How to connect to Denodo from Python - a starter for Data Scientists

Revision 20230213

Denodo Inc.: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA www.denodo.com

1 INTRODUCTION

Python is one of the most widely used programming languages in the Data Science community for its flat learning curve and a very rich analytical stack, that spans from data preparation (e.g. pandas, numpy), through machine learning (e.g. scikit-learn) to data visualization (e.g. bokeh, seaborn, Apache Superset). Denodo can get its spot in the Data Scientist's workflow as a powerful data preparation and governance tool, that offers the most optimized and widest access both to internal and external data sources.

For this reason, in this article we will discuss the different options available for establishing the connection and importing data from Denodo to Python and how we can transform them into pandas dataframes, the data structure that represents the starting point of data exploration, analysis and visualization as well as training of machine learning algorithms.

As we are addressing pandas as a computing environment, we are limiting this analysis to single-machine environments, with data batches that must be loadable in memory.

2 CONNECTION METHODS

There are several Python libraries that we can use to establish the connection between Python and Denodo, we can see an overview of the main ones in the following table:

Library	Denodo Interface	Comment
Denodo Dialect for SQLAlchemy	ODBC/libpq	SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.
psycopg2	ODBC/libpq	psycopg2 is a popular adapter for PostgreSQL that can be also used for Denodo.
turbodbc	ODBC	turbodbc developers explicitly target Data Scientists. This library has built-in numpy and Apache Arrow optimizations
pyodbc	ODBC	The most popular and mature ODBC-based library to connect to relational databases.
jaydebeapi	JDBC	The main JDBC-based python library, it makes use of JPype to bridge Python and Java code

To all these libraries, Denodo is exposed as a standard relational database. All these libraries comply with the Python standard DB-API v2.0.

3 RECOMMENDATIONS

The easiest way to connect to the Denodo Platform from Python is to use the Denodo Dialect for SQLAlchemy or the psycopg2 driver. We recommend the use of the Denodo Dialect for SQLAlchemy for greater convenience, and due to the higher-level set of functionality it offers and the fact that it makes use of psycopg2 underneath. The SQLAlchemy option also makes it easy to work with pandas dataframes.

However, if you need to establish a connection to Denodo using Kerberos or OAuth2 authentication, this is not supported by psycopg2 so you will need to switch to one of the options that make use of the Denodo ODBC driver. In this case we recommend the use of the turbodbc library. This option might also provide a minor performance improvement in some scenarios and, if you intend to use pandas dataframes, using the `fetchallnumpy` option may improve performance a bit more.

The use of pyodbc is less recommended because performance is worse in most scenarios than that obtained using the Denodo Dialect for SQLAlchemy, psycopg2 directly, or turbodbc. Finally, jaydebeapi shows some performance issues in this Python - Denodo configuration that make its use not recommended for connecting to Denodo from Python unless the scenario requires it.

4 CONNECTION EXAMPLES

4.1 PREREQUISITES

If you are using the Denodo Dialect for SQLAlchemy or the psycopg2 adapter directly you don't need to install any Denodo drivers. You just need to install the necessary libraries through a package manager and import them from your python code.

If you are using pyodbc or turbodbc you must install the Denodo ODBC driver in the machine where your Python interpreter is installed in order to be able to establish the connection between Denodo and Python through ODBC.

- If your machine runs Windows, follow the instructions here.
- If your machine runs Linux or other Unix OS, follow the instructions here.

In case you use jaydebeapi you will need to place the Denodo JDBC driver in the machine where your Python interpreter is installed in order to be able to establish the connection between Denodo and Python through JDBC.

4.2 EXAMPLE CODE: DENODO DIALECT FOR SQLALCHEMY

In the following example we use Denodo Dialect for SQLAlchemy to fetch data from a Denodo virtual database. The main benefit of using Denodo Dialect for SQLAlchemy is that you don't need to install the Denodo ODBC driver in the target machine and you do not rely on an external DSN. Also, if you want to use SQLAlchemy-based tools like Apache Superset you need this dialect.

Note that the Denodo Dialect for SQLAlchemy will transparently install and make use of psycopg2 as a dependency.

```
##          Script          name:          sqlalchemy-denodo-connection.py

## Importing the main library used to connect to Denodo via sqlalchemy
import               sqlalchemy               as               dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from               socket               import               gethostname

##    Connection    parameters    to    the    Denodo    VDP    Server
denodoserver_name                    =                    "denodoserver"

##        Default        port        for        ODBC        connections
denodoserver_odbc_port                       =                       "9996"

denodoserver_database                =                "distributed_tpcds"
```

```
denodoserver_uid                          =                          "tpcds_usr"

denodoserver_pwd                          =                          "tpcds_usr"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                           =                           gethostname()

useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                Establishing                a                connection
cnxn_str            =            "denodo://%s:%s@%s:%s/%s"            %\
                (denodoserver_uid,  denodoserver_pwd,  denodoserver_name,
denodoserver_odbc_port, denodoserver_database)

engine=dbdriver.create_engine(cnxn_str)

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query = "select * from bv_date_dim"
result_set=engine.execute(query)

##  Finally  fetch  the  results.  `results`  is  a  list  of  lists.
##  If  you  don't  want  to  load  all  the  records  in  memory,  you  may
##     want     to     use     cur.fetchone()     or     cur.fetchmany()
results = result_set.fetchall()
```

4.3 EXAMPLE CODE: PSYCOPG2

In the following example we use psycopg2 (directly) to fetch data from a Denodo virtual database. As with the SQLAlchemy dialect, a benefit of using psycopg2 directly is that you don’t need to install the Denodo ODBC driver in the target machine and you do not rely on an external DSN. But in this case you will not benefit from the features SQLAlchemy offers.

```
##             Script             name:             psycopg2-connection.py

## Importing the main library used to connect to Denodo via ODBC (libpq)
import                psycopg2                as                dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from               socket               import               gethostname

##    Connection    parameters    to    the    Denodo    VDP    Server
denodoserver_name                         =                         "denodoserver"

##        Default        port        for        ODBC        connections
denodoserver_odbc_port                          =                          "9996"

denodoserver_database                =                "distributed_tpcds"
```

```
denodoserver_uid                          =                          "tpcds_usr"

denodoserver_pwd                          =                          "tpcds_usr"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                           =                            gethostname()

useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                Establishing                 a                 connection
cnxn_str   =   "user=%s   password=%s   host=%s   dbname=%s   port=%s"   %\
                  (denodoserver_uid,  denodoserver_pwd,  denodoserver_name,
denodoserver_database,                                  denodoserver_odbc_port)

cnxn                      =                     dbdriver.connect(cnxn_str)

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query         =         "select         *         from         bv_date_dim"

##      Define      a      cursor      and      execute      the      results
cur                               =                               cnxn.cursor()

cur.execute(query)

##  Finally  fetch  the  results.  `results`  is  a  list  of  lists.
##  If  you  don't  want  to  load  all  the  records  in  memory,  you  may
##     want     to     use     cur.fetchone()     or     cur.fetchmany()
results = cur.fetchall()
```

4.4

4.5 EXAMPLE CODE: DSN-LESS TURBODBC

In the following example, we connect to a Denodo server using the turbodbc library in DSN-less mode. Note that the additional feature of putting the results into a numpy array requires turbodbc 3.3.0.

```
##          Script          name:          turbodbc-dsnless-connection.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  ODBC
import                turbodbc                as                dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from              socket              import              gethostname

##    Connection    parameters    to    the    Denodo    VDP    Server
denodoserver_name                         =                         "denodoserver"
##  ODBC  driver.  It  must  be  installed  beforehand.  See  instructions
```

```
##
https://community.denodo.com/docs/html/browse/8.0/vdp/developer/access_through_odbc/access_through_odbc
odbcdriver              =             "DenodoODBC             Unicode(x64)"

##         Default         port         for         ODBC         connections
denodoserver_odbc_port = "9996"

denodoserver_database                  =                  "distributed_tpcds"

denodoserver_uid                         =                        "tpcds_usr"

denodoserver_pwd                         =                        "tpcds_usr"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                          =                          gethostname()
useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                 Establishing                 a                 connection
## The connect function has an additional 'turbodbc_options' parameter that
may
## be used to tune performance parameters. See the documentation for further

## information (to read the doc. from a Python client, you can issue:
##                          turbodbc.make_options?                          )
cnxn                           =                              dbdriver.connect(
                                                      driver  =  odbcdriver,
                                                  server  =  denodoserver_name,
                                               port  =  denodoserver_odbc_port,
                                            database  =  denodoserver_database,
                                                     uid  =  denodoserver_uid,
                                                     pwd  =  denodoserver_pwd,
                                                      useragent  =  useragent,
                                                           turbodbc_options  =
dbdriver.make_options(use_async_io=True)
                                                                              )

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query        =        "select        *        from        bv_store_returns"

##     Define     a     cursor     and     execute     the     results
cur                                =                                cnxn.cursor()
cur.execute(query)

##   Finally   fetch   the   results.   `results`   is   a   list   of   lists.
##  If  you  don't  want  to  load  all  the  records  in  memory,  you  may
##     want     to     use     cur.fetchone()     or     cur.fetchmany()
results                            =                             cur.fetchall()

#                                >>                               len(results)
#                                                                       287514
#                                >>                              type(results)
```

```
#                                                                    list

#                            >>                              type(results[0])
#                                                                    list

#                                >>                                  results[0]
# [2451794,40096,1,...]

## An interesting feature of turbodbc is that it can return the data
## already in numpy MaskedArray, one per column. This will:
## 1. speed up the data fetch process.
## 2. dramatically speed up further calculations on the input data, in
##    particular the transformation to a pandas DataFrame.
## 3. lower by a fairly large factor the memory footprint of the loaded
Python
##    object
results_array = cur.fetchallnumpy()

# >> type(results_array)
# collections.OrderedDict
# >> results["sr_returned_date_sk"]
# masked_array(data=[2451794, 2452608, 2452694, ..., 2452388, 2452548,
#                    2452649],
#              mask=[False, False, False, ..., False, False, False],
#     fill_value=999999,
#              dtype=int64)

# >> type(results["sr_returned_date_sk"])
#                                              numpy.ma.core.MaskedArray
```

4.6 EXAMPLE CODE: PYODBC WITH DSN

In the following example, we connect to a Denodo server using the pyodbc library and by specifying a DSN (Data Source Name).

When choosing to use a DSN-based or a DSN-less connection, take into account that by using a DSN-based connection we keep the connection script cleaner as we hide connection parameters and complexity to the final users. However DSN definition, maintenance and modification are typically a sysadmin task, meaning that their use may slow down development iterations in early phases of the projects.

```
##            Script            name:            pyodbc-dsn-connection.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  ODBC
import                  pyodbc                  as                  dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from                socket                import                gethostname
```

```
## ODBC DSN. It must be configured beforehand. See instructions
##
https://community.denodo.com/docs/html/browse/8.0/vdp/developer/access_through_odbc/access_through_odbc
denodoserver_dsn                    =                    "Denodo_DSN_tpcds"

##     Create     the     useragent     has     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                          =                          gethostname()
useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                 Establishing                 a                 connection
cnxn  =  dbdriver.connect(  "DSN=%s;UserAgent=%s"  %  (  denodoserver_dsn  ,
useragent                                    )                                )

##     Query     to     be     sent     to     the     Denodo     VDP     Server
query        =        "select        *        from        bv_store_returns"

##     Define     a     cursor     and     execute     the     results
cur                                  =                                cnxn.cursor()
cur.execute(query)

##  Finally  fetch  the  results.  `results`  is  a  list  of  pyodbc.Row,
##     that     is     a     datatype     similar     to     a     tuple
## If you don't want to load all the records in memory, you may want to use
cur.fetchone()                                                          or
##                                                                cur.fetchmany()
results                                =                         cur.fetchall()

#                                 >>                                len(results)
#                                                                         287514
#                                 >>                              type(results)
#                                                                         list

#                               >>                              type(results[0])
#                                Out[5]:                              pyodbc.Row

#                                  >>                                 results[0]
#  (2451794,  40096,  1,  7157,  910283,  6421,  37312,  8,  16,  9750,  41,
Decimal('72.57'), Decimal('6.53'), ...)
```

4.7 EXAMPLE CODE: DSN-LESS PYODBC

In the following example, we connect to a Denodo server with a DSN-less connection using the pyodbc library.

```
##           Script           name:           pyodbc-dsnless-connection.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  ODBC
import                  pyodbc                  as                  dbdriver
```

```
##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from                socket                import                gethostname

##     Connection     parameters     to     the     Denodo     VDP     Server
denodoserver_name                       =                       "denodoserver"
##  ODBC  driver.  It  must  be  installed  beforehand.  See  instructions
##
https://community.denodo.com/docs/html/browse/8.0/vdp/developer/access_throu
gh_odbc/access_through_odbc
odbcdriver              =              "DenodoODBC              Unicode(x64)"
##         Default         port         for         ODBC         connections
denodoserver_odbc_port                          =                          "9996"
denodoserver_database                   =                 "distributed_tpcds"
denodoserver_uid                           =                           "admin"
denodoserver_pwd                           =                           "admin"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                         =                         gethostname()
useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                 Establishing                 a                 connection
cnxn                            =                            dbdriver.connect(
                                                        driver  =  odbcdriver,
                                                    server  =  denodoserver_name,
                                                  port  =  denodoserver_odbc_port,
                                               database  =  denodoserver_database,
                                                       uid  =  denodoserver_uid,
                                                       pwd  =  denodoserver_pwd,
                                                        useragent  =  useragent,
                                                                              )

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query        =        "select        *        from        bv_store_returns"

##     Define     a     cursor     and     execute     the     results
cur                                =                                cnxn.cursor()
cur.execute(query)

##  Finally  fetch  the  results.  `results`  is  a  list  of  pyodbc.Row,
##     that     is     a     datatype     similar     to     a     tuple
## If you don't want to load all the records in memory, you may want to use
cur.fetchone()                                                          or
##                                                                cur.fetchmany()
results                            =                            cur.fetchall()

#                             >>                              len(results)
#                                                                   287514
#                             >>                              type(results)
#                                                                     list

#                            >>                            type(results[0])
```

```
#                              Out[5]:                                pyodbc.Row

#                                 >>                                  results[0]
#  (2451794,  40096,  1,  7157,  910283,  6421,  37312,  8,  16,  9750,  41,
Decimal('72.57'), Decimal('6.53'), ...)
```

4.8 EXAMPLE CODE: JAYDEBEAPI

In the following example, we connect to a Denodo server using the jaydebeapi library and the Denodo JDBC driver.

To be able to establish the connection between Denodo and Python the JDBC driver must be accessible from the machine where the Python interpreter is installed and a Java interpreter must be installed in the same machine.

```
##             script             name:             jaydebeapi-connection.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  JDBC
import                jaydebeapi                as                dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from                socket                import                gethostname


# Connection parameters of the Denodo Server that we are connecting to
denodoserver_name                    =                    "denodoserver"

#    This    is    the    standard    port    for    jdbc    connections
denodoserver_jdbc_port                        =                        "9999"

denodoserver_database                  =                "distributed_tpcds"

denodoserver_uid                        =                      "tpcds_usr"

denodoserver_pwd                        =                      "tpcds_usr"

denododriver_path  =  "/opt/denodo/8.0/tools/client-drivers/jdbc/denodo-vdp-
jdbcdriver.jar"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                         =                         gethostname()
useragent     =     "%s-%s"     %     (dbdriver.__name__,client_hostname)

## Creating a variable with the connection uri. We add here the UserAgent
## so the query can be better identified on the server. To append parameters
you
## can use the syntax <param_name>=<param_value> and separate them with '&'.
##   The   full   list   of   accepted   parameters   is   available   here
##          https://community.denodo.com/docs/html/browse/7.0/vdp/developer/
```

```
##    access_through_jdbc/parameters_of_the_jdbc_connection_url/
##    parameters_of_the_jdbc_connection_url
conn_uri    =    "jdbc:vdb://%s:%s/%s?userAgent=%s"    %    (denodoserver_name,
                                          denodoserver_jdbc_port,
                                                  denodoserver_database,
                                                          useragent)

cnxn        =        dbdriver.connect(        "com.denodo.vdp.jdbc.Driver",
                      conn_uri,
                      driver_args     =     {"user":     denodoserver_uid,
                                           "password":     denodoserver_pwd},
                      jars               =               denododriver_path
                      )

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query        =        "select        *        from        bv_store_returns"

##      Define      a      cursor      and      execute      the      results
cur                              =                                  cnxn.cursor()
cur.execute(query)

##   Finally   fetch   the   results.   `results`   is   a   list   of   tuples,
##   If   you   don't   want   to   load   all   the   records   in   memory,
##    you    may    want    to    use    cur.fetchone()    or    cur.fetchmany()
results                                =                                  cur.fetchall()

#                                >>                                  len(results)
#                                                                          287514
#                                >>                                type(results)
#                                                                           list
#                               >>                               type(results[0])
#                                                                         tuple
#                                  >>                                 results[0]
# (2451794, 40096, 1, 7157, 910283, 6421, 37312, ...)
```

5 ODBC/JDBC WITH KERBEROS AUTHENTICATION

5.1 PREREQUISITES

The Virtual DataPort Server must be configured to accept Kerberos Authentication. This setup is explained step-by-step in the Kerberos Authentication section of the documentation.

Also if you are using Denodo 8 without any update or Denodo 7, to be able to connect with Kerberos to a database via ODBC you have to enable the option **Use Kerberos/OAuth authentication for ODBC connections**, please see here for instructions and some considerations about this configuration option. If you are on update 20210209 of Denodo 8 or later, you are not concerned by this.

In the case of JDBC, connection details and configuration are explained in this documentation page.

Assuming here that the data fetching code is to be run in a Linux machine, you will need to have a ticket cache with the credentials used to connect to Virtual DataPort.

You can check whether the ticket cache already exists in your machine.

```
$ klist
klist: No credentials cache found (filename: /tmp/krb5cc_1000)
```

This output indicates that the cache does not exist yet.

```
$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: HTTP/john@LOCALENV.COM

Valid starting Expires Service principal
03/16/2021 11:11:46 03/16/2021 21:11:46 krbtgt/LOCALENV.COM@LOCALENV.COM
        renew until 03/17/2021 11:11:46
```

This output indicates that the cache already exists in file `/tmp/krb5cc_1000`, for the user principal `HTTP/john@LOCALENV.COM`

The ticket cache can be created with the command:

```
$ kinit -k -t john.keytab HTTP/john@LOCALENV.COM
```

With `john.keytab` being the keytab file and `HTTP/john@LOCALENV.COM` the user principal.

5.2 EXAMPLE CODE: DSN-LESS TURBODBC WITH KERBEROS

In this code snippet, we are connecting to the Virtual DataPort Server with a Kerberos user principal, without specifying user and password. The only modification to the previous DSN-less connection code consists in passing a string to the `connect` method and that we are specifying `UseKerberos=1`. Make sure you review the section of the relevant documentation section on ODBC DSN-less connection with Kerberos.

```
##        Script        name:        turbodbc-dsnless-connection-kerberos.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  ODBC
import                turbodbc                as                dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from               socket               import               gethostname

##    Connection    parameters    to    the    Denodo    VDP    Server
denodo_vdp_server           =           "denodo8-sa-20210209.localenv.com"

##  ODBC  driver.  It  must  be  installed  beforehand.  See  instructions:
##
https://community.denodo.com/docs/html/browse/8.0/vdp/developer/access_throu
gh_odbc/access_through_odbc
odbcdriver             =             "DenodoODBC             Unicode(x64)"

##    '9996'    is    the    default    port    for    ODBC    connections
denodo_odbc_port                          =                          "9996"

##      Database      that      we      want      to      connect      to
denodo_db                       =                       "distributed_tpcds"

##     Create     the     useragent     as     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                       =                       gethostname()

useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                Establishing                a                connection
cnxn_str   =   "DRIVER={%s};SERVER=%s;DATABASE=%s;PORT=%s;UseKerberos=1"   %
(odbcdriver,      denodo_vdp_server,      denodo_db,      denodo_odbc_port)

cnxn            =            dbdriver.connect(connection_string=cnxn_str)

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query         =         "select         *         from         bv_date_dim"

##      Define      a      cursor      and      execute      the      results
cur                           =                           cnxn.cursor()

cur.execute(query)

##                  Fetch                  the                  results
```

```
results = cur.fetchallnumpy()
```

5.3 EXAMPLE CODE: JAYDEBEAPI WITH KERBEROS

The only difference with the code snippet seen before explaining the use of jaydebeapi without Kerberos is that we are now passing the Kerberos related parameters (useKerberos, useTicketCache and ticketCache) instead of user and password in the connection uri.

To know more about access to Denodo via JDBC with Kerberos, please check this documentation page.

```
##          script          name:          jaydebeapi-connection-kerberos.py

##  Importing  the  main  library  used  to  connect  to  Denodo  via  JDBC
import                jaydebeapi                as                dbdriver

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from                socket                import                gethostname

# Connection parameters of the Denodo Server that we are connecting to
denodoserver_name                        =                        "denodoserver"
#    This    is    the    standard    port    for    jdbc    connections
denodoserver_jdbc_port                          =                          "9999"
denodoserver_database                  =                "distributed_tpcds"
denododriver_path  =  "/opt/denodo-vdp-jdbcdriver/denodo-vdp-jdbcdriver.jar"

##     Create     the     useragent     has     the     concatenation     of
##    the    client    hostname    and    the    python    library    used
client_hostname                          =                          gethostname()

useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

## Creating a variable with the connection uri. We add here the UserAgent
## so the query can be better identified on the server. To append parameters
you
## can use the syntax <param_name>=<param_value> and separate them with '&'.
##   The   full   list   of   accepted   parameters   is   available   here
##          https://community.denodo.com/docs/html/browse/7.0/vdp/developer/
##                access_through_jdbc/parameters_of_the_jdbc_connection_url/
##                                      parameters_of_the_jdbc_connection_url
conn_uri    =    "jdbc:vdb://%s:%s/%s?userAgent=%s"    %    (denodoserver_name,
                                                    denodoserver_jdbc_port,
                                                     denodoserver_database,
                                                                 useragent)

cnxn        =        dbdriver.connect(        "com.denodo.vdp.jdbc.Driver",
                                                                    conn_uri,
                                        driver_args  =  {  "useKerberos":"true",
                                                      "useTicketCache":"true",
```

```
                                        "ticketCache":"/tmp/krb5cc_1000"
                                                                      },
                                             jars  =  denododriver_path
                                                                      )

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query       =       "select       *       from       bv_store_returns"

##     Define     a     cursor     and     execute     the     results
cur = cnxn.cursor()

cur.execute(query)

##  Finally  fetch  the  results.  `results`  is  a  list  of  tuples,
##  If  you  don't  want  to  load  all  the  records  in  memory,
##   you   may   want   to   use   cur.fetchone()   or   cur.fetchmany()
results = cur.fetchall()
```

6 GETTING THE DATA INTO A PANDAS DATAFRAME

In this section, we will see how the results imported in python can be transformed to a Pandas dataframe, the de-facto reference data structure to perform data transformation, cleaning and analysis in Python.

Note that the types_dic portion is subject to change depending on the types present in the result set from the source.

```
##          Script          name:          results-to-pandas-dataframe.py

##   You   must   run   one   of   the   scripts   pyodbc-*-connection.py   in
##  the  same  session  so  that  'cur'  and  'results'  objects  are  defined.
##  You  may  also  want  to  run  other  *-connection.py  script  instead
##  but  the  types  conversion  will  fail,  as  the  types  depend  on  the
##                              import                                  library

import                   pandas                   as                    pd
import                                                                decimal
import                   numpy                    as                    np

##      Get      the      column      names      from      the      cursor
columns       =       [c[0]       for       c       in       cur.description]

##    Get    the    data    in    'results'    into    a    Pandas    dataframe
df_results    =    pd.DataFrame.from_records(results,columns    =    columns)

#                           >>                                df_results.info()
#                  <class                    'pandas.core.frame.DataFrame'>
#       RangeIndex:       287514       entries,       0       to       287513
#          Data          columns          (total          20          columns):
#    #     Column                                  Non-Null  Count     Dtype
#  ---    ------                                   --------------     -----
#      0        sr_returned_date_sk           277502   non-null      float64
#     1        sr_return_time_sk              277568   non-null      float64
#    2      sr_item_sk                        287514  non-null    int64
#                                                                          ...
#    19    sr_net_loss                        277541  non-null    object
#          dtypes:          float64(9),          int64(2),          object(9)
#              memory              usage:              43.9+              MB

## As we can see in the above informative table, types are not translated
## correctly: int are read as floats (because of the  presence of null
values)
## and decimal are read as object. As an additional step you may want to do,

## is to fix it by defining a type mapping dictionary and then loop over
## the columns to fix their type, one by one. The types dictionary is to be
## modified if we use other libraries to import the data (ex. jaydebeapi)
```

```
types_dict                                  =                                  {
                                                                  int:pd.Int64Dtype(),
                                                               decimal.Decimal:np.float
}

types     =     [types_dict[c[1]]     for     c     in     cur.description]
for         c,tp         in                    zip(df_results.columns,types):
                         df_results[c]      =      df_results[c].astype(tp)

#                             >>                              df_results.head()
#        sr_returned_date_sk    sr_return_time_sk    ...    sr_store_credit
sr_net_loss
# 0                 2451794                 40096   ...              39.60
600.91
# 1                 2452608                 38280   ...              37.94
490.10
# 2                 2452694                 30733   ...              84.11
45.17
# 3                 2451783                 46066   ...             106.62
295.05
# 4                 2452016                 33347   ...               9.82
106.21

#                             >>                              df_results.info()
#                    <class                    'pandas.core.frame.DataFrame'>
#       RangeIndex:       287514       entries,       0       to       287513
#         Data         columns          (total         20         columns):
#    #     Column                                  Non-Null  Count     Dtype
#  ---    ------                                   --------------     -----
#     0        sr_returned_date_sk          277502   non-null     Int64
#     1       sr_return_time_sk             277568   non-null     Int64
#    2      sr_item_sk                      287514  non-null    Int64
#                                                                        ...
#     11     sr_return_amt                    277486   non-null     float64
#                                                                        ...
#                dtypes:                Int64(11),                float64(9)
# memory usage: 46.9 MB
```

You can do the cursor creation, query execution, fetching and transform to pandas dataframe in a single instruction, that is `pd.read_sql`.

To understand the usage of this function, compare this code snippet.

```
cnxn             =             dbdriver.connect(connection_string=cnxn_str)

query          =          "select          *          from          bv_date_dim"

df_results = pd.read_sql(query, cnxn)
```

With the approach used above:

```
cnxn             =             dbdriver.connect(connection_string=cnxn_str)

query         =         "select         *         from         bv_date_dim"

cur = cnxn.cursor()

cur.execute(query)

results                          =                          cur.fetchall()

columns       =       [c[0]       for       c       in       cur.description]

df_results = pd.DataFrame.from_records(results,columns = columns)
```

By default pandas performs a fetchall when using read_sql but offers the possibility of reading in chunks. Here is an example of SQLAlchemy with pandas reading in chunks

```
##          Script          name:          sqlalchemy-denodo-connection.py

## Importing the main library used to connect to Denodo via sqlalchemy
import sqlalchemy as dbdriver

## Importing pandas
import                   pandas                   as                   pd

##    Importing    the    gethostname    function    from    socket    to
##     put     the     hostname     in     the     useragent     variable
from               socket               import               gethostname

##    Connection    parameters    to    the    Denodo    VDP    Server
denodoserver_name                        =                        "denodoserver"

##         Default         port         for         ODBC         connections
denodoserver_odbc_port                          =                          "9996"

denodoserver_database                  =                "distributed_tpcds"

denodoserver_uid                         =                         "tpcds_usr"

denodoserver_pwd                         =                         "tpcds_usr"

##    Create    the    useragent    as    the    concatenation    of
##    the    client    hostname    and    the    python    library    used
client_hostname                          =                          gethostname()

useragent      =      "%s-%s"      %      (dbdriver.__name__,client_hostname)

##                Establishing                a                connection
cnxn_str            =            "denodo://%s:%s@%s:%s/%s"            %\
                (denodoserver_uid,  denodoserver_pwd,  denodoserver_name,
denodoserver_odbc_port, denodoserver_database)
```

```
engine=dbdriver.create_engine(cnxn_str)

##    Query    to    be    sent    to    the    Denodo    VDP    Server
query = "select * from bv_date_dim"
connection = engine.connect().execution_options(
        stream_results=True, max_row_buffer=1000)

for df in pd.read_sql(query, connection, chunksize=1000):
    display(df)
connection.close()
```

7 APPENDIX

7.1 PASSWORD ENCRYPTION

In many of the above scripts we used a plain password to connect to the Denodo Server. This may be acceptable for development and testing purposes but it is often forbidden in more security-sensitive installations such as production environments.

One of the possible workarounds for this is using the pycryptodome package. Provided that pycryptodome package is installed, the following code encrypts and saves the password for the “tpcds_usr” user:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_EAX)
ciphertext, tag = cipher.encrypt_and_digest(b'tpcds_usr')

file_out = open("/tmp/pwd.bin", "wb")
[ file_out.write(x) for x in (cipher.nonce, tag, ciphertext) ]
file_out.close()

## Of course it is not a good idea to store the
## password and the key in the same place, but we are just giving an
## example here
key_out = open("/tmp/key.bin", "wb")
key_out.write(key)
key_out.close()
```

And the following code decrypts the password with the given key. The `pwd` variable is to be used in the `dbdriver.connect` function

```
## Read the key and the encrypted password then decrypt it
from Crypto.Cipher import AES

file_in = open("/tmp/pwd.bin", "rb")
nonce, tag, ciphertext = [ file_in.read(x) for x in (16, 16, -1) ]
file_in.close()

file_key_in = open("/tmp/key.bin","rb")
key = file_key_in.read()
file_key_in.close()

cipher = AES.new(key, AES.MODE_EAX, nonce)
pwd = cipher.decrypt_and_verify(ciphertext, tag).decode("utf-8")
```

7.2 PYTHON ENVIRONMENT

A test Python environment may be set up with the following steps, that will configure a working environment when deployed in the conditions listed in section Software Versions.

Installation of the required system packages:

```
sudo apt install g++ python3-dev unixodbc unixodbc-dev libboost-all-dev
openjdk-14-jdk virtualenv
```

Initialization of a Python virtual environment:

```
mkdir /opt/pyvenv/
virtualenv --python=python3.8 /opt/pyvenv/denodo-to-python
```

Activation and preparation of the Python virtual environment with the required modules:

```
cd /opt/pyvenv/denodo-to-python
source ./bin/activate
pip install -r <requirements.txt>
```

Where `requirements.txt` is the environment file, that may copied and pasted from here:

```
asttokens==2.2.1
backcall==0.2.0
colorama==0.4.6
decorator==5.1.1
denodo-sqlalchemy==20220921
executing==1.2.0
greenlet==2.0.1
ipython==8.7.0
JayDeBeApi==1.2.3
jedi==0.18.2
JPype1==1.4.1
matplotlib-inline==0.1.6
numpy==1.24.1
packaging==22.0
pandas==1.5.2
parso==0.8.3
pickleshare==0.7.5
pip==22.3.1
prompt-toolkit==3.0.36
psycopg2-binary==2.9.5
pure-eval==0.2.2
Pygments==2.14.0
pyodbc==4.0.35
python-dateutil==2.8.2
```

```
pytz==2022.7
setuptools==65.5.0
six==1.16.0
SQLAlchemy==1.4.45
stack-data==0.6.2
traitlets==5.8.0
turbodbc==4.5.6
wcwidth==0.2.5
```

For both the operating system packages, installed via apt, and the python packages, installed via pip, of course you will need them all only when you want to access Denodo views with all the methods exposed in this article. More realistically you will choose your access method and install the minimum set of packages that allows its usage: for example openjdk is only needed for jaydebeapi, while unixodbc and unixodbc-dev are only needed for ODBC-based methods.

7.3 TESTED SOFTWARE VERSIONS

Python	3.11.1
Denodo & Denodo JDBC/ODBC drivers	8.0