg_install_location>\teradataml\data\notebooks\sqlalchemy\teradataml filtering - Using SQLAlchemy ClauseElements.ipynb to learn about these.

Data Type Mapping

Data Type Mapping between Advanced SQL Engine, teradataml DataFrame dtypes and Python

This table lists the data type mapping between Analytics Database, teradataml DataFrame dtypes, and Python.

| Analytics Database Data Type | teradataml DataFrame dtypes | Python |
|------------------------------|-----------------------------|----------|
| BIGINT | int | int |
| BLOB(n) | bytes | bytes |
| BYTE(n) | bytes | bytes |
| BYTEINT | int | int |
| CHAR(n) | str | int |
| CLOB(n) | str | str |
| DATE FORMAT 'YY/MM/DD' | datetime.date | datetime |
| DECIMAL(p,q) / Numeric | decimal.decimal | decimal |
| DOUBLE PRECISION | float | decimal |
| FLOAT | float | float |
| INTEGER | int | int |
| INTERVAL DAY(n) | str | str |
| INTERVAL DAY(n) TO MINUTE | str | str |
| INTERVAL DAY(n) TO SECOND(n) | str | str |
| INTERVAL HOUR(n) | str | str |
| INTERVAL MINUTE(n) | str | str |
| INTERVAL MONTH(n) | str | str |
| INTERVAL SECOND(p,q) | str | str |
| INTERVAL YEAR(n) | str | str |
| NUMBER(p,q) | decimal.decimal | decimal |
| PERIOD(DATE) | str | str |

| Analytics Database Data Type | teradataml DataFrame dtypes | Python |
|-------------------------------------|-----------------------------|----------|
| PERIOD(TIME(n)) | str | str |
| PERIOD(TIME(n) WITH TIME ZONE) | str | str |
| PERIOD(TIMESTAMP(n) WITH TIME ZONE) | str | str |
| REAL | float | float |
| SMALLINT | int | int |
| TIME(n) | datetime.time | datetime |
| TIME(n) WITH TIME ZONE | datetime.time | datetime |
| TIMESTAMP(n) | datetime.datetime | datetime |
| TIMESTAMP(n) WITH TIME ZONE | datetime.datetime | datetime |
| VARBYTE(n) | bytes | bytes |
| VARCHAR(n) | str | str |
| XML | str | str |

Model Cataloging

Note:

teradataml Model Cataloging feature is a deprecated feature in this release and is not guaranteed to work. It will be removed in future release versions.

Note:

Some examples in the following sections use ML Engine analytic functions. These functions require that your system has the Vantage Machine Learning Engine, which is a separate machine learning legacy engine that is not part of the current standard Vantage ClearScape Analytics offer/capability. If your Vantage system does not have the required ML Engine, an error or no-op behavior will occur when these functions are invoked.

Model Cataloging allows users to save model-related information in a way that it can be reused by the supported functions of Machine Learning Engine or Analytics Database via SQL, Teradata Package for Python (teradataml) or Teradata Package for R (tdplyr).

For example, an ML Engine DecisionForest model saved by using SQL can be retrieved to use with teradataml for scoring with the DecisionForestPredict function from ML Engine or Analytics Database. Similarly, an ML Engine or Analytics Database model saved by using tdplyr can be described and retrieved by teradataml. See [save_model\(\)](#) for more information on what information is saved.

teradataml offers APIs to use Model Cataloging, allowing users to:

- Save information and related objects to the catalog for the model created using the teradataml analytic functions, using [save_model\(\)](#);
- List the saved models, using [list_models\(\)](#);
- Describe a model, using [describe_model\(\)](#);
- Retrieve a model for reuse, using [retrieve_model\(\)](#);
- Publish a model to set the access level and status of saved model, using [publish_model\(\)](#);
- Delete a model, using [delete_model\(\)](#).

It also provides mechanisms to persist models with teradataml, which would have been garbage collected otherwise.

Note:

Currently, teradataml only supports model catalog operations on ML Engine or Analytics Database generated models.

To use the teradataml offering related to model cataloging, the Model Catalog must be set up on Vantage system.

The scripts along with the instructions to set up the Model Catalog can be downloaded from the **Teradata Package for Python - teradataml** page on <https://downloads.teradata.com/>.

save_model()

The `save_model()` API allows a user to save model-related information to the Model Catalog, and also persist the model's output DataFrames to tables on Vantage (which would have otherwise been purged at the end of the session) without having to call the [copy_to_sql\(\)](#) or the [to_sql\(\) DataFrame Method](#).

This allows the user to create a connection anytime later using a fresh `teradataml` (or even a SQL) session to use the saved models in workflows.

For example, a user can create a GLM model using the ML Engine GLM function (`teradataml.analytics.mle.GLM.GLM`), and then save this model by calling the `save_model()` function.

The `save_model()` function persists not only the model's result DataFrames, but also all the arguments to the model generating function and some other details as mentioned in the following list, so that a user can understand how the model is generated:

- The target column from the training data input to the model, when available.

Note:

The target column is currently saved only for the functions that use formula, and when the formula is used, in case it is optional.

- The prediction type for the function: CLASSIFICATION, REGRESSION, CLUSTERING, or OTHER.
- The engine used to generate the model: ML Engine, or Analytics Database.
- The client language used to generate the model: `teradataml`.
- The name of the algorithm associated with the function.
- The time in seconds required to run the model generating function.
- The model creating user's name.
- The status and access level of the model (see the [publish_model\(\)](#) section).
- The location where the model is stored: Analytics Database.
- The date and time when the model is saved.
- If possible, the name of the table corresponding to the DataFrames passed as input to the model generating function along with their dimensions.
- The user provided `model_project`, `entity_target`, and `performance_metrics`.

Once the model is saved, all of this information is available in the output of [describe_model\(\)](#).

Note:

A model can be saved only by the creator of the model.

The required arguments:

- *model* specifies the `teradataml` analytic function model to be saved;

- *name* specifies the unique name to identify the saved model;
- *description* specifies a note describing the model to be saved.

The optional arguments *model_project* specifies the project that the model is associated with, and *entity_target* specifies a group or team that the model is associated with.

Another optional argument *performance_metrics* specifies the performance metrics for the model, as a dictionary of the form { "<metric>" : { "measure" : <value> }, ... }. For example: { "AUC" : { "measure" : 0.5 }, ... }.

Example

- Load the data.

```
>>> # Load the data to run the example
>>> load_example_data("decisionforest", ["housing_train"])
```

- Create teradataml DataFrame.

```
>>> # Create teradataml DataFrame objects.
>>> housing_train = DataFrame.from_table("housing_train")
```

- Create a classification tree that can be input to the DecisionForestPredict.

```
>>> # This example uses home sales data to create a
>>> # classification tree that predicts home style, which can be input
>>> # to the DecisionForestPredict.

>>> formula = "homestyle ~ driveway + recroom + fullbase + gashw + airco +
prefarea + price + lotsize + bedrooms + bathrms + stories + garagepl"
>>> rft_model = DecisionForest(data=housing_train,
                               formula = formula,
                               tree_type="classification",
                               ntree=50,
                               tree_size=100,
                               nodesize=1,
                               variance=0.0,
                               max_depth=12,
                               maxnum_categorical=20,
                               mtry=3,
                               mtry_seed=100,
                               seed=100
                               )
```

- Save the generated model.

```
>>> # Let's save this generated model.
>>> save_model(model=rft_model, name="decision_forest_model",
```

```
description="Decision Forest test")
Persisting model information.
Persisted table: "ALICE"."ml__td_decisionforest0_1589787736719763"
Persisted table: "ALICE"."ml__td_decisionforest1_1589794611679956"
Persisted table: "ALICE"."ml__td_sqlmr_out__1589787498246673"
Successfully persisted model.
```

As the output message suggests, the model is saved successfully.

list_models()

The `list_models()` function provides the user with the ability to list all models that are accessible to the user, filtering the results based on the following optional arguments:

- The model name - using the argument *name*;
- The name of the algorithm that was used to generate the model - using the argument *algorithm_name*;
- The model generating engine - using the argument *engine*;
- Flag to list models created by or accessible to the user - using the argument *accessible*:
 - If True, all models to which the user has access are listed.
 - If False, all models created by the user are listed.
- Flag to list models with the access level set to PUBLIC - using the argument *public*.

See the [publish_model\(\)](#) section for more details on this argument.

Note:

A user can only see the models that he or she has access to.

Example Prerequisites

Follow the steps in [save_model\(\)](#) to create a classification tree model that can be input to `DecisionForestPredict` and save the generated model.

Example: List all models saved and accessible to the current user.

```
>>> list_models()
      ModelName  ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy      CreatedDate
0 decision_forest_model DecisionForest      ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

Example: List models accessible to the user with name = 'decision_forest_model'.

You can pass a substring as well.

```
>>> list_models(name = "decision_forest_model")
      ModelName ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy      CreatedDate
0 decision_forest_model DecisionForest      ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

Example: List all models accessible to the user, with specific algorithm name.

This example lists all models accessible to the user with algorithm name 'DecisionForest'. You can pass a substring as well.

```
>>> list_models(algorithm_name = "DecisionForest")
      ModelName ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy      CreatedDate
0 decision_forest_model DecisionForest      ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

Example: List all models accessible to user, with specific algorithm name and model name.

This example lists all models accessible to user with algorithm name 'DecisionForest' and model name containing string 'forest'.

```
>>> list_models(name = "forest", algorithm_name = "DecisionForest")
      ModelName ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy      CreatedDate
0 decision_forest_model DecisionForest      ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

Example: List all models accessible to user, with specific algorithm name and engine name.

This example lists all models accessible to user with algorithm name 'DecisionForest' and model generated using 'ML Engine'.

```
>>> list_models(algorithm_name = "DecisionForest", engine = "ML Engine")
      ModelName ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy      CreatedDate
0 decision_forest_model DecisionForest      ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

Example: List all models created by the user, with specific algorithm name, engine name and access level.

This example lists all models created by the user with algorithm name 'DecisionForest' and model generated using 'ML Engine', and access not set to PUBLIC.

```
>>> list_models(algorithm_name = "DecisionForest", engine = "ML Engine",
accessible = False)
           ModelName  ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy          CreatedDate
0  decision_forest_model  DecisionForest          ML Engine
teradataml      ALICE  2020-05-17 23:59:48.740000
```

The `list_models()` function outputs basic information about the models that are accessible to the user and match any filter criteria that may have been provided.

For details about a specific model, use the [describe_model\(\)](#) function.

describe_model()

The `describe_model()` function lists the following details of a model:

- Model name provided while saving the model.
- Description provided while saving the model.
- Algorithm.
- Prediction type.
- Target column, if any.
- Project provided while saving the model.
- Entity target.
- Name of the engine that generated the model.
- Name of the client that generated the model.
- The access level set for the model.
- The status of the model.
- The time required to run the model generating function.
- The name of the user who created the model.
- The date and time when the model was saved.
- The arguments that were passed to the model generating function.
- The input details, if they were saved.
- The output names and the names of the underlying tables which are the actual model tables for the model saved.

The required argument *name*, which is also the only argument, specifies the name of the model to list the details for.

Note:

A user can only describe the models that he or she has access to.

Example Prerequisites

Follow the steps in [save_model\(\)](#) to create a classification tree model that can be input to DecisionForestPredict and save the generated model.

Example

List all details of the saved model 'decision_forest_model'.

```
>>> # List all details of recently saved model 'decision_forest_model'.
>>> describe_model(name="decision_forest_model")

*** 'decision_forest_model': Model Details ***
ModelName                decision_forest_model
ModelDescription          Decision Forest test
ModelAlgorithm            DecisionForest
ModelPredictionType       CLASSIFICATION
ModelTargetColumn         homestyle
ModelProject              None
ModelEntityTarget         None
ModelGeneratingEngine     ML Engine
ModelGeneratingClient     teradataml
ModelAccess               Private
ModelStatus               In-Development
ModelBuildTime            0
ModelLocation             Advanced SQL Engine
CreatedBy                 ALICE
CreatedDate               2020-05-17 23:59:48.740000

*** 'decision_forest_model': Model Attributes ***
AttrName                  AttrValue
0      mtry_seed           100
1      nodesize            1
2      mtry                 3
3      outofbag             False
4      seed                 100
5      display_num_processed_rows  False
6      ntree                50
7      formula             homestyle ~ driveway + recroom + fullbase + ga...
8      maxnum_categorical   20
9      max_depth            12
10     categorical_encoding graycode
11     tree_type            classification
12     variance             0
13     tree_size            100

*** 'decision_forest_model': Model Training Data ***
InputName      InputTableName  NRows  NCols
0      data    "ALICE"."housing_train"  492    14

*** 'decision_forest_model': Model Training Objects ***
OutputName      OutputTableName
0      monitor_table  "ALICE"."ml_td_decisionforest1_1589794611679956"
1      predictive_model  "ALICE"."ml_td_decisionforest0_1589787736719763"
2      output          "ALICE"."ml_td_sqlmr_out__1589787498246673"
```

retrieve_model()

The `retrieve_model()` API allows a user to recreate a `teradataml` Analytic Function object from the information saved with the Model Catalog using `save_model()`. This analytic function object can then be used in the user's workflow.

For example, if a user has built a user-base classification model, he or she can save the information related to this model and reuse it at every instance when there is need to score a new set of user data.

The required argument *name*, which is also the only argument, specifies the name of the model to retrieve.

Note:

A user can only retrieve the models he or she has access to.

Example Prerequisites

Follow the steps in [save_model\(\)](#) to create a classification tree model that can be input to `DecisionForestPredict` and save the generated model.

Example

- View the saved models.

```
>>> list_models()
           ModelName  ModelAlgorithm ModelGeneratingEngine
ModelGeneratingClient CreatedBy          CreatedDate
0  decision_forest_model  DecisionForest          ML Engine
teradataml      ALICE   2020-05-17 23:59:48.740000
```

- Retrieve the saved model.

```
>>> # Retrieve the saved model
>>> retrieved_rft_model = retrieve_model("decision_forest_model")
```

- Use the retrieved model in predict.

```
>>> # Use the retrieved model in predict.
>>> decision_forest_predict_out = DecisionForestPredict(object
= retrieved_rft_model,
                                                    newdata
= housing_test,
                                                    id_column = "sn",
                                                    detailed = False,
                                                    terms = ["homestyle"],
newdata_order_column=['sn', 'price'],
```



```
object_order_column=['worker_ip', 'task_index']
)
```

- Print the result.

```
>>> # Print the results.
>>> print(decision_forest_predict_out.result)
##### STDOUT Output #####

  homestyle   sn prediction  confidence_lower  confidence_upper
0   Classic  260   Classic           0.90           0.90
1   Classic   13   Classic           0.98           0.98
2   Classic  142   Classic           0.86           0.86
3 Eclectic   53   Eclectic           0.98           0.98
4 Eclectic   38   Eclectic           0.90           0.90
5   Classic  111   Classic           0.96           0.96
6   Classic  251   Classic           0.76           0.76
7 Eclectic  408   Eclectic           0.96           0.96
8   Classic   16   Classic           0.90           0.90
9   Classic  132   Classic           0.92           0.92
```

publish_model()

Use the `publish_model()` function to set the access level of a saved model in the catalog, along with the status of the model.

For example, the status of a model can be 'In-Development' when the model is being fine tuned. Once it is tested, it can be set to 'Active' or 'Production'.

The required argument *name* specifies the name of the model to be published.

At least one of the following two optional arguments must be specified:

- *grantee*: specifies a user or role to update the model's access level to one of the following:
 - Private

When a model is saved using the `save_model()` function, the access is set to 'Private'.
 - Public

A user can set the access level to Public by passing 'PUBLIC' as the value to the *grantee* argument.
 - Team

The access level is set to Team when the user passes a user or a role other than PUBLIC, or his or her own username to the *grantee* argument.

Note:

Currently,

- The `publish_model()` function allows the setting of access level only when the current access level is 'Private', and setting to 'Private' from any other access level is not supported.
- When setting a *grantee* (or access level), the `publish_model()` function shows the user which model tables need to be granted access to the *grantee*. Once these SQL GRANT commands are issued, the model becomes accessible to the intended user(s).

- *status*: specifies a string to set the status of the model to one of the following permitted values:
 - Active
 - Retired
 - Candidate
 - Production
 - In-Development

Note:

A model can only be published by its creator.

Example Prerequisites

Follow the steps in [save_model\(\)](#) to create a classification tree model that can be input to `DecisionForestPredict` and save the generated model.

Example: Update only the access.

```
>>> publish_model('decision_forest_model', grantee='john')
Model published successfully!
Please execute the following GRANT statements:
GRANT SELECT ON "ALICE"."ml__td_decisionforest0_1589787736719763" to john;
GRANT SELECT ON "ALICE"."ml__td_decisionforest1_1589794611679956" to john;
GRANT SELECT ON "ALICE"."ml__td_sqlmr_out__1589787498246673" to john;
```

To make sure 'John' can access this model's information, you must issue the SQL GRANT statements as suggested in the output message.

You can use `describe_model()` function to check the update.

Example: Update only the status.

```
>>> publish_model('decision_forest_model', status='Active')
Model published successfully!
```

You can use `describe_model()` function to check the update.

Example: Update both the access and status.

```
>>> publish_model('decision_forest_model', grantee='PUBLIC', status='Production')
Model published successfully!
Please execute the following GRANT statements:
GRANT SELECT ON "ALICE"."ml__td_decisionforest0_1589787736719763" to PUBLIC;
GRANT SELECT ON "ALICE"."ml__td_decisionforest1_1589794611679956" to PUBLIC;
GRANT SELECT ON "ALICE"."ml__td_sqlmr_out__1589787498246673" to PUBLIC;
```

To make sure the role 'PUBLIC' can access this model's information, you must issue the SQL GRANT statements as suggested in the output message.

You can use `describe_model()` function to check the update.

delete_model()

Use the `delete_model()` API to remove the information of a saved model from the Model Catalog, and optionally also delete the model tables associated with the model.

The required argument *name* specifies the name of the model to be deleted.

The optional argument *delete_objects* specifies whether to drop the model objects as well:

- If True, the model objects related to the model are deleted/dropped.
- If False, only the information from the catalog will be removed.

This is the default value.

Note:

A model can only be deleted by its creator.

Example Prerequisites

Follow the steps in [save_model\(\)](#) to create a classification tree model that can be input to `DecisionForestPredict` and save the generated model.

Example: Only delete model information from the Model Catalog.

```
>>> delete_model('decision_forest_model')
Deleted model 'decision_forest_model' successfully.
Model Objects that can be dropped:
['"ALICE"."ml__td_decisionforest0_1589787736719763"',
'"ALICE"."ml__td_decisionforest1_1589794611679956"',
'"ALICE"."ml__td_sqlmr_out__1589787498246673"'].
```

Example: Delete model information from the Model Catalog and drop model objects as well.

```
>>> delete_model('decision_forest_model', True)
Deleted model 'decision_forest_model' successfully.
Model Objects dropped successfully.
```

ML Engine Specific Settings and Examples

The teradataml settings and examples in this section are only applicable if your system has the Vantage Machine Learning Engine, which is a separate machine learning legacy engine that is not part of the current standard Vantage offer.

If your Vantage system does not have the required ML Engine, an error or no-op behavior will occur when the listed functions are invoked.

Significance of formula argument in teradataml Analytic Functions

teradataml offers some analytic functions with the argument 'formula'.

The 'formula' argument accepts a string that specifies the model to be fitted for an analytic function. It accepts string of the following format:

```
"Y ~ X1 + X2 + X3 + .. + Xn"
```

Where,

- 'y' is a dependent variable, that is to say, a response column;
- 'X1, X2, X3, .., Xn' are independent variables.

The 'formula' argument as a string allows you to specify dependent and independent variables in its simplest form. In teradataml, analytic functions require names of these dependent and independent variables. teradataml extracts the required information from 'formula' to retrieve dependent variables and independent variables. It further processes the independent variables to classify those as either categorical or numerical and uses the same while running the analytic functions. Data for these dependent and independent variables is automatically picked up internally in Vantage when the analytical function is run. Thus, we accept dependent or independent variables as a string in 'formula'. No extra processing or data structures are required to achieve the same.

In case you want to use all the variables except response variable as independent variables. you can pass 'formula' as " Y ~ . .".

teradataml internally classifies dependent variables as either categorical or numerical, based on the types of the columns in the DataFrame. You can explicitly specify how teradataml treat columns as categorical even though they are of numerical type. To do so, teradataml provides 'as_categorical()' function.

For example, columns X1, X3 are of numerical type. To treat them as categorical, you can provide 'formula' as follows:

```
"Y ~ X2 + .. + Xn + {}".format(as_categorical(["X1", "X3"]))
```

The following examples shows how to use 'formula' in various ways.

For example, Binary Classification is used on the housing data to build a model using training data that contains couple of labels (Responses) - classic and eclectic, specifying the style of a house, based on the 12 other attributes of the house, such as bedrooms, stories, price and so on. The following examples show various ways to specify dependent variables in formula argument.

A list of numerical and categorical columns contained in an input DataFrame:

- Numerical Columns: 'lotsize', 'price', 'stories', 'garagepl', 'bedrooms', 'bathrms', 'sn'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom', 'homestyle'

Example Prerequisites

```
>>> from teradataml import *
```

Load the required dataset and create a DataFrame on the same.

```
>>> load_example_data("xgboost", ["housing_train_binary"])
```

```
>>> housing_train_binary = DataFrame.from_table("housing_train_binary")
>>> display.print_sqlmr_query = True
```

Print columns and types.

```
>>> housing_train_binary.dtypes
sn                int
price             float
lotsize           float
bedrooms          int
bathrms           int
stories           int
driveway          str
recroom           str
fullbase          str
gashw             str
airco             str
garagepl          int
prefarea          str
homestyle         str
```

Example 1: Basic example of using formula, classify arguments based on their Default types

```
formula = "homestyle ~ lotsize + fullbase + stories + driveway + prefarea + gashw
+ airco + garagepl + price + recroom + bedrooms + bathrms"
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl', 'bedrooms', 'bathrms'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom'

```
>>> formula = "homestyle ~ lotsize + fullbase + stories + driveway + prefarea +
gashw + airco + garagepl + price + recroom + bedrooms + bathrms"
```

```
>>> XGBoost_out1 = XGBoost(data=housing_train_binary,
...                          id_column='sn',
...                          formula=formula,
...                          num_boosted_trees=2,
...                          loss_function='binomial',
...                          prediction_type='classification',
...                          reg_lambda=1.0,
...                          shrinkage_factor=0.1,
...                          iter_num=10,
...                          min_node_size=1,
...                          max_depth=10
...                          )
```

```
SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574668544765277)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')

NumericInputs('lotsize','stories','garagepl','price','bedrooms','bathrms')

CategoricalInputs('fullbase','driveway','prefarea','gashw','airco','recroom')
) as sqlmr
```

```
>>> XGBoost_out1
##### STDOUT Output #####

0          message
1          Parameters:
2      \tNumber of total trees (all subtrees): 20
3      \tColumnSubSampling Features: 12
4      \tMaxDepth : 10
5      \tNumber of boosted trees : 2
6  XGBoost model created in table specified in Ou...
7      \tVariance : 0.0
8      \tPrediction Type : CLASSIFICATION
9      \tMinNodeSize : 1
10     \tNumber of boosting iterations : 10
```

```
##### model_table Output #####
```

```

tree_id iter class_num
tree
0 0 4 0
{"sum_":0.05052627000000515,"sumSq_":119.01992... {"1536":0.05461976,"1537":0.039027166,"769":0...
1 0 8 0
{"sum_":2.2463420700000007,"sumSq_":71.3756123... {"1792":0.02246395,"1536":0.033232495,"1793":0...
2 1 1 0
{"sum_":-2.399999989632917E-6,"sumSq_":193.644... {"1792":0.07149073,"1536":0.07176345,"1664":0...
3 1 5 0
{"sum_":-0.41850037999999556,"sumSq_":103.1428... {"136":-0.04735258,"1168":-0.06423314,"1169":-...
4 1 8 0
{"sum_":-3.291647789999995,"sumSq_":70.1287556... {"10":-0.04521699,"267":-0.042273022,"268":-0...
5 0 7 0
{"sum_":2.032135579999999,"sumSq_":80.44522018... {"1536":0.03432062,"1537":0.04007167,"1538":0...
6 -1 -1 -1 {"classifier":"CLASSIFICATION","lossType":"BIN...
7 1 3 0
{"sum_":0.62826041999999869,"sumSq_":137.453551... {"1152":-0.08542035,"512":-0.056720935,"1153":...
8 0 3 0
{"sum_":-0.8873116000000039,"sumSq_":137.04920... {"1536":0.056829628,"1537":0.04017536,"769":0...
9 1 9 0
{"sum_":-3.8959221599999942,"sumSq_":62.115716... {"260":-0.03840291,"10":-0.043216076,"522":-0...

```

Example 2: Use all features and columns in a DataFrame as independent features

This examples uses all features and columns in a DataFrame as independent features, except dependent one 'homestyle'. Use dot (.) notation.

```
formula ="homestyle ~ ."
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl', 'bedrooms', 'bathrms', 'sn'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom'

```
>>> formula ="homestyle ~ ."
```

```

>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
...                   reg_lambda=1.0,
...                   shrinkage_factor=0.1,
...                   iter_num=10,
...                   min_node_size=1,
...                   max_depth=10
...                   )

```

```

SELECT * FROM XGBoost(
  ON "housing_train_binary" AS InputTable
  OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574668651101580)
  USING

```



```

        IdColumn('sn')
        NumBoostedTrees('2')
        LossFunction('binomial')
        PredictionType('classification')
        MaxDepth('10')
        ResponseColumn('homestyle')

NumericInputs('bathrms','bedrooms','lotsize','stories','garagepl','sn','price')

CategoricalInputs('fullbase','gashw','recroom','prefarea','driveway','airco')
) as sqlmr

```

```

>>> result
##### STDOUT Output #####

0                                     message
1           Parameters:
2           \tNumber of total trees (all subtrees): 20
3           \tMaxDepth : 10
4           \tMinNodeSize : 1
5           \tColumnSubSampling Features: 13
6           XGBoost model created in table specified in Ou...
7           \tVariance : 0.0
8           \tNumber of boosted trees : 2
9           \tNumber of boosting iterations : 10
           \tRegularization : 1.0

##### model_table Output #####

  tree_id  iter  class_num      region_prediction
tree
0         0       7         0
{"sum_-":0.7908522399999951,"sumSq_":78.1650401... {"1284":-0.042114023,"1285":-0.06251619,"1286"...
1         1      10         0
{"sum_-":-0.3352809000000364,"sumSq_":55.00726... {"1800":0.03133564,"9":-0.040307123,"1801":0.0...
2         1       2         0
{"sum_-":1.2196361999999998,"sumSq_":161.1797732... {"1024":-0.10994268,"257":-0.060235456,"513":-...
3         1       3         0
{"sum_-":1.3794220799999997,"sumSq_":138.6020714... {"1152":-0.0758285,"256":-0.09012855,"129":-0...
4         0       6         0
{"sum_-":0.5030569700000029,"sumSq_":88.9128124... {"768":0.026927602,"769":0.026302144,"1282":-0...
5         0      10         0
{"sum_-":2.18527407,"sumSq_":53.63244861988457,... {"771":0.033950675,"1540":0.03275624,"1541":0...
6         1       9         0
{"sum_-":-0.8708544000000003,"sumSq_":62.909632... {"1288":-0.037777215,"9":-0.050687656,"1289":-...
7         1       1         0
{"sum_-":2.3999999949619877E-6,"sumSq_":193.64... {"1792":0.07149073,"1536":0.07176345,"1920":0...
8         0       2         0
{"sum_-":-0.07000995000000743,"sumSq_":156.7885... {"1152":-0.096734196,"384":0.0422824,"1153":-0...
9         1       6         0
{"sum_-":-0.03682765999999926,"sumSq_":91.18234... {"256":-0.05808351,"1280":-0.047064222,"1281":...

```

Example 3: Use only as_categorical, all numeric columns explicitly specified in list

```

formula ="homestyle ~ {}".format(as_categorical(['stories', 'garagepl',
'bedrooms', 'bathrms']))

```

For numeric columns 'stories', 'garagepl', 'bedrooms', 'bathrms' in the DataFrame, to treat these columns as categorical in analytic function execution, pass the numeric columns as input to the `as_categorical()` function first, and combine the output with formula string to be passed to the *formula* argument of analytic function.

In this example, dependent variables are classified as:

- Numeric columns: None

- Categorical Columns: 'stories', 'garagepl', 'bedrooms', 'bathrms'

```
>>> formula = "homestyle ~ {}".format(as_categorical(['stories', 'garagepl',
'bedrooms', 'bathrms']))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
...                   reg_lambda=1.0,
...                   shrinkage_factor=0.1,
...                   iter_num=10,
...                   min_node_size=1,
...                   max_depth=10
...                   )
```

```
SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574664682274576)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')
    CategoricalInputs('stories','garagepl','bedrooms','bathrms')
) as sqlmr
```

```
>>> result
##### STDOUT Output #####

message
0      Parameters:
1      \tNumber of total trees (all subtrees): 20
2      \tMaxDepth : 10
3      \tColumnSubSampling Features: 4
4      \tNumber of boosted trees : 2
5  XGBoost model created in table specified in Ou...
6      \tVariance : 0.0
7      \tMinNodeSize : 1
8      \tNumber of boosting iterations : 10
9      \tRegularization : 1.0

##### model_table Output #####

tree_id iter class_num
tree
0      1 7 0 region_prediction
{"sum_":3.8962158200000037,"sumSq_":157.130603... {"66":-0.0482643,"68":-0.01546383,"69":0.00128...
1      0 4 0 {"68":-0.062050756,"71":0.029349836,"12":-0.06...
{"sum_":1.9165063099999996,"sumSq_":164.8552544... {"68":-0.062050756,"71":0.029349836,"12":-0.06...
2      0 7 0 {"sum_":3.3884664899999994,"sumSq_":152.650843... {"68":-0.05208172,"71":0.026769033,"12":-0.052...
3      1 6 0
```

```
{ "sum_":3.451967329999996,"sumSq_":160.9293557... {"66":-0.05077838,"68":-0.016889283,"69":0.001...
4      0      9      0
{"sum_":4.102372060000006,"sumSq_":147.3163931... {"12":-0.047053833,"140":-0.009371045,"141":-0...
5      1      3      0
{"sum_":1.7828434100000041,"sumSq_":177.218600... {"66":-0.060207672,"68":-0.02213716,"69":0.001...
6      1      9      0
{"sum_":4.6274621499999995,"sumSq_":151.195696... {"66":-0.043921467,"68":-0.012991273,"69":0.00...
7      0      6      0
{"sum_":2.9445456999999946,"sumSq_":156.045832... {"68":-0.055027403,"71":0.027585244,"12":-0.05...
8      0      3      0
{"sum_":1.3424553600000002,"sumSq_":170.569046... {"66":-0.06626853,"68":-0.016341366,"69":-0.00...
9      1      10     0
{"sum_":4.8805180300000055,"sumSq_":148.854092... {"72":-0.011381564,"73":0.021376234,"74":0.034...
```

Example 4: Explicitly specify all columns to be classified as categorical

This example explicitly specifies all columns to be classified as categorical Using dot (.) notation in 'as_categorical()'

```
formula ="homestyle ~ {}".format(as_categorical("."))
```

In this example, dependent variables are classified as:

- Numeric columns: None
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom', 'lotsize', 'price', 'stories', 'garagepl', 'bedrooms', 'bathrms', 'sn'

```
>>> formula ="homestyle ~ {}".format(as_categorical("."))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
...                   reg_lambda=1.0,
...                   shrinkage_factor=0.1,
...                   iter_num=10,
...                   min_node_size=1,
...                   max_depth=10
...                   )
```

```
SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574664350539035)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')
```

```
CategoricalInputs('fullbase','bathrms','bedrooms','lotsize','stories','garagepl',
'gashw','sn','recroom','prefarea','driveway','airco','price')
) as sqlmr
```

```
>>> result
##### STDOUT Output #####

message
Parameters:
0      \tNumber of total trees (all subtrees): 20
1      \tMaxDepth : 10
2      \tMinNodeSize : 1
3      \tColumnSubSampling Features: 13
4      XGBoost model created in table specified in Ou...
5      \tVariance : 0.0
6      \tNumber of boosted trees : 2
7      \tNumber of boosting iterations : 10
8      \tRegularization : 1.0
9

##### model_table Output #####

tree_id iter class_num region_prediction
tree
0      0      2      0      {"sum_-": -0.0676631500000023, "sumSq_-": 156.7910... {"1664": 0.06318546, "1792": 0.06280633, "256": -0.0...
1      0      10     0      {"sum_-": 3.3792588700000046, "sumSq_-": 53.6413343... {"780": 0.030176912, "781": 0.020259991, "32": -0.0...
2      1      2      0      {"sum_-": 1.2196361999999992, "sumSq_-": 161.179773... {"1536": 0.07016906, "1408": -0.10994268, "896": 0.0...
3      1      3      0      {"sum_-": 1.5425781199999997, "sumSq_-": 137.183413... {"1152": -0.10468066, "1280": -0.09020142, "1536": ...
4      0      1      0      {"sum_-": 1.1999999862677413E-6, "sumSq_-": 185.513... {"1792": 0.069990516, "1280": -0.16304731, "1664": ...
5      0      5      0      {"sum_-": 0.4210142700000011, "sumSq_-": 100.984244... {"776": 0.045522384, "777": 0.0248722, "780": 0.045...
6      0      9      0      {"sum_-": 2.9740613299999996, "sumSq_-": 60.4799287... {"22": -0.05125433, "32": -0.045820765, "33": -0.04...
7      1      1      0      {"sum_-": -2.3999999949619877E-6, "sumSq_-": 193.64... {"1792": 0.07149073, "1920": 0.07176345, "1536": 0.0...
8      1      7      0      {"sum_-": 1.6305844200000004, "sumSq_-": 78.7009234... {"780": 0.035701063, "781": 0.02491147, "1808": 0.0...
9      0      6      0      {"sum_-": 1.043163880000001, "sumSq_-": 88.31769190... {"1536": 0.034180116, "1537": 0.023620382, "769": 0.0...
```

Example 5: Treat only feature 'bedroom' as categorical column instead of numerical

This example treats feature 'bedroom' as categorical column instead of numerical, whereas other features are classified based on their datatypes.

```
formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories \ + garagepl + {}".format(as_categorical("bedrooms"))
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'bedrooms'

```
>>> formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical("bedrooms"))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                    id_column='sn',
...                    formula=formula,
```

```

...         num_boosted_trees=2,
...         loss_function='binomial',
...         prediction_type='classification',
...         reg_lambda=1.0,
...         shrinkage_factor=0.1,
...         iter_num=10,
...         min_node_size=1,
...         max_depth=10
...     )

```

```

SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574667836047592)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')
    NumericInputs('lotsize','price','stories','garagepl')
    CategoricalInputs('fullbase','driveway','prefarea','bedrooms')
) as sqlmr

```

```

>>> result
##### STDOUT Output #####

                                message
0                               Parameters:
1      \tNumber of total trees (all subtrees): 20
2      \tColumnSubSampling Features: 8
3      \tMaxDepth : 10
4      \tNumber of boosted trees : 2
5  XGBoost model created in table specified in Ou...
6      \tVariance : 0.0
7      \tPrediction Type : CLASSIFICATION
8      \tMinNodeSize : 1
9      \tNumber of boosting iterations : 10

##### model_table Output #####

tree_id iter  class_num
tree
0      0      1      0      region_prediction
1  {"sum_":1.1999999933731686E-6,"sumSq_":185.513... {"1280":-0.16304731,"1792":0.06969074,"128":-0...
2  {"sum_":3.8102337499999974,"sumSq_":62.4927878... {"768":0.02159745,"769":0.021676127,"1794":0.0...
3  {"sum_":1.6989790399999976,"sumSq_":91.0981258... {"1536":0.035986308,"1537":0.04640655,"770":0...
4  {"sum_":4.505127490000002,"sumSq_":55.56162297... {"1536":0.020756908,"1537":0.02097446,"769":0...
5  {"sum_":1.2315772999999983,"sumSq_":161.194586... {"1536":0.068237044,"1024":-0.10994268,"1792":...
6  {"sum_":1.3533744699999999,"sumSq_":103.2307089... {"768":0.02628861,"769":0.026451785,"770":0.02...

```

Example 6: Treat multiple features 'bedroom' and 'bathrms' as categorical column

This example treats multiple features 'bedroom' and 'bathrms' as categorical column instead of numerical, whereas other features are classified based on their datatypes.

```
formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical(['bedrooms', 'bathrms']))
```

In this example, dependent variables are classified as

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'bedrooms', 'bathrms'

```
>>> formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical(['bedrooms', 'bathrms']))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
...                   reg_lambda=1.0,
...                   shrinkage_factor=0.1,
...                   iter_num=10,
...                   min_node_size=1,
...                   max_depth=10
...                   )
```

```
SELECT * FROM XGBoost(
  ON "housing_train_binary" AS InputTable
  OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574670886269754)
  USING
  IdColumn('sn')
  NumBoostedTrees('2')
  LossFunction('binomial')
  PredictionType('classification')
  MaxDepth('10')
  ResponseColumn('homestyle')
  NumericInputs('lotsize','price','stories','garagepl')
  CategoricalInputs('fullbase','driveway','prefarea','bedrooms','bathrms')
) as sqlmr
```

```
>>> result
##### STDOUT Output #####

0                                     message
1                                     Parameters:
2         \tColumnSubSampling Features: 9
3         \tNumber of total trees (all subtrees): 20
4         \tMaxDepth : 10
5         \tNumber of boosted trees : 2
6 XGBoost model created in table specified in Ou...
7         \tVariance : 0.0
8         \tPrediction Type : CLASSIFICATION
9         \tMinNodeSize : 1
          \tNumber of boosting iterations : 10

##### model_table Output #####

tree_id iter  class_num
tree
0      1      2      0
{"sum_":1.231577299999995,"sumSq_":161.1945869... {"1536":0.068237044,"1024":-0.10994268,"1792":...
1      1     10      0
{"sum_":-1.0933073900000003,"sumSq_":57.153931... {"1536":0.04046509,"1537":0.041388433,"1538":0...
2      2      0
{"sum_":-0.11185043000000783,"sumSq_":156.7384... {"768":0.0422824,"769":0.030718312,"773":0.030...
3      0      3      0
{"sum_":-0.9086513100000011,"sumSq_":137.02733... {"774":0.02954042,"775":0.029739518,"8":-0.103...
4      1      1      0
{"sum_":-2.3999999949619877E-6,"sumSq_":193.64... {"1536":0.07176345,"1792":0.07149073,"1920":0...
5      1      5      0
{"sum_":-0.015070190000000205,"sumSq_":103.562... {"1536":0.05872538,"769":0.027473932,"1537":0...
6      1      9      0
{"sum_":-1.1304371800000002,"sumSq_":64.182343... {"1536":0.041750137,"1537":0.042725187,"258":-...
7      0      1      0
{"sum_":1.1999999862677413E-6,"sumSq_":185.513... {"1280":-0.16304731,"1792":0.06969074,"128":-0...
8      0      7      0
{"sum_":2.3138567199999995,"sumSq_":80.4486193... {"1024":-0.088556945,"768":0.024777533,"769":0...
9      1      6      0
{"sum_":-0.28911455999999935,"sumSq_":91.23620... {"1536":0.051188946,"1537":0.025760211,"769":0...
```

Example 7: Treat some features based on their datatypes, remaining be categorized as categorical

This example considers some features to be classified based on their types and categorize the remaining features as categorical. Use dot (.) notation in as 'as_categorical()'.

```
formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical("."))
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom', 'bedrooms', 'bathrms', 'sn'

```
>>> formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories garagepl + {}".format(as_categorical("."))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
```

```

...         reg_lambda=1.0,
...         shrinkage_factor=0.1,
...         iter_num=10,
...         min_node_size=1,
...         max_depth=10
...     )

```

```

SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574666028883640)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')
    NumericInputs('lotsize','price','stories','garagepl')

```

```

CategoricalInputs('fullbase','driveway','prefarea','sn','recroom','bathrms','bedr
ooms','airco','gashw')
) as sqlmr

```

```

>>> result
##### STDOUT Output #####

0                                     message
1                                     Parameters:
2         \tNumber of total trees (all subtrees): 20
3                                     \tMaxDepth : 10
4                                     \tMinNodeSize : 1
5         \tColumnSubSampling Features: 13
6 XGBoost model created in table specified in Ou...
7                                     \tVariance : 0.0
8         \tNumber of boosted trees : 2
9         \tNumber of boosting iterations : 10
                                     \tRegularization : 1.0

##### model_table Output #####

  tree_id iter  class_num
tree
0         1         1         0         region_prediction
{"sum_-": -2.399999989632917E-6, "sumSq_-": 193.644... {"1792": 0.07149073, "1536": 0.07176345, "1664": 0.0...
1         1         9         0         {"sum_-": 2.8963861200000007, "sumSq_-": 61.3832401... {"288": -0.03885908, "32": -0.04048751, "33": -0.05...
2         0         1         0         {"sum_-": 1.1999999933731686E-6, "sumSq_-": 185.513... {"1280": -0.16304731, "1792": 0.069990516, "128": -0.0...
3         0         4         0         {"sum_-": -0.21222700000000022, "sumSq_-": 116.7474... {"768": 0.037519634, "769": 0.026459606, "773": 0.0...
4         1         2         0         {"sum_-": 1.2196361999999998, "sumSq_-": 161.179773... {"1536": 0.07016906, "1408": -0.10994268, "1537": 0.0...
5         1         6         0         {"sum_-": 1.5360788999999984, "sumSq_-": 89.1511348... {"1536": 0.037206504, "1537": 0.02619288, "769": 0.0...
6         1         10        0         {"sum_-": 3.20626700000000013, "sumSq_-": 54.4304510... {"16": -0.038871493, "280": -0.037311606, "281": -0.0...
7         0         2         0         {"sum_-": -0.07118335000000012, "sumSq_-": 156.78730... {"1664": 0.06353445, "1792": 0.06280633, "256": -0.0...
8         0         8         0         {"sum_-": 2.44500522, "sumSq_-": 69.26084179328336, ... {"800": 0.032711692, "32": -0.04807788, "33": -0.04...
9         1         5         0         {"sum_-": 1.2610935399999945, "sumSq_-": 101.986445... {"1792": 0.050715607, "1793": 0.029154127, "16": -0.0...

```


Example 8: Treat some features based on their datatypes, remaining be categorized as categorical

This example considers some features to be classified based on their types and categorize the remaining features as categorical. Use dot (.) notation in as 'as_categorical()'.

```
formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical(["bedrooms", "."]))
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'gashw', 'airco', 'recroom', 'bedrooms', 'bathrms', 'sn'

```
>>> formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea +
stories + garagepl + {}".format(as_categorical(["bedrooms", "."]))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
...                   num_boosted_trees=2,
...                   loss_function='binomial',
...                   prediction_type='classification',
...                   reg_lambda=1.0,
...                   shrinkage_factor=0.1,
...                   iter_num=10,
...                   min_node_size=1,
...                   max_depth=10
...                   )
```

```
SELECT * FROM XGBoost(
  ON "housing_train_binary" AS InputTable
  OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574666256087363)
  USING
  IdColumn('sn')
  NumBoostedTrees('2')
  LossFunction('binomial')
  PredictionType('classification')
  MaxDepth('10')
  ResponseColumn('homestyle')
  NumericInputs('lotsize','price','stories','garagepl')
```

```
CategoricalInputs('fullbase','driveway','prefarea','sn','recroom','bathrms','bedr
```

```
ooms','airco','gashw')
) as sqlmr
```

```
>>> result
##### STDOUT Output #####

                                message
0                                Parameters:
1      \tNumber of total trees (all subtrees): 20
2                                \tMaxDepth : 10
3                                \tMinNodeSize : 1
4      \tColumnSubSampling Features: 13
5  XGBoost model created in table specified in Ou...
6                                \tVariance : 0.0
7                                \tNumber of boosted trees : 2
8      \tNumber of boosting iterations : 10
9                                \tRegularization : 1.0

##### model_table Output #####

  tree_id  iter  class_num          region_prediction
tree
0         1         2         0  {"sum_":1.2196361999999999,"sumSq_":161.1797732... {"1536":0.07016906,"1408":-0.10994268,"385":0...
1         1         10        0  {"sum_":3.2062669999999997,"sumSq_":54.43045103... {"16":-0.038871493,"280":-0.037311606,"281":-0...
2         0         2         0  {"sum_":-0.071183349999999776,"sumSq_":156.7873... {"1792":0.06280633,"1664":0.06353445,"256":-0...
3         0         3         0  {"sum_":-1.38502758999999875,"sumSq_":136.45000... {"1536":0.060084146,"769":0.02824491,"1537":0...
4         1         1         0  {"sum_":-2.39999999949619877E-6,"sumSq_":193.64... {"1536":0.07176345,"1792":0.07149073,"1920":0...
5         1         5         0  {"sum_":1.26109353999999967,"sumSq_":101.986445... {"1792":0.050715607,"1793":0.029154127,"16":-0...
6         1         9         0  {"sum_":2.8963861199999998,"sumSq_":61.38324016... {"32":-0.04048751,"288":-0.03885908,"33":-0.05...
7         0         1         0  {"sum_":1.1999999862677413E-6,"sumSq_":185.513... {"1792":0.069990516,"1280":-0.16304731,"768":0...
8         0         7         0  {"sum_":1.9211457200000006,"sumSq_":78.3929691... {"777":0.022986688,"11":-0.07360158,"1552":0.0...
9         1         6         0  {"sum_":1.5360789000000006,"sumSq_":89.1511348... {"1536":0.037206504,"769":0.02619288,"1537":0...
```

Example 9: Treat some features in both regular usage and specified in 'as_categorical()'

This example shows the scenario when columns specified in formula (regular usage) and columns specified in 'as_categorical()' have some intersecting features.

```
formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea
+ stories + garagepl + {}".format(as_categorical(['stories', 'garagepl',
'bedrooms', 'bathrms']))
```

In this example, dependent variables are classified as:

- Numeric columns: 'lotsize', 'price', 'stories', 'garagepl'
- Categorical Columns: 'fullbase', 'driveway', 'prefarea', 'stories', 'garagepl', 'bedrooms', 'bathrms'

```
>>> formula = "homestyle ~ lotsize + price + fullbase + driveway + prefarea
+ stories + garagepl + {}".format(as_categorical(['stories', 'garagepl',
'bedrooms', 'bathrms']))
```

```
>>> result = XGBoost(data=housing_train_binary,
...                   id_column='sn',
...                   formula=formula,
```

```

...         num_boosted_trees=2,
...         loss_function='binomial',
...         prediction_type='classification',
...         reg_lambda=1.0,
...         shrinkage_factor=0.1,
...         iter_num=10,
...         min_node_size=1,
...         max_depth=10
...     )

```

```

SELECT * FROM XGBoost(
    ON "housing_train_binary" AS InputTable
    OUT TABLE OutputTable(ALICE.ml__td_xgboost0_1574665459476007)
    USING
    IdColumn('sn')
    NumBoostedTrees('2')
    LossFunction('binomial')
    PredictionType('classification')
    MaxDepth('10')
    ResponseColumn('homestyle')
    NumericInputs('lotsize','price','stories','garagepl')

```

```

CategoricalInputs('fullbase','driveway','prefarea','stories','garagepl','bedrooms',
    'bathrms')
) as sqlmr

```

```

>>> result
##### STDOUT Output #####

                                message
0                                Parameters:
1      \tNumber of total trees (all subtrees): 20
2                                \tMaxDepth : 10
3                                \tMinNodeSize : 1
4                                \tColumnSubSampling Features: 11
5  XGBoost model created in table specified in Ou...
6                                \tVariance : 0.0
7                                \tNumber of boosted trees : 2
8                                \tNumber of boosting iterations : 10
9                                \tRegularization : 1.0

##### model_table Output #####

tree_id  iter  class_num          region_prediction
tree
0         0         1              {"sum_-":1.1999999862677413E-6,"sumSq_-":185.513... {"1280":-0.16304731,"1792":0.06969074,"128":-0...
1         0         9              {"sum_-":2.6447140800000013,"sumSq_-":64.2522225... {"776":0.024007909,"777":0.024123752,"1548":0...
2         1         1              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
3         1         4              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
4         0         2              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
5         0         6              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
6         0         10             {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
7         1         2              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
8         1         8              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...
9         1         8              {"sum_-":1.1443255100000025,"sumSq_-":120.172518... {"384":0.04151426,"385":0.02985721,"388":0.030...

```

```
9
0
5
0
{"sum_":1.4472957900000003,"sumSq_":103.417631... {"1808":0.026224444,"16":-0.068559565,"784":0...
```

column_casesensitive_handler

The **column_casesensitive_handler** is a boolean configuration property that controls whether to treat the column names passed to analytic function arguments as 'Case Sensitive' or 'Case Insensitive'.

- When the property is set to False (default), column names passed to analytic function wrappers are treated as 'Case Insensitive'.
- When the property is set to True, column names are treated as 'Case Sensitive'.

When using teradataml with Vantage 1.1 or later, you must set the value of this property match the **Column Name Handling** property of the QueryGrid ML Engine connector. When QueryGrid ML Engine connector property **Column Name Handling** is set to CASE-SENSITIVE, you must set the **column_casesensitive_handler** to True, otherwise False.

Note:

Set the value of the **column_casesensitive_handler** property to match the **Column Name Handling** property of the QueryGrid ML Engine connector. This makes sure that column case sensitivity errors are not observed while executing the analytic functions.

For detailed information about QueryGrid ML Engine connector property, see the section "ML Engine Connector Properties" in *Teradata Vantage™ User Guide*, B700-4002.

When using teradataml with Vantage 1.0 MU 2 or previous versions, set the **column_casesensitive_handler** property to True.

Example Setup

```
>>> from teradataml.analytics.mle import Antiselect
```

```
>>> # Load the example dataset
... load_example_data("antiselect", "antiselect_input_mixed_case")
```

```
>>> # Create required DataFrame on a table with column names with mixed case.
... antiselect_input_mixed_case = DataFrame.from_table("antiselect_input_mixed_case")
>>> antiselect_input_mixed_case
```

| region | rowids | OrderId | ORDERDATE | PRIORITY | quantity | sales | Discount | shipmode | custname | province |
|---------|--------|----------------|-----------|-----------------|----------|------------|----------|----------------|--------------------|----------|
| | | custsegment | | prodcat | | | | | | |
| 86 | | 515 | 10/08/28 | not specified | 21 | 146.6900 | 0.05 | regular air | carlos soltero | nunavut |
| nunavut | | consumer | | furniture | | | | | | |
| 50 | | 293 | 12/10/01 | high | 27 | 244.5700 | 0.01 | regular air | barry french | nunavut |
| nunavut | | consumer | | office supplies | | | | | | |
| 49 | | 293 | 12/10/01 | high | 49 | 10123.0200 | 0.07 | delivery truck | barry french | nunavut |
| nunavut | | consumer | | office supplies | | | | | | |
| 1 | | 3 | 10/10/13 | low | 6 | 261.5400 | 0.04 | regular air | muhammed macintyre | nunavut |
| nunavut | | small business | | office supplies | | | | | | |
| 80 | | 483 | 11/07/10 | high | 30 | 4965.7595 | 0.08 | regular air | clay rozendal | nunavut |
| nunavut | | corporate | | technology | | | | | | |
| 85 | | 515 | 10/08/28 | not specified | 19 | 394.2700 | 0.08 | regular air | carlos soltero | nunavut |
| nunavut | | consumer | | office supplies | | | | | | |
| 97 | | 613 | 11/06/17 | high | 12 | 93.5400 | 0.03 | regular air | carl jackson | nunavut |
| nunavut | | corporate | | office supplies | | | | | | |

Example 1: When QueryGrid ML Engine Connector Property Column Name Handling is set to CASE-SENSITIVE

If `column_casesensitive_handler` configuration option is set to False (default), error is raised for cases where mixed case columns are present and QueryGrid ML Engine Connector property values does not match the teradataml configuration property.

```
>>> # Set the teradataml Configuration property
'column_casesensitive_handler' False
>>> configure.column_casesensitive_handler = False

>>> # Check the configuration property value for teradataml
>>> print("configure.column_casesensitive_handler is set
to: {}".format(str(configure.column_casesensitive_handler)))
configure.column_casesensitive_handler is set to: False

>>> # Execute the Antiselect function.
>>> result = Antiselect(data=antiselect_input_mixed_case,
...
exclude=['rowids', 'ORDERDATE', 'Discount', 'province', 'custsegment']
...
)
Traceback (most recent call last):
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/
analytics/mle/Antiselect.py", line 223, in __execute
    UtilFuncs._create_view(sqlmr_stdout_temp_tablename, self.sqlmr_query)
.
.
.
terdatasql.OperationalError: [Version 16.20.0.39] [Session 10534] [Teradata
Database] [Error 9134] QGIEC: 60104: (afb007ce-ed78-4d17-995b-000000002c85)
tdap1110t1 : FFMeta : ANTISELECT: The column ORDERDATE in Exclude cannot be found.
()
.
.
.
at runtime.goexit asm_amd64.s:2361'
```

Set the `column_casesensitive_handler` configuration option to True.

```
>>> # Set the teradataml Configuration property
'column_casesensitive_handler' True
>>> configure.column_casesensitive_handler = True

>>> # Check the configuration property value for teradataml
>>> print("configure.column_casesensitive_handler is set
```

```
to: {}".format(str(configuration.column_casesensitive_handler)))
configuration.column_casesensitive_handler is set to: True
```

```
>>> # Execute the Antiselect function.
>>> result = Antiselect(data=antiselect_input_mixed_case,
...                      exclude=['rowids', 'ORDERDATE', 'Discount', 'province', 'custsegment'])
...
>>>
>>> result
##### STDOUT Output #####

  OrderId    PRIORITY quantity    sales    shipmode    custname    region    prodcat
0      515  not specified      21    146.6900    regular air    carlos soltero    nunavut    furniture
1      293         high      27    244.5700    regular air    barry french    nunavut    office supplies
2      293         high      49   10123.0200    delivery truck    barry french    nunavut    office supplies
3         3         low       6     261.5400    regular air    muhammed macintyre    nunavut    office supplies
4      483         high      30   4965.7595    regular air    clay rozendal    nunavut    technology
5      515  not specified      19    394.2700    regular air    carlos soltero    nunavut    office supplies
6      613         high      12     93.5400    regular air    carl jackson    nunavut    office supplies
```

Example 2: When QueryGrid ML Engine Connector Property Column Name Handling is set to CASE-INSENSITIVE

The `column_casesensitive_handler` configuration option is set to False (Default). teradataml configuration property matches with the QueryGrid ML Engine Connector property value.

```
>>> from teradataml.analytics.mle import Antiselect

>>> # Set the teradataml Configuration property
'column_casesensitive_handler' False
>>> configuration.column_casesensitive_handler = False

>>> # Check the configuration property value for teradataml
>>> print("configuration.column_casesensitive_handler is set to: {}".format(str(configuration.column_casesensitive_handler)))
configuration.column_casesensitive_handler is set to: False
>>>
>>> result = Antiselect(data=antiselect_input_mixed_case,
...                      exclude=['rowids', 'ORDERDATE', 'Discount', 'province', 'custsegment'])
...
>>>
>>> result
##### STDOUT Output #####

  orderid    priority quantity    sales    shipmode    custname    region    prodcat
0      613         high      12     93.5400    regular air    carl jackson    nunavut    office supplies
1      515  not specified      21    146.6900    regular air    carlos soltero    nunavut    furniture
2      293         high      49   10123.0200    delivery truck    barry french    nunavut    office supplies
3         3         low       6     261.5400    regular air    muhammed macintyre    nunavut    office supplies
4      515  not specified      19    394.2700    regular air    carlos soltero    nunavut    office supplies
5      293         high      27    244.5700    regular air    barry french    nunavut    office supplies
6      483         high      30   4965.7595    regular air    clay rozendal    nunavut    technology
```

If the `column_casesensitive_handler` configuration option is set to True, error is raised for cases where mixed case columns are present and QueryGrid ML Engine Connector property values does not match the teradataml configuration property.

```
>>> from teradataml.analytics.mle import Antiselect

>>> # Set the teradataml Configuration property
'column_casesensitive_handler' True
>>> configuration.column_casesensitive_handler = True

>>> # Check the configuration property value for teradataml
>>> print("configuration.column_casesensitive_handler is set to:
```

```

{}".format(str(configure.column_casesensitive_handler)))
configure.column_casesensitive_handler is set to: True
>>>
>>> # Execute the Antiselect function
>>> result = Antiselect(data=antiselect_input_mixed_case,
...
exclude=['rowids', 'ORDERDATE', 'Discount', 'province', 'custsegment']
...
)
Traceback (most recent call last):
  File "/Users/pp186043/Github_Repos/teradataml_repo/pyTeradata/teradataml/
analytics/mle/Antiselect.py", line 223, in __execute
    UtilFuncs._create_view(sqlmr_stdout_temp_tablename, self.sqlmr_query)
.
.
.
teradata.sql.OperationalError: [Version 16.20.0.39] [Session 10559] [Teradata
Database] [Error 9134] QGIEC: 60104: (afb007ce-ed78-4d17-995b-000000002c9f)
tdap1110t1 : FFMeta : ANTISELECT: The column "ORDERDATE" in Exclude cannot be
found. ()
  at github.td.teradata.com/gosql/gosqldriver.git/teradata.sql.
(*teradataConnection).formatDatabaseError TeradataConnection.go:1057
.
.
.
at runtime.goexit asm_amd64.s:2361'

```

Using the Naïve Bayes Model with Teradata Package for Python

This section uses the "iris" dataset for illustration. The dataset contains 3 classes of 50 instances each, where each class refers to a type of iris plant.

In this example, you separate the "iris" dataset into training dataset and test dataset, then build a Naïve Bayes Classifier model based on the training dataset and apply the model to the test dataset to evaluate the performance of the model.

It is assumed that the "iris" dataset is already in the database table "nb_input_iris".

1. Import the required modules.

```

from teradataml.analytics.mle.NaiveBayes import NaiveBayes

from teradataml.analytics.sqle.NaiveBayesPredict import NaiveBayesPredict

from teradataml.dataframe.dataframe import DataFrame

```

2. Create a teradataml DataFrame "nb_iris_input_train" for the training dataset from the "nb_input_iris" table.

```
nb_iris_input_train = DataFrame.from_query("SELECT * FROM nb_input_iris WHERE
id MOD 5 <> 0")
```

3. Train a new Naïve Bayes Classifier model based on the teradataml DataFrame "nb_iris_input_train" from the training dataset, using the NaiveBayes function from teradataml package.

```
# Run the train function
naivebayes_train = NaiveBayes(formula="species ~ petal_length + sepal_width +
petal_width + sepal_length", data=nb_iris_input_train)
```

Once the model is created, you can apply the model to the test dataset.

4. Create a teradataml DataFrame "nb_iris_input_test" for the test dataset from the "nb_input_iris" table.

```
nb_iris_input_test = DataFrame.from_query("SELECT * FROM nb_input_iris WHERE
id MOD 5 = 0")
```

5. Predict the flower type by applying the Naïve Bayes model to the teradataml DataFrame "nb_iris_input_test" from the test dataset, using the NaiveBayesPredict.

```
# Generate prediction using output of train function
naivebayes_predict_result = NaiveBayesPredict(newdata=nb_iris_input_test,
                                              modeldata = naivebayes_train,
                                              id_col = "id",
                                              responses
= ["virginica","setosa","versicolor"]
)
```

6. Inspect the results.

```
naivebayes_predict_result
```

Using the Decision Forest Model with Teradata Package for Python

This section uses the housing data for illustration. The dataset contains 492 samples, each with 12 features describing a home style, and a 13th column indicates the home style.

In this example, you build a Decision Forest model based on the training dataset and apply the model to the test dataset to evaluate the performance of the model.

1. Import the required modules.

```
from teradataml.analytics.mle.DecisionForest import DecisionForest
```



```

from teradataml.analytics.sqlc.DecisionForestPredict
import DecisionForestPredict

from teradataml.dataframe.dataframe import DataFrame

from teradataml.data.load_example_data import load_example_data

```

2. If the input tables `housing_train` and `housing_test` do not already exist, create them and load the datasets into them.

```
load_example_data("decisionforestpredict", ["housing_train", "housing_test"])
```

3. Create a `teradataml` `DataFrame` `"housing_train"` consisting the tokens from the training dataset.

```
housing_train = DataFrame.from_table("housing_train")
```

4. Train a new Decision Forest model based on the `teradataml` `DataFrame` `"housing_train"` from the training dataset, using the `DecisionForest` function from `teradataml` package.

```

formula = "homestyle ~ driveway + recroom + fullbase + gashw + airco +
prefarea + price + lotsize + bedrooms + bathrms + stories + garagepl"

rft_model = DecisionForest(data=housing_train,
                           formula = formula,
                           tree_type="classification",
                           ntree=50,
                           tree_size=100,
                           nodesize=1,
                           variance=0.0,
                           max_depth=12,
                           maxnum_categorical=20,
                           mtry=3,
                           mtry_seed=100,
                           seed=100
                           )

```

Once the model is created, you can apply the model to the test dataset.

5. Create a `teradataml` `DataFrame` `"housing_test"` with the tokens from the test dataset.

```
housing_test = DataFrame.from_table("housing_test")
```

6. Predict the home styles by applying the Decision Forest model to the `teradataml` `DataFrame` `"housing_test"` from the test dataset, using the `DecisionForestPredict`.

```

decision_forest_predict_out = DecisionForestPredict(object = rft_model,
                                                    newdata = housing_test,

```

```
id_column = "sn",
detailed = False,
terms = ["homestyle"]
)
```

7. Inspect the results.

```
decision_forest_predict_out
```

Using Text Analysis with Teradata Package for Python

This example investigates a log of vehicle complaints that have been categorized as crash-related or not crash-related. Use this log to build a Naïve Bayes Text Classifier model, and then apply the model to a new log data to predict if the complaint is associated with a crash.

It is assumed that the training and test datasets are already in the database.

The following table "complaints" contains the training dataset.

| doc_id | text_data | category |
|--------|--|----------|
| 1 | consumer was driving approximately 45 mph hit a deer with the front bumper and then ran into an embankment head-on passenger's side air bag did deploy hit windshield and deployed outward. driver's side airbag cover opened but did not inflate it was still folded causing injuries. | crash |
| 2 | when vehicle was involved in a crash totalling vehicle driver's side/ passenger's side air bags did not deploy. vehicle was making a left turn and was hit by a ford f350 traveling about 35 mph on the front passenger's side. driver hit his head-on the steering wheel. hurt his knee and received neck and back injuries. | crash |
| 3 | consumer has experienced following problems; 1.) both lower ball joints wear out excessively; 2.) head gasket leaks; and 3.) cruise control would shut itself off while driving without foot pressing on brake pedal. | no_crash |
| 4 | transfer case was repaired under recall. after the work was completed noise was heard intermittently. consumer took vehicle back to dealer. the dealer re-inspected vehicle and informed the owner that the driveshaft was hitting the transfer case. the manufacturer has been notified. | no_crash |
| 5 | transmission would start to slip when traveling just 10mph. the rpms would be over 3 thousand. had vehicle checked at dealership & was informed transmission was stuck & that it's a factory defect almost blew up. also speedometer does not keep accurate speeds. if speed is increased, it would fail to work. this was referred to mechanic by manufacturer. | no_crash |
| 6 | due to the defective ignition cable which burned the coil the vehicle stalled unexpectedly which could have resulted in a crash. also dealer replaced the r&r drive belts/speed control cable and performed vehicle tune up. please provide further information. | no_crash |
| 7 | when switch is turned on windshield wipers would not work properly. would have to jiggle switch & then wipers would move. wipers do turn off/on by themselves. recall 97v017000. | no_crash |

| doc_id | text_data | category |
|--------|---|----------|
| 8 | consumer was driving in a rain storm when the windshield wipers stopped this happened periodically. | no_crash |
| 9 | at 66900 miles transmission has malfunctioned and will not shift into first gear. repairs were made at owner's expense wants reimbursement. *ml | no_crash |
| 10 | when truck was sitting on an incline it rolled on its own. manufacturer was aware of the problem. problem has not been corrected. the truck is owned by walnut hill recker manufactured in 1998. | no_crash |
| 11 | car engine raced while slowing to park. car lurched forward and crashed into a fence and a building. car had been in shop approximately one week prior to incident for high idle condition. | crash |
| 12 | rear ended another vehicle at 65 to 70mph and neither driver's side or passenger's side airbags deployed. dealer has vehicle. | crash |
| 13 | while vehicle was parked for an hour a fire started on the left side of the engine compartment. owners son smelled smoke owner saw fire coming from around drivers side front wheel. referenced in ea02-025 | no_crash |
| 14 | after vehicle was repaired under recall 99v029000 ignition switch the airbag light stayed on . the dealer and the manufacturer has been notified. | no_crash |
| 15 | electrical control module is shortening out causing the vehicle to stall. engine will become totally inoperative. consumer had to change alternator/ battery and starter and module replaced 4 times but defect still occurring cannot determine what is causing the problem. | no_crash |
| 16 | at 68000 miles power steering broke off the housing pump causing total loss of power steering which also caused the vehicle to shut down. | no_crash |
| 17 | on two occasions dual airbags did not deploy. consumer rear-ended another vehicle at approximately 50 mph and at 80 mph hit a truck head-on upon impact air bags did not deploy. driver sustained injuries. dealer did not determine why air bags did not deploy. | crash |
| 18 | sunroof is leaking. | no_crash |
| 19 | motor and the frame separated from vehicle. manufacturer will be notified. | no_crash |
| 20 | rear front wheel bearing broke causing vehicle to pull to the left when slowing down. consumer had brake's replaced about four times and still dealer can't determine the problem. | no_crash |

The following table "nbayes_test" contains the test dataset.

| doc_id | text_data |
|--------|--|
| 1 | ELECTRICAL CONTROL MODULE IS SHORTENING OUT, CAUSING THE VEHICLE TO STALL. ENGINE WILL BECOME TOTALLY INOPERATIVE. CONSUMER HAD TO CHANGE ALTERNATOR/ BATTERY AND STARTER, AND MODULE REPLACED 4 TIMES, BUT DEFECT STILL OCCURRING CANNOT DETERMINE WHAT IS CAUSING THE PROBLEM. |

| doc_id | text_data |
|--------|--|
| 2 | ABS BRAKES FAIL TO OPERATE PROPERLY, AND AIR BAGS FAILED TO DEPLOY DURING A CRASH AT APPROX. 28 MPH IMPACT. MANUFACTURER NOTIFIED. |
| 3 | WHILE DRIVING AT 60 MPH GAS PEDAL GOT STUCK DUE TO THE RUBBER THAT IS AROUND THE GAS PEDAL. |
| 4 | THERE IS A KNOCKING NOISE COMING FROM THE CATALYTIC CONVERTER, AND THE VEHICLE IS STALLING. ALSO, HAS PROBLEM WITH THE STEERING. |
| 5 | CONSUMER WAS MAKING A TURN, DRIVING AT APPROX 5-10 MPH WHEN CONSUMER HIT ANOTHER VEHICLE. UPON IMPACT, DUAL AIRBAGS DID NOT DEPLOY. ALL DAMAGE WAS DONE FROM ENGINE TO TRANSMISSION, TO THE FRONT OF VEHICLE, AND THE VEHICLE CONSIDERED A TOTAL LOSS. |
| 6 | WHEEL BEARING AND HUBS CRACKED, CAUSING THE METAL TO GRIND WHEN MAKING A RIGHT TURN. ALSO WHEN APPLYING THE BRAKES, PEDAL GOES TO THE FLOOR, CAUSE UNKNOWN. WAS ADVISED BY MIDAS NOT TO DRIVE VEHICLE- WHEEL COULD COME OFF. |
| 7 | DRIVING ABOUT 5-10 MPH, THE VEHICLE HAD A LOW FRONTAL IMPACT IN WHICH THE OTHER VEHICLE HAD NO DAMAGES. UPON IMPACT, DRIVER'S AND THE PASSENGER'S AIR BAGS DID NOT DEPLOY, RESULTING IN INJURIES. PLEASE PROVIDE FURTHER INFORMATION AND VIN#. |
| 8 | THE AIR BAG WARNING LIGHT HAS COME ON, INDICATING AIRBAGS ARE INOPERATIVE. THEY WERE FIXED ONE AT THE TIME, BUT PROBLEM HAS REOCCURRED. |
| 9 | CONSUMER WAS DRIVING WEST WHEN THE OTHER CAR WAS GOING EAST. THE OTHER CAR TURNED IN FRONT OF CONSUMER'S VEHICLE, CONSUMER HIT OTHER VEHICLE AND STARTED TO SPIN AROUND, COULDN'T STOP, RESULTING IN A CRASH. UPON IMPACT, AIRBAGS DIDN'T DEPLOY. |
| 10 | WHILE DRIVING ABOUT 65 MPH AND THE TRANSMISSION MADE A STRANGE NOISE, AND THE LEFT FRONT AXLE LOCKED UP. THE DEALER HAS REPAIRED THE VEHICLE. |

This example shows the steps to build a Naïve Bayes Text Classifier model and then apply the model to the new log data.

1. Import the required modules and load the example datasets.

```
from teradataml.analytics.mle.NaiveBayesTextClassifier
import NaiveBayesTextClassifier

from teradataml.analytics.sqle.NaiveBayesTextClassifierPredict
import NaiveBayesTextClassifierPredict

from teradataml.analytics.mle.TextTokenizer import TextTokenizer

from teradataml.dataframe.dataframe import DataFrame
```

```
from teradataml import load_example_data

load_example_data("TextTokenizer","complaints")
```

2. Create a teradataml DataFrame from the training dataset.

```
complaints = DataFrame.from_table("complaints")
```

- a. Create tokens from the training dataset.

```
text_tokenizer_out = TextTokenizer(data=complaints,
                                   text_column='text_data',
                                   output_byword = True,
                                   accumulate=['doc_id', 'category'])
```

- b. Create a teradataml DataFrame "tddf_nbayes_tokens" consisting of the tokens from the training dataset, ignoring the case (*lower()*).

```
tddf_nbayes_tokens = text_tokenizer_out.result.assign(drop_columns = True,
                                                    doc_id
= text_tokenizer_out.result.doc_id,
                                                    token
= text_tokenizer_out.result.token.str.lower(),
                                                    category
= text_tokenizer_out.result.category)
```

3. Train a new Naïve Bayes Text Classifier based on the teradataml DataFrame from the training dataset, using the NaiveBayesTextClassifier function from teradataml package.

```
nb_textclassifier_model = NaiveBayesTextClassifier(data = tddf_nbayes_tokens,
                                                  data_partition_column
= "category",
                                                  token_column = "token",
                                                  doc_category_column
= "category")
```

Next, apply the model to the test data.

4. Create a teradataml DataFrame from the test dataset.

```
nbayes_test = DataFrame.from_table("nbayes_test")
```

- a. Create tokens from the test dataset.

```
text_tokenizer_out_2 = TextTokenizer(data=nbayes_test,
                                     text_column='text_data',
```

```
output_byword = True,
accumulate='doc_id')
```

- b. Create a teradataml DataFrame "tddf_nbayes_tokens_test" with the tokens from the test dataset, ignoring the case (*lower()*).

```
tddf_nbayes_tokens_test = text_tokenizer_out_2.result.assign(drop_columns
= True,
doc_id
= text_tokenizer_out_2.result.doc_id,
token
= text_tokenizer_out_2.result.token.str.lower() )
```

5. Predict the categories ('crash' or 'no crash') by applying the Naïve Bayes Text Classifier model to the teradataml DataFrame from the test dataset, using the NaiveBayesTextClassifierPredict function.

```
nb_textclassifier_pred = NaiveBayesTextClassifierPredict(newdata
= tddf_nbayes_tokens_test,
object
= nb_textclassifier_model,
newdata_partition_column = "doc_id",
input_token_column
= "token",
doc_id_columns
= ["doc_id"])
```

6. Inspect the results.

```
nb_textclassifier_pred
```

Using Sentiment Extraction with Teradata Package for Python

This example uses the sentiment extraction function `SentimentExtractor()` from the `teradataml` package to evaluate and classify a set of restaurant reviews.

1. Import the required modules.

```
from teradataml.analytics.mle.SentimentExtractor import SentimentExtractor
from teradataml.dataframe.dataframe import DataFrame
from teradataml.data.load_example_data import load_example_data
```

2. If the input table "restaurant_reviews" does not already exist, create the table and load the datasets into the table.

```
load_example_data("sentimentextractor", "restaurant_reviews")
```

3. Create a teradataml DataFrame from the input data table "restaurant_reviews".

```
tddf_restaurant_reviews = DataFrame("restaurant_reviews")
```

Note:

This example uses the default values for many arguments of the `SentimentExtractor()` function.

One of those is the **object** argument which specifies the source used to assign sentiment values to words. The default value of the **object** argument is a built-in dictionary based on the WordNet lexical database.

4. Call the sentiment extraction function.

```
td_sentiment_extractor_out = SentimentExtractor(
    object = "dictionary",
    newdata = tddf_restaurant_reviews,
    level = "document",
    text_column = "review_text",
    accumulate = ["id"]
)
```

5. Inspect the result.

```
print(td_sentiment_extractor_out.result)
```

Clustering Using KMeans with Teradata Package for Python

This example uses the US Arrests data of 50 samples containing statistics for arrests made per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973, along with the percentage of the population living in urban areas.

Example here shows how to use KMeans clustering function.

1. Import the required modules.

```
from teradataml.analytics.mle.KMeans import KMeans
from teradataml.dataframe.dataframe import DataFrame
from teradataml.data.load_example_data import load_example_data
import matplotlib.pyplot as plt
```

2. If the input table "kmeans_us_arrests_data" does not already exist, create the table and load the datasets into the table.

```
load_example_data("kmeans", "kmeans_us_arrests_data")
```

3. Create a teradataml DataFrame from "kmeans_us_arrests_data".

```
## Creating TeradataML dataframes
df_train = DataFrame('kmeans_us_arrests_data')

# Print train data to see how the sample train data looks like
print("\nHead(10) of Train data:")
print(df_train.head(10))
```

4. In the training dataset, the features are 'sno', 'state', 'murder', 'assault', 'urban_pop' and 'rape'. And 'sno' to 'state' is a one-to-one mapping. For training, drop off the repeated feature 'state'.

```
# A dictionary to get 'sno' to 'state' mapping. Required for plotting.
df1 = df_train.select(['sno', 'state'])
sno_to_state = dict(df1.to_pandas()['state'])
print(sno_to_state)

# No need of 'state' columns, instead we have 'sno' column for the same
df_train = df_train.drop(['state'], axis=1)
colnames = ["sno", "murder", "assault", "urban_pop", "rape"]
```

5. Apply KMeans algorithm to generate two clusters and inspect the outputs.
 - a. Apply KMeans algorithm.

```
## KMmeans algorithm on data, with 2 centroids
kMeans_output = KMeans(data=df_train,
                        centers=2, data_sequence_column=['sno'])
```

```
# Print the KMeans results
print(kMeans_output)
```

- b. The 'cluster_centroids' output dataframe presents the centroid values of the two clusters, the withinss and the number of samples in each cluster out of all the samples.

```
# Features (4-dimensional) used to train KMeans algorithm.
features = 'murder assault urban_pop rape'.split()
```

```
# Feature Centroids for cluster 1
centroid_values = kMeans_output.clusters_centroids.to_pandas()['murder
assault urban_pop rape'][0].split()
```



```
print("4-dimensional Centroid of Cluster 1:")
print(dict(zip(features, centroid_values)))
```

```
# Feature Centroids for cluster 2
centroid_values = kMeans_output.clusters_centroids.to_pandas()['murder
assault urban_pop rape'][1].split()
print("4-dimensional Centroid of Cluster 2:")
print(dict(zip(features, centroid_values)))
```

Check the withinss to see if the withinss is the cluster sum of squares (from teradataml DataFrame output 'clusters_centroids').

```
# Withinss for cluster 1
print("Withinss for cluster 1: " +
      str(kMeans_output.clusters_centroids.to_pandas()['withinss'][0]))
```

```
# Withinss for cluster 2
print("Withinss for cluster 2: " +
      str(kMeans_output.clusters_centroids.to_pandas()['withinss'][1]))
```

- c. The 'clustered_output' dataframe presents the 'sno' representing state and the corresponding mapping number to cluster for all the samples.

```
kMeans_output.clustered_output.head(30)
```

- d. The 'output' dataframe presents overall summary of the KMeans output.

```
print(kMeans_output.output)
```

- 6. Plot clusters based on feature to analyze the clustering output.

```
## Inner join of clustered_output to actual dataset df_train We shall use the
data from df1 to plot.
df1 = df_train.join(kMeans_output.clustered_output, how='inner',
on=['sno'], lsuffix='t1',
                      rsuffix='t2')

print("\nInner join of clustered_output to actual dataset df_train:")
print(df1)
```

- a. Plot clusters based on the two features 'urban_pop' and 'murder'.

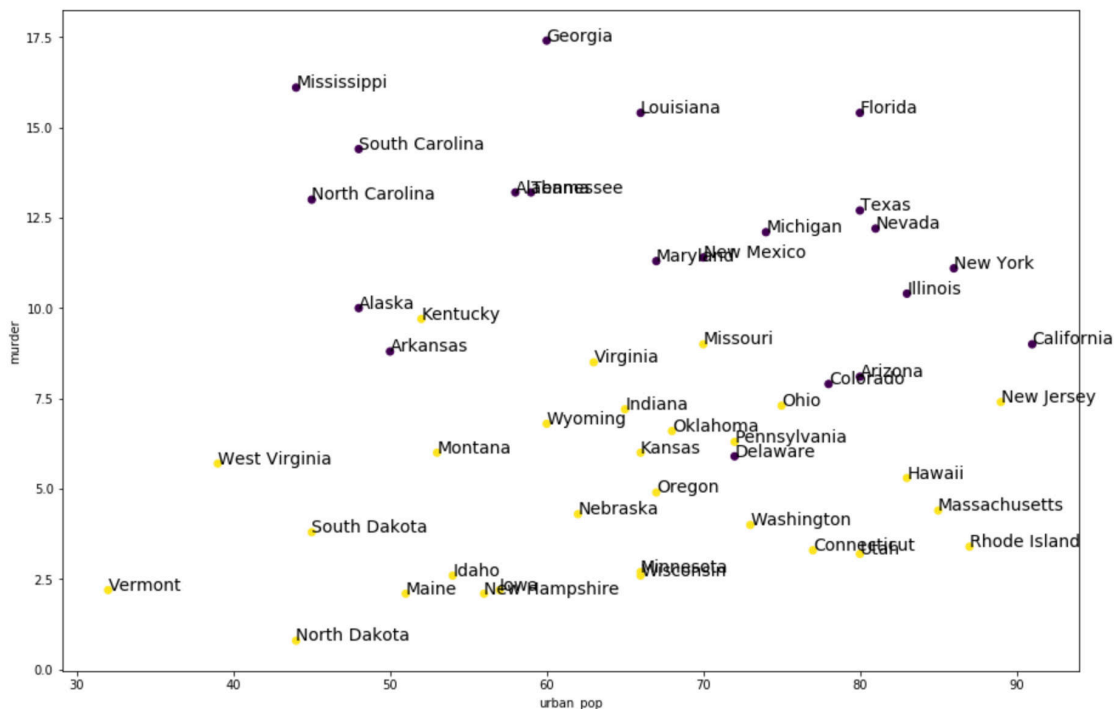
```
# Selecting only the necessary features for plot.
df3 = df1.select(['t1_sno', 'urban_pop', 'murder', 'clusterid'])
```

```
# Since there is no plotting possible for teradataml DataFrame, we are
# converting it to
# pandas dataframe and then to numpy_array 'numpy_df' to use matplotlib
# library of python.
pandas_df = df3.to_pandas()
numpy_df = pandas_df.values
```

```
# Setting figure display size.
plt.rcParams['figure.figsize'] = [15, 10]
```

```
# Coloring based on cluster_id.
plt.scatter(numpy_df[:,1], numpy_df[:,2], c=numpy_df[:,3])
for ind, value in enumerate(numpy_df[:, 0]):
    # sno_to_state is used here to get state names.
    plt.text(numpy_df[ind,1], numpy_df[ind,2],
sno_to_state[int(value)], fontsize=14)
plt.xlabel('urban_pop')
plt.ylabel('murder')
plt.show()
```

After running these commands, the following plot shows:



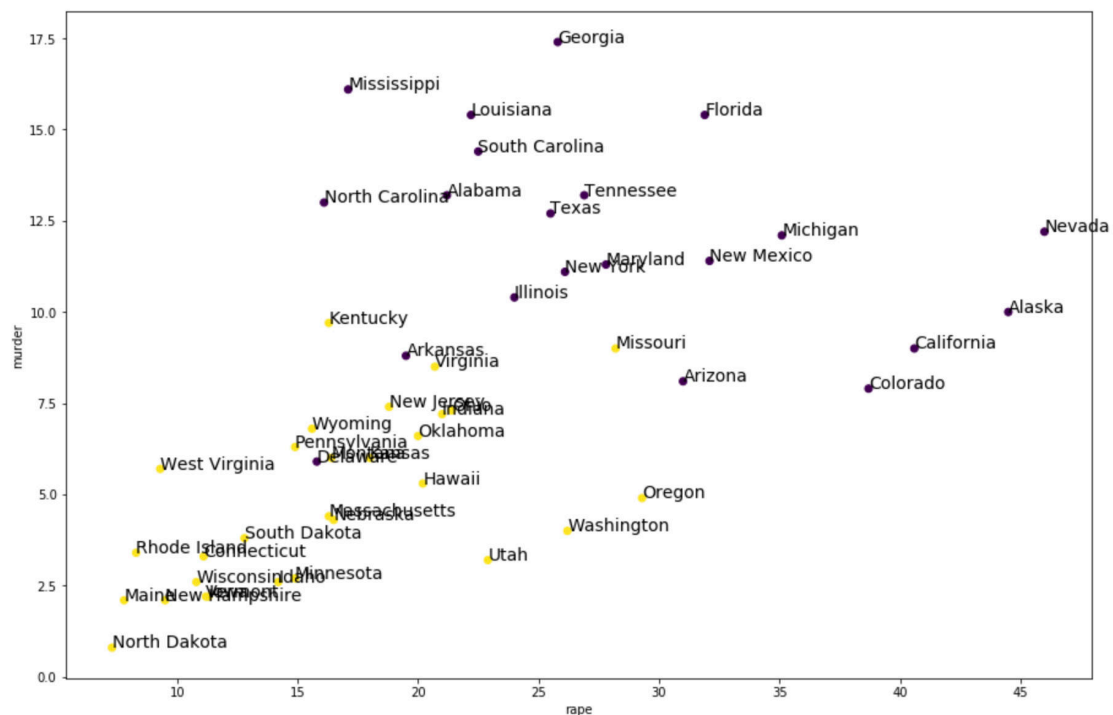
- b. Plot the clusters based on the two features 'rape' and 'murder'.

```
# Selecting only the necessary features for plot.
df3 = df1.select(['t1_sno', 'rape', 'murder', 'clusterid'])

# Since there is no plotting possible for teradataml DataFrame, we are
# converting it to
# pandas dataframe and then to numpy_array 'numpy_df' to use matplotlib
# library of python.
pandas_df = df3.to_pandas()
numpy_df = pandas_df.values

# Coloring based on cluster_id.
plt.scatter(numpy_df[:,1], numpy_df[:,2], c=numpy_df[:,3])
for ind, value in enumerate(numpy_df[:, 0]):
    # sno_to_state is used here to get state names.
    plt.text(numpy_df[ind,1], numpy_df[ind,2],
sno_to_state[int(value)], fontsize=14)
plt.xlabel('rape')
plt.ylabel('murder')
plt.show()
```

After running these command, the following plot shows:



Additional Information

Changes and Additions

The table lists changes and additions in this release of the *Teradata Package for Python User Guide*, B700-4006.

| Date | Release | Description |
|---------------|-------------|---|
| May 2023 | 17.20.00.03 | <ul style="list-style-type: none"> Added new arguments <i>lprefix</i> and <i>rprefix</i> and note to the DataFrame.join method Added deprecation note to the Model Cataloging section Removed the limitation "DataFrame Creation on Volatile Tables is not Supported" Removed PMMLPredict, ONNXPredict, H2OPredict from the import necessary modules step in the corresponding BYOM examples |
| March 2023 | 17.20.00.02 | <ul style="list-style-type: none"> Added new function <code>set_auth_token()</code> to Working with Open Analytics section Added new notes to Working with Open Analytics functions (<code>remove_env</code>, <code>remove_all_envs</code>, <code>install_lib</code>, <code>update_lib</code>, <code>uninstall_lib</code>) Added new DataFrame Manipulation function <code>drop_duplicate()</code> Added new DataFrame Data Rotation functions (<code>pivot()</code>, <code>unpivot()</code>) Added new <code>teradataml</code> option Updated use case for Using Decision Forest Model with <code>teradataml</code> Package Updated <code>apply</code> method Updated <code>print_options</code> function <code>suppress_vantage_runtime_warnings()</code> |
| January 2023 | 17.20.00.01 | <ul style="list-style-type: none"> Added input classes for UAF functions Added notes regarding dynamic imports for Analytics Database functions, UAF functions, NOS functions and BYOM functions in all related sections: <ul style="list-style-type: none"> New Considerations item New or updated Usage Notes in sections for these functions Updated arguments in <code>copy_to_sql</code>, <code>fastload</code>, <code>fastexport</code> and <code>read_csv</code> functions Add new section of Data Transfer using <code>teradataml</code> |
| November 2022 | 17.20.00.00 | <ul style="list-style-type: none"> To support Open Analytics Framework on VantageCloud Lake: <ul style="list-style-type: none"> Added new APIs to manage user environment, and manage files and libraries inside a user environment Add new <code>apply</code> class and supporting functions for <code>Apply</code> table operator Added new DataFrame.apply method manipulate DataFrame rows to leverage <code>APPLY</code> table operator Added new configuration options (<code>auth_token</code>, <code>ues_url</code>, <code>certificate_file</code>) and function (<code>set_config_params</code>) |

| Date | Release | Description |
|------|---------|--|
| | | <ul style="list-style-type: none"> • Added new section in BYOM and new use case in Using Analytic Functions for ONNXPredict • Added usage note regarding column specification for analytic functions in Analytics Database • Updated parameters for map_row() and map_partition() • Updated create_context() to include URL encoding of password while creating a context • Added new general function list_td_reserved_keywords() • Added new workflows for BYOM and OpenAF • Added new Limitation and Consideration about issues with VIEW creation, and solutions and examples |

Teradata Links

| Link | Description |
|---|--|
| https://docs.teradata.com/ | Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books. |
| https://support.teradata.com | Helpful resources in one place: <ul style="list-style-type: none"> • Support requests • Account management and software downloads • Knowledge base, community, and support policies • Product documentation • Learning resources, including Teradata University |
| https://www.teradata.com/University/Overview | Teradata education network |
| https://support.teradata.com/community | Link to Teradata community |