DENODO VIRTUAL DATAPORT 6.0 DEVELOPER GUIDE

Update May 26th, 2016

Denodo Technologies North America & APAC: 530 Lytton Avenue, Suite 301. Palo Alto, CA 94301. USA
Denodo Technologies Iberia & Latinoamérica: C/Montalbán, 5, 28014 Madrid. SPAIN
Denodo Technologies EMEA: 19th Floor, Portland House, Bressenden Place, London SW1E 5RS. UK
www.denodo.com

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (*http://www.openssl.org/*)

This product includes cryptographic software written by Eric Young (*eay@cryptsoft.com*)

This product includes software written by Tim Hudson (*tjh@cryptsoft.com*)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

SCOPE

Denodo Virtual DataPort enables applications to access integrated views of different heterogeneous and distributed data sources. This document introduces users to the mechanisms available for client applications to use Denodo Virtual DataPort. It also provides the necessary information to develop individual client applications.

WHO SHOULD USE THIS DOCUMENT

This document is aimed at developers seeking to access Virtual DataPort from a client application. Detailed information on how to install the system and administrate it is provided in other manuals to which reference will be made as the need arises.

SUMMARY OF CONTENTS

More specifically this document describes:

- How to access a Virtual DataPort server using the Denodo JDBC driver from your Java application.
- How to access a Virtual DataPort server from your ODBC application.
- How to access Virtual DataPort views published as Web Services from client applications.
- The generic API based on Web Services for executing any VQL statement on a Virtual DataPort server.
- How to develop extensions for a Virtual DataPort server: custom functions, Denodo Stored Procedures and custom wrappers.

1 INTRODUCTION

Denodo Virtual DataPort is a **global solution** for integrating heterogeneous and distributed data sources.

Virtual DataPort integrates the data that are relevant to the company, regardless of their origin, format and level of structure. It incorporates these data into its data system in real time or with configurable preloads and facilitates the construction of services of high strategic and functional value for both corporate and business use.

Virtual DataPort is based on a client-server architecture, where clients issue statements to the server written in VQL (V*irtual Query Language*), a SQL-like language used to query and define data views (see the *Administration Guide* [ADMIN_GUIDE] and the *VQL Advanced Guide* [VQL]).

This document introduces product users to the mechanisms available for client applications to use Denodo Virtual DataPort, making the most of its data integration facilities. See the *Administration Guide* [ADMIN_GUIDE] to obtain information on how to install, configure and administer the Virtual DataPort server and how to use it to create unified views of data from heterogeneous and distributed data sources.

Client applications can access Virtual DataPort in several ways:

- Using the JDBC interface (Java Database Connectivity) [JDBC]. Virtual DataPort provides a JDBC driver that client applications can use for this purpose (see section 2)
- Using the ODBC interface (Open Database Connectivity) [ODBC]. Virtual DataPort provides an ODBC interface and an ODBC driver for ODBC clients (see section 3)
- Using an ADO.NET Data Provider (see section 4).
- Using the SOAP and REST Web Service interfaces:
    - Access through the generic RESTful Web Service interface. This service is automatically deployed on *http://localhost:9090/denodo-restfulws*.
    - With the Virtual DataPort administration tool, you can publish SOAP or REST Web services that publish the contents of one or more views. See more about this in the section "Publication of Web services" of the Virtual DataPort Administration Guide [ADMIN_GUIDE].

1.1 EXAMPLES

In the `DENODO_HOME/samples/vdp/` directory, there are several examples of clients that retrieve data from Virtual DataPort:

- `vdp-clients` contains sample programs that connect to Virtual DataPort through the JDBC interface and a SOAP Web service.
- `vdp-clients-ADO.NET` contains a C# program that, using an ADO.Net data provider, it connects directly to Virtual DataPort's ODBC interface or through a Windows ODBC DSN. Then, it executes a query.

- `vdp-clients-C++` contains C++ program that connects to Virtual DataPort through the ODBC interface and executes a query.

- `vdp-clients-EntityFramework` contains a C# program that sends a query to Virtual DataPort, using the Entity Framework.

See more about these examples in the `README` file of these directories.

2 ACCESS THROUGH JDBC

JDBC (Java Database Connectivity) is a Java API that allows executing statements on a relational database regardless of the Database Management System used.

Virtual DataPort provides a driver that implements the main characteristics of the JDBC 4.1 API [JDBC]. These are some of the features of the JDBC specification supported by the Virtual DataPort JDBC driver:

- The data types supported are defined in the Advanced VQL Guide [VQL] (includes support for all basic types and for fields of the type array and register).
- Execution of statements to query, insert, update and delete data. In addition, to create new elements such as data sources, views, etc.
- Support for metadata description statements and listing of server catalog elements.
- Support for `PreparedStatements`.
- Support for canceling the current statement execution by using the `cancel()` method of the `java.sql.Statement` class.
  When a query is cancelled, the Virtual DataPort Server will cancel all current accesses to data sources and cache. After invoking the `cancel` method, it is still possible for the server to return some results, if these were retrieved before the source access canceling were effective. Therefore, the query cancellation does not imply closing the `ResultSet` that is being used.
- Invocation of stored procedures using the `CALL` statement [VQL].
- Support for submitting batches of commands.
- The `ResultSet` objects returned by Virtual DataPort are not updatable (i.e. `CONCUR_READ_ONLY`) and have a cursor that moves forward only (i.e. `TYPE_FORWARD_ONLY`). In addition, the `ResultSet` objects are closed when the current transaction is committed (i.e. `CLOSE_CURSORS_AT_COMMIT`).

The JDBC driver is at
`<DENODO_HOME>/tools/client-drivers/jdbc/vdp-jdbcdriver-core/denodo-vdp-jdbcdriver.jar`

This directory also contains the file `denodo-vdp-jdbcdriver-basic.jar` is. The section 2.3 explains when you should use this jar file instead of the `denodo-vdp-jdbcdriver.jar`.

The class that implements the driver is `com.denodo.vdp.jdbc.Driver`.

The syntax of the database URL is

```
jdbc:vdb://<hostName>:<port>/[<databaseName>]
    [?<paramName>=<paramValue> [&<paramName>=<paramValue>]* ]
```

Figure 1 Syntax of the JDBC connection URL

For example:

```
jdbc:vdb://localhost:9999/admin?queryTimeout=100000&chunkTimeout=100
0&userAgent=myApplication
```

Figure 2 JDBC connection URL sample

If the name of the database contains non-ASCII characters, they have to be URL-encoded. For example, if the name of the database is "テスト", the connection URL to the database will be this:

```
jdbc:vdb://localhost:9999/%E3%83%86%E3%82%B9%E3%83%88?queryTimeout=9
00000&chunkTimeout=1000&userAgent=myApplication&autoCommit=true
```

Figure 3 JDBC connection URL sample with non-ASCII characters

The path `DENODO_HOME/samples/vdp/vdp-clients` contains examples of client programs accessing DataPort through JDBC (the `README` file of this path explains how to generate and publish the views accessed by the clients in the example).

2.1 PARAMETERS OF THE JDBC CONNECTION URL

The table below lists the optional configuration parameters of the URL and their default value:

Parameter of the URL	Description
`autoCommit`	If `true`, the invocations to the methods of the JDBC API responsible of managing transactions are ignored. I.e. the driver ignores the invocations to the methods `setAutoCommit(…)`, `commit()` and `rollback()`. This is useful to make sure that an application does not start transactions inadvertently. Even with this parameter set to true, an application can start and finish transactions by executing the statements `BEGIN`, `COMMIT` and `ROLLBACK`. If the client invokes `setAutoCommit(false)`, after executing a `COMMIT`, the driver will not start a new transaction until it has to execute another statement. If the client invokes `setAutoCommit(false)`, take into account the limits on the duration of a transaction: • By default, transactions cannot last for more than 30 minutes. • Once the execution of a statement finishes, the client has to execute another statement in less than 30 seconds. See more about the limits on the duration of transaction in the section "Transactions in Virtual DataPort" of the Advanced VQL Guide. Default value: none.
`chunkSize`	The results of a query can be divided into blocks (chunks), so the Server does not have to wait for the query to finish, in order to begin sending part of the results to the client. This parameter establishes the maximum number of results that a block can contain. When the Server obtains enough results to complete a block, it sends this block to the driver and continues processing the next results. In an application that uses this driver, you can either add this parameter to the connection URL and/or before executing the query, invoke the method `setFetchSize` of the class `Statement`. The value set with the `setFetchSize` method overrides the value set in the URL. Default value: `1000`

chunkTimeout	This parameter establishes the maximum time (in milliseconds) the Server waits before returning a new block to the driver. When this time is exceeded, the Server sends the current block to the driver, even if it does not contain the number of results specified in the chunkSize parameter. **Note**: if chunkSize and chunkTimeout are 0, the Server returns all the results in a single block. If both values are different than 0, the Server returns a chunk whenever one of these conditions happen first: • The chunk is filled (chunkSize) • Or, after a certain time of not sending any chunk to the client (chunkTimeout) Default value: 90000 milliseconds (90 seconds)
i18n	This parameter establishes the internationalization (i18n) configuration of the connection with the Server. If not present, the driver assumes the i18n of the database that you are connecting to. The parameter i18n in the CONTEXT clause of the queries overrides the value of this parameter. Default value: I18N of the database that you are connecting to
identifiersUppercase	If true, when executing SELECT queries, the names of the fields are returned in uppercase. The default value is false. Default value: false
initSize	Number of connections that the driver will establish with the server during the initialization process. These connections will remain idle, ready to be used. Default value: 0
maxActive	When the parameter poolEnabled is true, this is the maximum number of active connections that the pool can manage at the same time. If 0, there is no limit. Default value: 30
maxIdle	When the parameter poolEnabled is true, this is the maximum number of active connections that will remain idle in the pool without being closed. If 0, there is no limit. Default value: 20

password	Default value: N/A
poolEnabled	**Note**: this parameter is deprecated and may be removed in future versions of the Denodo Platform. The parameters initSize, maxActive and maxIdle are also deprecated. If true, the driver creates a pool of connections so, when a client request a connection, instead of establishing a new connection, it returns one from the this pool. This reduces the time required to obtain a connection with the Server. If this property is false, the driver does not create a pool of connections and ignores the properties initSize, maxActive and maxIdle. Default value: false
publishViewsAsTables	If false, the metadata published by the JDBC driver describes base views as TABLE elements and the derived and interface view as VIEW elements. If true, the metadata describes all the views as TABLE elements. Some third-party tools require the JDBC metadata to publish all the views as tables in order to recognize the associations created between views. For these applications, add this parameter to the URL with the value true. Default value: false
queryTimeout	Maximum time (in milliseconds) the driver will wait for a query to finish. After this period, it will throw an Exception. This parameter is optional. If it is not set, the query timeout has the default value (900000 milliseconds). If 0, the driver will wait indefinitely until the query finishes. This parameter sets the default timeout for all the queries. In addition, you can change the timeout for a single query by adding the parameter 'QUERYTIMEOUT'='<value>' to the CONTEXT clause of the query. See more about this in the section "CONTEXT Clause" of the Advanced VQL Guide. Default value: 900000 milliseconds (15 minutes)
reuseRegistrySocket, pingQuery and pingQueryTimeOut	Parameters needed when connecting to Virtual DataPort through a load balancer. The section 2.2 explains how to use them.

`user` and `password`	User name and password used to authenticate against Virtual DataPort. In some scenarios, you may need to provide the credentials as parameters of the connection URI. Default value: N/A
`userAgent`	Sets the user agent of the connection. The section "Setting the User Agent of an Application" of the Administration Guide explains why we recommend setting this parameter. Default value: <empty>

Table 1 Parameters of the JDBC driver and their default value

Autocommit

By default, the connections opened by the Denodo JDBC driver have the property "autocommit" set to `true`. This is the recommended value and its effect is that the queries are not performed inside a transaction.

You should not change this property to `false` unless you need the statements to be executed inside the same transaction. The reason is that Virtual DataPort uses a distributed transaction manager, which uses a 2-phase commit protocol. This protocol introduces some overhead over the queries. Therefore, if you set this property to `false` without needing it, your queries will run unnecessarily slower.

2.2 CONNECTING TO VIRTUAL DATAPORT THROUGH A LOAD BALANCER

Read this section when the JDBC client is connecting to Virtual DataPort through a load balancer or another intermediate resource that holds a pool of connections to Virtual DataPort.

The table below lists the parameters of the URL that are useful when connecting to Virtual DataPort through a load balancer:

Parameter of the URL	Description
`reuseRegistrySocket`	**Important**: set this property to `false` when connecting to Virtual DataPort through a load balancer. If `false`, the requests will be more evenly distributed across the Virtual DataPort servers of the cluster than if the property is not set or set to `true`. Default value: `true`

`pingQuery` and `pingQueryTimeOut`	**Important**: only use these two properties if the load balancer or the client will execute a ping query without support to set a timeout for that query. When a client executes the query set on the parameter `pingQuery`, the JDBC driver returns an error if that query does not finish in the number of milliseconds set on `pingQueryTimeout`. See below for a more detailed explanation of these properties. Default value for both parameters: <empty>

Table 2 Parameters of the JDBC driver useful to set-up a cluster of Denodo servers

Sample URL for JDBC applications that connect to Virtual DataPort through a load balancer:

```
jdbc:vdb://acme:9999/support?reuseRegistrySocket=false
```

Sample URL for JDBC applications with the parameters `pingQuery` and `pingQueryTimeout`:

```
jdbc:vdb://acme:9999/admin?reuseRegistrySocket=false&pingQuery=SELEC
T 1&pingQueryTimeOut=1000
```

With the URL above, if the query `SELECT 1` does not finish in one second, the driver returns an error.

You need to add the parameters `pingQuery` and `pingQueryTimeout` to the connection URL if the load balancer or the client meet these conditions:

- It will execute a ping query to check that the Virtual DataPort server is alive, or a connection to it is still valid.
- And it does not support setting a timeout for that ping query.

At runtime, when the JDBC driver receives the query set on the parameter `pingQuery`, it will wait for a maximum of `pingQueryTimeout` milliseconds for the query to finish. If the query does not finish in that time, the driver will return an error, which will indicate the client or the load balancer that the connection is no longer valid. A connection to a Virtual DataPort server can become invalid when it has timed out or dropped by a firewall.

2.3 WHEN TO USE THE "BASIC" VERSION OF THE JDBC DRIVER

The Denodo JDBC driver depends on the following libraries of the Apache Foundation:

- Commons Codec 1.3
- Commons Collections 3.2.1
- Commons Collections4 4.0

- Commons Lang 2.6
- Commons Pool 1.6
- Log4j 1.2.15

There are two versions of the JDBC driver to connect to Virtual DataPort (both located in the directory `<DENODO_HOME>/tools/client-drivers/jdbc/`)

1. `denodo-vdp-jdbcdriver.jar` (*recommended version*).
2. `denodo-vdp-jdbcdriver-basic.jar`

Both versions are the same except that the "basic" one does *not* contain the third-party dependencies required by the driver.

Denodo includes a "basic" version of the driver because there are scenarios where these libraries are already provided by the environment where the driver will be loaded. For example, in certain Web application servers. In these cases, you can use the "basic" driver. Make sure that the application server loads all the dependencies of the driver and their version is the same or compatible with what the driver expects. Otherwise, the driver may work incorrectly.

Unless needed, use `denodo-vdp-jdbcdriver.jar`. That way you do not have to manually download these dependencies and add them to the classpath of your application.

2.4 CONNECTING TO VIRTUAL DATAPORT WITH SSL

If SSL was enabled in the Virtual DataPort server to secure the communications, set the environment variable `javax.net.ssl.trustStore` to point to the *trustStore* that contains the certificate used by the Denodo servers. Otherwise, the driver will not be able to establish the connection with the Server.

See more about this in the section "Enabling SSL for External Clients" of the Installation Guide.

2.5 CONNECTING TO VIRTUAL DATAPORT USING KERBEROS AUTHENTICATION

Virtual DataPort provides support to authenticate its clients using the Kerberos authentication protocol.

Even if the Virtual DataPort server is configured to use Kerberos authentication, JDBC clients, by default, will use the standard authentication method.

To use Kerberos authentication from a JDBC client, do the following:

1. In your JDBC client, define these system properties:

```
-Djava.security.krb5.realm=<domain realm>
-Djava.security.krb5.kdc=<Key distribution center 1>[:<key
distribution center>]+
```

For example,

```
-Djava.security.krb5.realm=CONTOSO.COM
-Djava.security.krb5.kdc=dc-01.contoso.com
```

If there is more than one key distribution center (kdc) in your domain, add it to the java.security.krb5.kdc property separated by a colon. For example:

```
-Djava.security.krb5.realm=CONTOSO.COM
-Djava.security.krb5.kdc=dc-01.contoso.com:dc-02.contoso.com
```

2. If your application executes queries that involve a data source configured to use Kerberos and "pass-through session credentials", configure the Kerberos server (e.g. Active Directory) to return "forwardable" tickets to the user account used to run this JDBC application.

   "Forwardable" tickets allow other applications (in this case, the Virtual DataPort server) to request service tickets on behalf of the client. These service tickets will be used to connect to other services (e.g. databases, SOAP Web services, etc.) using Kerberos authentication, on behalf of your application.

   If the ticket used by the client is not forwardable, the requests that involve data sources with the option "pass-through session credentials" enabled, will fail.

   If changing the Kerberos server configuration is not possible, create the krb5 configuration file in the host where your application will run. The krb5 file has to contain the property `forwardable=true`. The appendix "Providing a KRB5 File for Kerberos Authentication" explains how to create a krb5 file and where to store it.

3. Define several connection properties (i.e. not parameters to the JDBC URI).

   The tables below list the driver properties you have to set when creating the JDBC connection. Look for the table that corresponds with the Kerberos authentication mode you want to use.

   The *documentation of the JRE regarding Kerberos* provides a detailed explanation of these properties.

Properties for the "Windows Single Sign-On (SSO)" authentication method (recommended option because is the easier to set-up):

Property	Value
`useKerberos`	`true`
`debug`	`true` Remove this property if there are no issues with the Kerberos authentication
`serverPrincipal`	Service Principal Name
`useTicketCache`	`true`

renewTGT	true

Properties for the "Use ticket cache" authentication mode with a ticket cache file:

Property	Value
useKerberos	true
debug	true Remove this property if there are no issues with the Kerberos authentication
serverPrincipal	Service Principal Name
useTicketCache	true
renewTGT	true
ticketCache	Path to the Ticket cache file

Before using this authentication mode, you need to generate a "ticket cache file" on the host where the JDBC application will run. That is, manually obtain and cache on a file a ticket-granting ticket (TGT). To do this, open a command line and execute the following:

```
%JAVA_HOME%\kinit.exe -f -c "<DENODO_HOME>\conf\vdp-
admin\ticket_cache"
```

The option `-f` requests the Key Distribution Center (KDC) to return "forwardable" tickets.

"User/password" authentication mode:

Property	Value
useKerberos	true
debug	true Remove this property if there are no issues with the Kerberos authentication
serverPrincipal	Service Principal Name
user and password	User name and password of the user. When using a JDBC client such as DbVisualizer, you can enter the credentials in the "User name" and "password" boxes, instead of providing them as a property.

2.6 EXTENSIONS TO THE STANDARD JDBC API

2.6.1 Obtaining the Names of Elements inside a Struct (Register)

When using the JDBC driver of Denodo to execute queries that return compound values, take into account the following:

- Values of type `register` are converted to `java.sql.Struct` objects.
- Values of type `array` are converted to `java.sql.Array` objects.
- `java.sql.Array` objects are arrays of `Struct` objects.

The standard JDBC API provides methods to obtain the values inside `java.sql.Struct` objects (i.e. inside `register` fields). However, it does not offer any way of obtaining the name of the subfields of a `Struct` or obtaining these values by the name of the subfield.

This section explains how to, using the Denodo JDBC API:

1. Obtain the name of the subfields of a `Struct` object.
2. Obtain a value of a subfield by its name, instead of its position inside the `register`.

For example, let us say that you have an application that executes a query that returns a `register` field whose subfields are `last_name` and `first_name`. For each row, the result set returns a `Struct` object. To obtain the values of each `Struct` object, the application has to invoke the method `Struct.getAttributes()`, which returns an array of two values: the last name and the first name. If later, you modify this register to add a subfield (e.g. `telephone`), the array returned by the method `Struct.getAttributes()` will have three elements instead of two. In addition, if the first element of the array is now the telephone and not the last name, the application will obtain invalid data.

To avoid this sort of maintainability issues you may want to use the classes of the Denodo JDBC API to obtain the values of a `Struct` by name and not by its position in the register. This will make your application more robust to changes.

The example below shows how to do this.

```
private void executeQueryThatReturnsCompoundValues()
        throws Exception {

    /*
     * The method getConnection() returns a Connection to Virtual
     * DataPort
     */
    Connection connection = getConnection();
    Statement st = connection.createStatement();
    String query = "SELECT * FROM view_with_compound_fields";
    ResultSet rs = st.executeQuery(query);

    /*
     * The classes 'VDBJDBCResultSetMetaData' and 'Field' are part
     * of the Denodo JDBC API. They do not belong to the standard
```

```
     * JDBC API.
     */
    VDBJDBCResultSetMetaData metaData =
            (VDBJDBCResultSetMetaData) rs.getMetaData();
    com.denodo.vdb.jdbcdriver.printer.Field[] fields =
            metaData.getFields();
    while (rs.next()) {

        int columnCount = metaData.getColumnCount();
        for (int i = 1; i <= columnCount; i++) {

            Object value = rs.getObject(i);
            if (value != null) {

                if (metaData.getColumnType(i) == Types.STRUCT) {
                    /*
                     * The JDBC API represents the values of type
                     * 'register' as 'Struct' objects.
                     */

                    /*
                     * The classes 'RegisterVO' and
                     * 'RegisterValueVO' are part of the Denodo JDBC
                     * API. They do not belong to the standard Java
                     * API.
                     */
                    RegisterVO vdpType =
                       ((RegisterVO) fields[i - 1].getVdpType());
                    List<RegisterValueVO> registerSubTypes =
                        vdpType.getElements();
                    Struct struct = (Struct) value;
                    Object[] structValues = struct.getAttributes();
                    String firstName = null, lastName = null;
                    for (int j=0; j <registerSubTypes.size(); j++) {
                        /*
                         * The variable 'registerSubTypes'
                         * contains the names of the names of the
                         * subfields.
                         */

                        String subFieldName =
                            registerSubTypes.get(j).getName();
                        switch (subFieldName) {
                        case "first_name":
                            firstName = (String) structValues[j];
                            break;

                        case "last_name":
                            lastName = (String) structValues[j];
                            break;
                        }
                        /*
                         * ...
                         */
                    }
                } else if (metaData.getColumnType(i)==Types.ARRAY) {
                    /*
                     * The JDBC API represents the values of type
```

```
                        * 'array' as 'Array' objects.
                        */
                       Object[] register =
                           (Object[]) rs.getArray(i).getArray();
                       for (Object o : register) {
                           /*
                            * In the Denodo JDBC API, the content of an
                            * 'Array' is an array of 'Struct' objects.
                            */

                           Struct s = (Struct) o;
                           /*
                            * ...
                            */
                       }
                   } // else ...
               }
           }
       }

       /*
        * Close ResultSet, Statement and Connection.
        */
}
```

Figure 4 Obtaining the name of a value inside a Struct object

3 ACCESS THROUGH ODBC

ODBC (Open Database Connectivity [ODBC]) is a standard API specification for using database management systems.

Virtual DataPort provides an ODBC interface and an ODBC driver.

By using an ODBC driver, we can use applications query Virtual DataPort from applications that do not support JDBC, such as Excel.

3.1 CONFIGURATION OF THE ODBC DRIVER ON WINDOWS

The Denodo Platform provides an ODBC driver for Windows, which is based on the ODBC PostgreSQL driver.

As any other ODBC drivers, you have to install it in the machine where the client application runs.

3.1.1 Install the ODBC Driver

Follow these steps:

1. Go the folder `<DENODO HOME>\tools\client-drivers\odbc` and unzip the appropriate ODBC driver.

   DenodoODBC_x86.zip contains the ODBC driver for 32-bit clients.

   DenodoODBC_x64.zip contains the ODBC driver for 64-bit clients.

   Select the 32 bits or the 64 bits version depending on the client that will use it. E.g., a 32 bits client such as Microsoft Excel 2003 can only use a 32 bits ODBC driver, even if it is running on a 64 bits O.S and is going to connect to a Virtual DataPort server running with the 64 bit JRE.

2. Execute the "msi" file extracted from the zip file.

3. The installation wizard is very simple: click "Next" in all the dialogs.

The installation is now complete.

You can install the 32-bit and the 64-bit ODBC driver on the same host so all the applications can use this ODBC driver regardless of its "bitness".

3.1.2 Set up a DSN on Windows

Make sure that you have logged in using an account with administrative privileges.

Follow these steps:

1. Open the ODBC Data Sources applet of the Windows Administrative Tools (Control Panel).

   Important: if you have installed and want to use the 32-bit ODBC driver in a 64 bits Windows, instead of opening this applet, run

`%SystemRoot%\SysWOW64\odbcad32.exe`
This command opens the dialog to configure 32 bits DSNs.

2. Open the System DSN tab and click on **Add**.

   The difference between a "System DSN" and a "User DSN" is that the "User DSN" can only be used by the current user and the "System DSN" can be used by all the users of the system.

   If you create a "User DSN", do so with the same user name you run the application that will connect to Denodo.

3. Select the **DenodoODBC Unicode** driver (*not* DenodoODBC ANSI) and click **Finish**.

4. In the configuration dialog fill in the following information:

   a. **Database**: database in Virtual DataPort. E.g. `admin`.
   If the name of the database contains non-ASCII characters, they have to be URL-encoded. For example, if the name of the database is "テスト", enter "%E3%83%86%E3%82%B9%E3%83%88".

   b. **Server** and **Port**: host name and port of the server that runs Virtual DataPort. The default ODBC port is 9996.

   c. **User Name** and **Password**: credentials of a Virtual DataPort user.

   d. If SSL is enabled on the Virtual DataPort server, in the **SSL Mode** list, select **require**.

   E.g.:

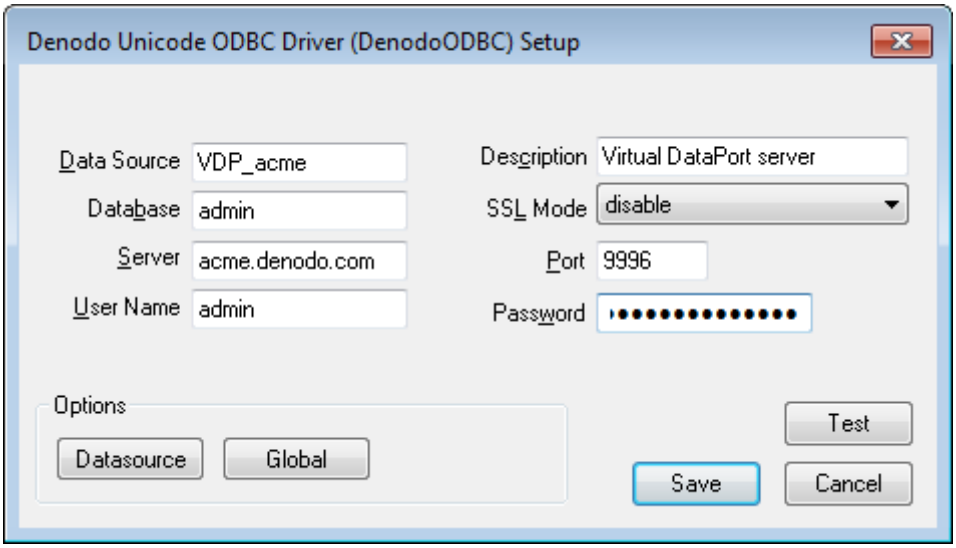

Figure 5 Denodo ODBC driver: configuration dialog

5. Click **Datasource** to open the **Advanced options** dialog (see Figure 6):

   a. Clear **KSQO (Keyset Query Optimization)**

   b. Select **Use Declare/Fetch**

c. Clear **CommLog (C:\psqlodbc_xxx.log)**. If selected, the driver logs all the requests received by this DSN to a file in C:\ whose name starts with `psqlodbc`.
In a production environment, we strongly recommend clearing this check box because logging all the requests impacts the performance of the driver and the log file may grow to a very large size.

d. Clear **MyLog (C:\mylog_xxx.log)**

e. In "Unknown sizes", select **Maximum**. See more about what this means in the section 3.3.3.

f. Clear **Bools as Char**

g. As "Use Declare/Fetch" is selected, the DSN will use DECLARE CURSOR/FETCH to handle SELECT statements. The effect is that the DSN will retrieve the rows of the result set in blocks, instead of retrieving them all at once. **Cache Size** establishes the number of rows of each block.

The "Cache size" of the DSN is equivalent to the "Fetch size" of the JDBC connections.

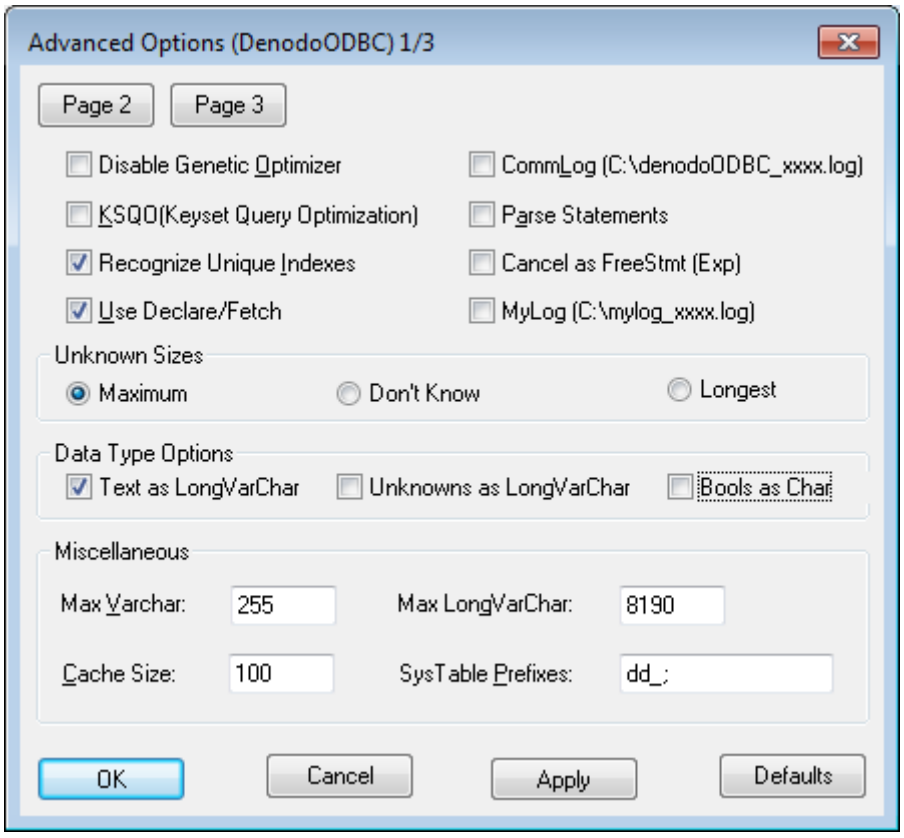

Figure 6 PostgreSQL ODBC driver: advanced configuration (Page 1)

6. Click on **Page 2** (see Figure 7):

a. Select **Server side prepare**

b. In the area "Level of rollback on errors", select **Transaction**

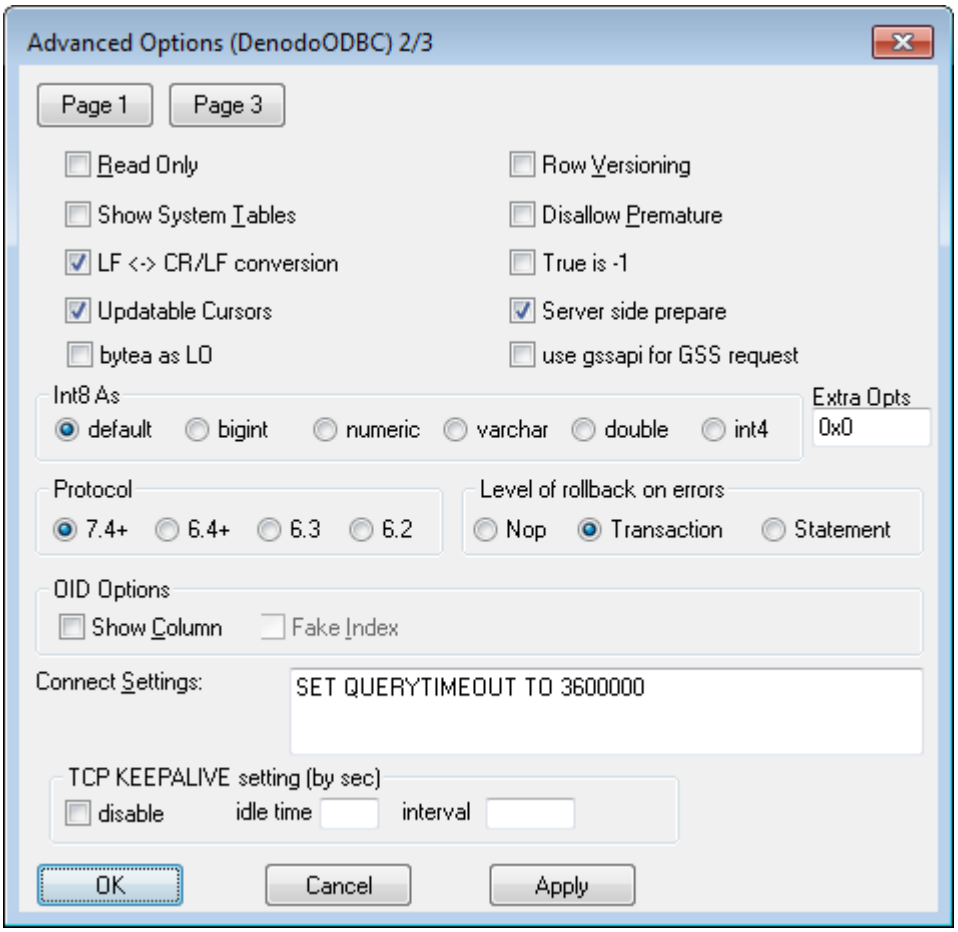

Figure 7 Denodo ODBC driver: advanced configuration (Page 2)

7. In the **Connect settings** box you can change the properties of the ODBC connection established with Virtual DataPort, by entering the following:

   a. `SET QUERYTIMEOUT TO <value>` to change the query time out (value in milliseconds).

   b. `SET i18n TO <i18n>` to change the i18n of the connection.

   For example, to set the default timeout of the queries to one hour, add the following:

   ```
   SET QUERYTIMEOUT TO 3600000;
   SET I18N TO us_pst;
   ```

   Note the ";" between each statement.

   The following table describes these properties and lists its default values:

Connection Property	Description	Default Value
Query timeout	Maximum time (in milliseconds) the driver will wait for a query to finish. After this period, it will throw an Exception. If 0, the driver will wait indefinitely until the query finishes. This parameter sets the default timeout for all the queries. In addition, you can change the timeout for a single query by adding the parameter `'QUERYTIMEOUT'='<value>'` to the `CONTEXT` clause of the query. See more about this in the section "CONTEXT Clause" of the Advanced VQL Guide.	900000 milliseconds (15 minutes)
i18n	Sets the internationalization (i18n) configuration of the connection with the Server. If not present, the driver assumes the i18n of the database that you are connecting to. The "date" fields of the queries' results are converted to the time zone assigned to the i18n of the connection. The parameter `i18n` in the `CONTEXT` clause of the queries overrides the value of this parameter.	<I18N of the database that you are connecting to>

Table 3 Parameters of the ODBC driver and their default value

c. Add the following to the "Connect settings" box to connect to Virtual DataPort using Kerberos authentication:

```
/*krbsrvname=HTTP*/
```

Important: This line has to be *the last line* on the "Connect settings" box.

To be able to use Kerberos authentication, the configuration of the DSN has to meet these conditions:

1. The database of Virtual DataPort that the DSN will connect to is configured with the option "ODBC/ADDO.net authentication type" set to "Kerberos".
2. The client has to belong to the Windows domain. The reasons is that the ODBC driver uses the ticket cache of the operating system to obtain "ticket-granting ticket" (TGT).

8. Click **Page 3** and set the option "The use of LIBPQ library" to **No**.
9. Click **Ok** to close the "Advanced Options" dialog.
10. Click on **Test** to test the connection to Virtual DataPort.
11. Click **Ok**.

The DSN is now configured and ready to be used.

After setting up the DSN, we recommend reading the section 3.3, specially the section "Increasing the Performance of the Denodo ODBC Driver".

3.2 CONFIGURATION OF THE ODBC DRIVER IN LINUX AND OTHER UNIX

The Denodo Platform provides an ODBC driver for Linux, which is based on the ODBC PostgreSQL driver.

As any other ODBC drivers, you have to install it in the machine where the client application runs.

3.2.1 Install the ODBC Driver

To install the ODBC driver on Linux/Unix you have to compile its source code.

To do this follow these steps:

1. Execute the following commands to extract the source code and compile it:

```
cd <DENODO_HOME>
cd tools/client-drivers/odbc
tar -zxf DenodoODBC_src.tar.gz
cd denodoodbc-09.03.0400
./configure
make
```

2. Execute the following command to install the driver:

```
sudo make install
```

The installation is now complete.

3.2.2 Set up a DSN on Linux and Other UNIX

Linux does not provide an ODBC driver manager, so you have to compile one and install it.

This section explains how to install and configure unixODBC [UX_ODBC]. We can install a binary package of unixODBC (i.e. in Fedora we can use `yum` to install it) or download the source code and compile it.

3.2.2.1 Install unixODBC

Follow these steps to download and compile the source code of unixODBC:

1. Download the latest version of the source code from *http://www.unixodbc.org/download.html*

2. Execute the following commands to extract the source code and compile it:

```
tar -zxf unixODBC*.tar.gz
cd unixODBC*
./configure.sh
make
```

3. Execute the following command:

```
sudo make install
```

3.2.2.2 Register the PostgreSQL ODBC Driver in unixODBC

Follow these steps to register the PostgreSQL ODBC driver in the unixODBC driver manager:

1. Create a new file `postgreSQLDriver.template` with the content below:

```
[DenodoODBCDriver]
Description=ODBC driver for Denodo 6.0
Driver=/usr/lib/denodoodbc.so
UsageCount=1
```

   Make sure that the file of the `Driver` property exists.

2. Execute the following command to register the PostgreSQL driver in the ODBC Driver Manager:

```
sudo odbcinst -install -driver -file postgreSQLDriver.template
```

To list the ODBC drivers registered in the driver manager, execute this:

```
sudo odbcinst -query -driver
```

The result should list the new driver: `postgreSQLDriver`.

To uninstall the driver, execute:

```
sudo odbcinst -uninstall -driver -name postgreSQLDriver
```

3.2.2.3 Configure the DSN in unixODBC

To register a new DSN in unixODBC, follow these steps:

1. Create a file called `denodoDSN.template` with the content below:

```
[VDP_acme_DSN]
Description         = VDP connection
Driver              = DenodoODBCDriver
Servername          = <host name>
Port                = <Virtual DataPort ODBC port. Default is 9996>
```

```
UserName            = <Virtual DataPort user name>
Password            = <Password>
Database            = <Virtual DataPort database>
Protocol            = 7.4
BoolsAsChar         = False
ByteaAsLongVarBinary= 1
ConnSettings       = SET QUERYTIMEOUT TO 3600000; SET I18N TO us_pst;
Debug               = No
DebugFile           = ~/unixODBC/log/debug.log
FakeOidIndex        = No
Fetch               = 1000
Ksqo                = 0
LFConversion       = Yes
Optimizer           = 0
ReadOnly            = No
RowVersioning       = No
ServerType          = Postgres
ShowOidColumn       = No
ShowSystemTables    = No
# Uncomment the "Sslmode" property if SSL is enabled in the
# Virtual DataPort Server
# Sslmode           = require
Trace               = No
TraceFile           = ~/unixODBC/log/trace.log
UniqueIndex         = Yes
UpdatableCursors    = Yes
UseDeclareFetch     = 1
UseServerSidePrepare= 1
```

Figure 8 PostgreSQL ODBC driver: default configuration for Linux

If the name of the database contains non-ASCII characters, they have to be URL-encoded. For example, if the name of the database is "テスト", set the property `Database` to `%E3%83%86%E3%82%B9%E3%83%88`.

As the property `UseDeclareFetch` is enabled, the DSN will use DECLARE CURSOR/FETCH to handle SELECT statements. The effect is that the DSN will retrieve the rows of the result set in blocks, instead of retrieving them all at once. The `Fetch` property establishes the number of rows of each block. This property is equivalent to the "Fetch size" of the JDBC connections.

If you set the property `Debug` to `Yes`, the driver logs all the requests received by this DSN to the file set in the property `DebugFile`.
In a production environment, we strongly recommend setting the value of this property to `No` because logging all the requests impacts the performance of the driver and the log file may grow to a very large size.

In the `ConnSettings` property, you can set the properties of the connection established with Virtual DataPort, by adding the following statements:

a. `SET QUERYTIMEOUT TO <value>` to change the query time out (value in milliseconds).

b. `SET i18n TO <i18n>` to change the i18n of the connection.

For example, to set the default timeout of the queries to one hour, set the value of the property `ConnSettings` to the following:

```
ConnSettings=SET QUERYTIMEOUT TO 3600000; SET I18N TO us_pst
```

Note the ";" between each statement.

Read Table 3 to learn how these properties work, and their default value.

If you have enabled SSL in the Virtual DataPort server to secure the communications, add the following property to this configuration file:

```
Sslmode=require
```

c. Add the following `ConnSettings` property to connect to Virtual DataPort using Kerberos authentication:

```
/*krbsrvname=HTTP*/
```

Important: This line has to be the last thing on the `ConnSettings` property.

To be able to use Kerberos authentication, the configuration of the DSN has to meet these conditions:

1. The DSN uses the version 09.03.0400 or newer of the ODBC driver.
2. The database that the DSN will connect to is configured with the option "ODBC/ADDO.net authentication type" set to "Kerberos".
3. The client has to belong to the Windows domain. The reasons is that the ODBC driver uses the ticket cache of the operating system to obtain "ticket-granting ticket" (TGT).

2. Execute this to register the new DSN:

```
odbcinst -install -s -l -f denodoDSN.template
```

The parameter `-l` registers the DSN as a "system DSN". "System DSNs" are available to all the users.

If you do not have enough privileges to register a "system DSN", replace `-l` with `-h` to register the DSN as a "user DSN" instead. If you do this, execute this command with the same user name that you execute the client application that needs to access to this DSN. The reason is that "user DSNs" are only available to the user that registers them.

To list the DSNs registered in the ODBC driver manager, execute this:

```
odbcinst -query -s
```

The result should list the new DSN: `denodo_acme_DSN`.

To delete a DSN from the driver manager, execute this:

```
odbcinst -uninstall -s -name denodo_acme_DSN
```

After setting up the DSN, we recommend reading the section 3.3, specially the section "Increasing the Performance of the Denodo ODBC Driver".

3.3 INTEGRATION WITH THIRD-PARTY APPLICATIONS

The following subsections describe certain issues that you may run into when connecting to the ODBC interface of Denodo, from a third-party application.

3.3.1 Increasing the Performance of the Denodo ODBC Driver

To increase the speed at which the Denodo ODBC driver returns numeric values, do the following:

1. Log in to the Virtual DataPort Administration Tool as an administrator user.
2. From the VQL Shell, execute this:

```
SET
'com.denodo.vdb.vdbinterface.server.odbc.forceBinaryTypes'='true';
```

Important: only set this property if *all* the clients that connect to Denodo through the ODBC interface use the Denodo ODBC driver for Windows. The connections opened from clients that use a different ODBC driver will crash.

3.3.2 Supporting Queries with Brackets

Some applications such as Microsoft Power Pivot, in the queries sent to Denodo, surround the schema and the name of the view with brackets (i.e. `[` and `]`) instead of double quotes. For example, they send a query like this one:

```
SELECT [customer_360].[customer].* FROM [customer_360].[customer]
```

Instead of sending one like this one:

```
SELECT "customer_360"."customer".* FROM "customer_360"."customer"
```

To configure a DSN to allow brackets instead of the double quotes to surround names of schemas and views, add the following to the connection settings of the DSN:

```
SET identifierdelimiter=brackets;
```

To configure this on Windows, open the configuration dialog of the DSN, click **Data source** and then, **Page 2**. Add this to the **Connect settings** box.

To configure this on Linux, add this to the property `ConnSettings` property of the file used to register the DSN, delete the DSN from the driver manager and add it again.

The sections 3.1.2 and 3.2.2.2 explain in detail how to configure a DSN on Windows and Linux respectively.

3.3.3 Maximum Length of Text Values

When an application executes a query through the ODBC interface of Virtual DataPort, this interface provides metadata about every field of the result set of the query. For

fields of type `text`, it reports among other things, the maximum length of the values of this field.

When a `text` field has its "Type size" defined in its "Source type properties", the ODBC interface reports this value. When the type size is not defined, the ODBC interface reports that the maximum size of the values of this field is unknown. In this case, as we configured the DSN with the option "Unknown size" = "Maximum" ("Page 1" dialog of the DSN configuration), the DSN will report that the maximum length of the field is the value specified in the "Max Varchar" property of the DSN.

If the length of a `text` value, whose field does not have its "Type size" defined, is longer than the "Max Varchar", the application that executes the query may do one of the following things:

- Leave the value as is.
- Truncate the value to the "Max Varchar" size.
- Set the value to NULL.

This behavior changes from application to application.

See how to set the "Source type properties" of a Virtual DataPort view in the section "Viewing the Schema of a Base View" of the Administration Guide.

3.3.4 Maximum Length of Error Messages

There are applications that fail when the length of an error message exceeds a certain length. To work around this problem, Virtual DataPort provides options to set a limit on the length of the error messages. To do this, you have two options:

1. Configure the ODBC interface of the Server to limit the length of the error messages.

   To do this, execute this command from the VQL Shell using an administrator account:

```
SET 'com.denodo.vdb.vdbinterface.server.odbc.errorMaxLength'='200';
```

   The statement above sets the limit to 200 characters.

   This change affects *all* the ODBC clients.

2. Configure an individual connection:

   a. For connections established using the Denodo ODBC driver, add the parameter `errorMaxLength` to the "Connect settings" of the DSN. For example:

```
SET ErrorMaxLength TO 200;
SET QueryTimeout TO 3600000;
```

   b. For connections established using the ADO.Net provider (see section 4), add the parameter `errorMaxLength` after the name of the database. For example:

```
support?errorMaxLength=200
```

The option set in the connection (option 2) overrides the option set in the Server (option 1). For example, if you configure the ODBC interface to limit the length of the error messages to 150 and in the DSN, to 100, the limit will be 100.

4 ACCESS THROUGH AN ADO.NET DATA PROVIDER

ADO.Net data providers are software components that allow their users to develop applications that are independent of the database they want to use.

Virtual DataPort is compatible with the Npgsql ADO.Net provider for PostgreSQL [NPGSQL] version 2.0. The recommended versions are 2.0.12, 2.2.0 and 2.2.3, which can be downloaded from *http://npgsql.projects.postgresql.org*.

From your application, you can do the following:

1. Create a new object of the class `NpgsqlConnection`, passing the connection string to the constructor. This is what the example of `DENODO_HOME\samples\vdp\vdp-clients-ADO.NET\Program.cs` does.

2. Or, define the ADO.Net provider in the global `machine.config` file or in the `.config` file of the application and from your application, request a connection to the Npgsql factory and set the appropriate connection string. This option allows you write code that is independent of database you are using.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <add name="Npgsql Data Provider" invariant="Npgsql"
support="FF"
           description="ADO.Net Data Provider to Denodo"
           type="Npgsql.NpgsqlFactory, Npgsql, Version=2.0.12.0,
Culture=neutral, PublicKeyToken=5d8b90d52f46fda7" />
     </DbProviderFactories>
  </system.data>
</configuration>
```

Figure 9 Sample `app.config` file with the provider definition

Example of connection string to Virtual DataPort:

```
string connectionString = "Server=acme;" +
    "Port=9996;" +
    "Username=admin;" +
    "Password=admin;" +
    "Database=admin" +
    "CommandTimeout=80000";
```

Figure 10 Sample `ConnectionString` to connect to Virtual DataPort

If the name of the database contains non-ASCII characters, they have to be URL-encoded. For example, if the name of the database is "テスト", set the property `Database` to `%E3%83%86%E3%82%B9%E3%83%88`.

The default query timeout of the connection is established in the `CommandTimeout` parameter (time in milliseconds). In this connection, the timeout will be 80 seconds.

The value of the i18n of the connection is set in the `Database` parameter of the connection string. The Table 3 describes this property and its default value.

If SSL was enabled in the Virtual DataPort server to secure the communications, add the following parameters to `ConnectionString`:

```
"SSL=True;Sslmode=Require"
```

The page *http://npgsql.projects.postgresql.org/docs/manual/UserManual.html#section3* lists the parameters of the `ConnectionString`.

4.1 USING KERBEROS AUTHENTICATION

To develop an application that logs into Virtual DataPort using Kerberos authentication, do the following:

1. Add the parameter "`krbsrvname=HTTP`" to the connection string.

2. In the "Server" property of the connection string, put the Server principal name (SPN) assigned to the Virtual DataPort server without the prefix "HTTP/".

   To find the SPN, open the administration tool, go to the "Kerberos configuration" dialog and copy it from the "Server principal" box.

   Let us say that SPN is `HTTP/host1.subnet1.contoso.com@CONTOSO.COM`, the value of the property "Server" in the connection string has to be `host1.subnet1.contoso.com@CONTOSO.COM`. I.e., the same without "HTTP/".

For example,

```
string connectionString =
    "Server=host1.subnet1.contoso.com@CONTOSO.COM;" +
    "Port=9996;" +
    "Database=admin" +
    "CommandTimeout=80000";
```

As we are using Kerberos authentication, we do not need to provide the properties "Username" nor "Password" in the connection string.

To use Kerberos authentication, these condition have to be met:

1. The application has to use the version 2.2.7 version of the Npgsql provider. Earlier versions do not support Kerberos authentication.

2. The Virtual DataPort database to which the application connects has to be configured with the option "ODBC/ADDO.net authentication type" set to "Kerberos".

3. The host where the application runs has to belong to the Windows domain. The reasons is that the adapter uses the ticket cache of the operating system to obtain "ticket-granting ticket" (TGT)

5 ACCESS THROUGH OLE DB

OLE DB (Object Linking and Embedding, Database) is an API designed by Microsoft that allows accessing data from a variety of sources in a uniform manner.

The Denodo Platform does not include an OLE DB adapter but our partner Intellisoft [INTELLISOFT] provides one. Please contact them for further information.

6 DEVELOPING EXTENSIONS

Denodo4E, an Eclipse plug-in which provides tools for creating, debugging and deploying Denodo extensions, including Custom Functions, Stored Procedures and Custom Wrappers, is included in the Denodo Platform.

We strongly recommend using this plugin to develop extensions: custom functions, stored procedures, custom wrappers and custom policies.

Read the `README` file in `<DENODO_HOME>/tools/denodo4e` for more information.

6.1 DEVELOPING CUSTOM FUNCTIONS

Custom functions allow users to extend the set of functions available in Virtual DataPort. Custom functions are implemented as Java classes included in a Jar file that is added to Virtual DataPort (see section "Importing Extensions" of the Administration Guide). These custom functions can be used in the same way as every other function like `MAX`, `MIN`, `SUM`, etc.

Virtual DataPort allows creating condition and aggregation custom functions. Each function must be in a different Java class, but it is possible to group them together in a single Jar file.

In the Virtual DataPort installation (in the directory `<DENODO_HOME>/samples/vdp/customFunctions`), there are examples of custom functions. The `README` file of this directory explains how to compile and use these examples.

We strongly recommend developing custom functions using Java annotations (see section 6.1.1). Although it is also possible to develop them following certain name conventions (see section 6.1.2), your custom function will not have access to all the features provided by the Denodo Platform.

These are the rules that every custom function must follow to work properly:

- Functions with the same name are not allowed. If a Jar contains one or more functions with the same name, the Server will not load anything from that Jar.
- All custom functions stored in the same Jar are added or removed together by uploading/removing the Jar in the Server.
- Each function can have many signatures. Each signature represents a different method in the Java class that defines the custom function.
- Functions can have arity n but only the last parameter of the signature can be repeated n times.
- Functions have to be stateless. That is, they should not store any data between executions. E.g., do not use global variables.
  If a custom function is implemented as stateful, it may not work properly in certain scenarios.

Custom functions signatures that return compound type values (register or array) need an additional method to compute the structure of the return type. This way

Virtual DataPort knows in advance the output schema of the query. This method is also needed if the output type depends on the input values of the custom function.

When defining custom functions simple types are mapped directly from Java objects to Virtual DataPort data objects. The following table shows how the mapping works and which Java types can be used:

Java	Virtual DataPort
`java.lang.Integer`	`int`
`java.lang.Long`	`long`
`java.lang.Float`	`float`
`java.lang.Double`	`double`
`java.math.BigDecimal`	`decimal`
`java.lang.Boolean`	`boolean`
`java.lang.String`	`text`
`java.util.Calendar`	`date`
`byte[]`	`blob`

Table 4 Equivalency between Java and Virtual DataPort data types

The parameters of a custom functions cannot be primitives: `int`, `long`, `double`, etc.

The last parameter of the function can be a "varargs" argument. For example:

```
function1(Integer parameter1, String... parameterN)
```

In Virtual DataPort, this function will have a variable number of arguments.

To use custom functions that rely on external jars, do the following:

- Copy the required jars to the directory `<DENODO_HOME>/extensions/thirdparty/lib`.
- Or, copy the contents of the required jars into the jar that contains the custom function. We have to copy the contents of the required jars, not the jars themselves.

6.1.1 Creating Custom Functions with Annotations

A Custom function created with annotations is a Java class with several annotations, which indicate Virtual DataPort that:

1. The Java class contains the code of a custom function.
2. Which method(s) contain the code that Virtual DataPort will have to run when the custom function is invoked.

Each Java class has to contain *one and only one* custom function, which may have one or more signatures. For example, in the same class you can define the function `function1` with the signatures `function1(int)`, `function1(int, text)`, etc.

To develop custom functions, add the library `<DENODO_HOME>/lib/contrib/denodo-commons-custom.jar` to the classpath of your development environment.

Then, follow these steps:

1. Create a Java class and annotate it with `@CustomElement` (package `com.denodo.common.custom.annotations`), which has the following parameters:
   - `name`. Name of the function.
   - `type`. Type of the function. Its value can be either:
     - `CustomElementType.VDPFUNCTION`: if the function is scalar.
     - Or, `CustomElementType.VDPAGGREGATEFUNCTION`: if this is an aggregation function.
2. Add a method for each signature that you want the function to have.

   For example, to develop the custom function `function1` with the signatures `function1(int)`, `function1(int, text)`, add two methods:

   a.
   ```
   @CustomExecutor
   public Integer method1(Integer i) { … }
   ```

   b.
   ```
   @CustomExecutor
   public Integer method2(Integer i, String s) { … }
   ```

   The type of the method parameters has to be a basic Java type (i.e. `String`, `Integer`, `Long`, `Float`, etc.). A parameter cannot have a primitive type.

   The methods that represent a signature of the function have to have the annotation `@CustomExecutor` (package `com.denodo.common.custom.annotations`).

   At runtime, the Server will run the appropriate method depending on the parameters passed to the function. For example, if a query invokes the function `function1(int)`, the Server will run the code of the first method. If a query invokes the function `function1(int, text)`, the Server will run the code of the second method.

   The class can have any number of methods, but it has to have at least one per signature. In addition, these methods have to be in the same class, but the custom function can invoke code of other classes.
3. Optionally, you can add the parameter `syntax` to the `@CustomExecutor` annotations. The Administration Tool will use the value of this parameter when displaying each signature of the custom function to the user (e.g. in the auto-completion feature of the expressions editor).

The value of the `syntax` parameter takes preference over the value of the `syntax` parameter of the `@CustomParam` annotations (see below). Therefore, use one, or the other.

4. If you want this custom function to be pushed-down to a database, add the parameters `delegationPatterns` and `implementation` to the `@CustomExecutor` annotations. The section 6.1.1.1 explains in more detail how to develop this type of functions.

5. In the methods that have the `@CustomExecutor` annotation, you can add the annotation `@CustomParam` with the `syntax` parameter, to each parameter of the method.

   The value of the `syntax` parameter gives a user-friendly name to the parameter of this function's signature when the autocomplete feature of the Administration Tool displays it. If this annotation is not used, the syntax of the method will be displayed as *arg1, args2…*

   The value of this parameter will be ignored if the annotation `@CustomExecutor` of the method has the parameter `syntax`.

   The value of the parameter `mandatory` of the `@CustomParam` annotation is ignored. It is only used when this annotation is used to develop Custom Policies.

6. If you are developing an aggregation function, mark the parameters that represent aggregation fields with the annotation `@CustomGroup`. The type of these parameters has to be `CustomGroupValue`.
   The `groupType` parameter is the type of the elements of the group.
   For example,

```
@CustomExecutor
public String aggregationFunction(
    @CustomGroup(name="field", groupType=String.class)
        CustomGroupValue<String> text_field) {

...
}
```

7. For each method annotated with `@CustomExecutor` that meets at least one of the following conditions, you have to add another method and annotate it with `@CustomExecutorReturnType`:

   - The return type of the function is an `array` or a `register`.
   - Or, the return type of the function depends on the type of the input parameters.

   See the section 6.1.4 for more details about this method.

6.1.1.1 Developing Custom Functions that Can Be Delegated to a Database

This section explains how to develop custom functions that, besides being executable by the Virtual DataPort server, can be delegated to JDBC data sources. That means that when possible, instead of executing the Java code of the custom function, the Server invokes a function of the database.

To do this, you just have to add the following parameters to the annotation(s) `@CustomExecutor` of the method(s) that implement the function:

- `implementation`: if `true`, it means that the code of the function also can return the proper result. The Server will execute this code when the function cannot be delegated to the database.
  If `false`, it means that the code of the custom function is not valid and the Server will never execute it. Therefore, the Server will return an error if it cannot delegate the function to the database.

- `delegationPatterns`: array of `DelegationPattern` annotations that represent the configuration of each database that the function can be delegated to.
  `DelegationPattern` has the following attributes:

  - `databaseName`: the name of the database that support this function.

    This value corresponds with the value of the parameter `DATABASENAME` of the `CREATE DATASOURCE JDBC` statement that creates the JDBC data sources that you want to delegate the function to.

  - `databaseVersions` (optional): array of versions of the database that support this function. When this parameter is not present, it means that the function can be delegated to any version of the database indicated in `databaseName`.
    The values of this array correspond with the values of the parameter `DATABASEVERSION` of the `CREATE DATASOURCE JDBC` statement that creates the JDBC data sources that you want to delegate the function to.

  - `pattern`: expression that will be delegated to the database. This parameter is necessary as the function may have a different name and signature in each database.
    This string is some sort of regular expression where `$0` represents the first parameter passed to the custom function, `$1` the second, etc.
    If a parameter has a variable number of arguments ("varargs"), you can use a pattern such as `$0[, $i]{1, n}`.

    For example, if the signature of the function is `f1(Integer I, String… param)`, the value of pattern could be like:
    `pattern="FUNCTION_IN_DB($0, $1[, $i]{2, n})"`.

    The *example 2* below shows how to define the pattern when one of the parameters has a variable number of arguments.

    In the `pattern` parameter, you can only use the "`[`" character to indicate a variable number of arguments (e.g. `$0[, $i]{1, n}`). This character cannot be used as a literal.

Note: you cannot develop custom functions that are delegable to a database using name conventions (described in section 6.1.2). You have to do it with annotations.

Example 1

Let us say that we have developed a custom function called `MAX_VALUE` that returns the maximum number of three numbers; that Microsoft SQL server has a function

called `MAXIMUM_N` that calculates the same and that Oracle has the same function, but is called `TOP_N` in the versions 10g, and 11g but not in the previous versions[1].

By adding a few parameters to the annotation `@CustomExecutor`, Virtual DataPort will delegate the execution of this function to Oracle 10g and 11g and to any version of SQL Server, whenever is possible.

```
@CustomElement(type = CustomElementType.VDPFUNCTION, name =
"MAX_VALUE")
public class CustomFunctionMaxNumber {
    @CustomExecutor(implementation = true, delegationPatterns = {
            @DelegationPattern(databaseName = "sqlserver",
                               pattern = "MAXIMUM_N($0, $1, $2)"),
            @DelegationPattern(databaseName = "oracle",
                               databaseVersions = { "10g", "11g" },
                               pattern = "TOP_N($0, $1, $2)") })
    public Double Max(
            @CustomParam(name = "arg0") Double arg0,
            @CustomParam(name = "arg1") Double arg1,
            @CustomParam(name = "arg2") Double arg2) {

        ....
     }
}
```

Figure 11 Example of how to annotate a custom function so it can be delegated to a database

The first `@DelegationPattern` annotation indicates that when the Server can delegate the function to SQL Server (any version), it will delegate it as the function `MAXIMUM_3`.

The second `@DelegationPattern` indicates that when the Server can delegate the function to the versions 10g and 11g of Oracle (the function cannot be delegated to other versions), it will delegate it as the function `TOP_3`.

Example 2

Let us say that we want do develop the same function but with a variable number of arguments. In this case, you have to define the parameter as "varargs" (note the `...` after the type of the parameter).

```
@CustomElement(type = CustomElementType.VDPFUNCTION, name =
"MAX_VALUE")
public class CustomFunctionMaxNumber {
    @CustomExecutor(implementation = true, delegationPatterns = {
            @DelegationPattern(databaseName = "sqlserver",
                               pattern = "MAXIMUM_N($0[, $i]{1,
n})"),
            @DelegationPattern(databaseName = "oracle",
                               databaseVersions = { "10g", "11g" },
                               pattern = "TOP_N($0[, $i]{1, n})") })
    public Double Max(
            @CustomParam(name = "values") Double... arg0) {
```

[1] SQL Server and Oracle do not have these functions. We made them up for the sake of the example.

```
            ....
        }
}
```

Figure 12 Example of how to annotate a custom function so it can be delegated to a database (2)

By adding "`...`" to the type of the parameter, the function admits one or more values. The `pattern` parameter, which defines how the function is delegated to the database, is `$0[, $i]{1, n}`. This means that if you pass the value "`2`" to the function, the Server will delegate `TOP_N(2)` to Oracle. If you pass the parameters "`2, 3, 4`", the Server will delegate `TOP_3(2, 3, 4)` to Oracle.

6.1.2 Creating Custom Functions Using Name Conventions

We recommend developing custom functions using annotations. However, it is also possible to do it following certain conventions for the name of the class and its methods.

In order to make a Java class recognizable as a custom function, the name of the class has to match the following rules:

- `<FunctionName>VdpFunction` for condition functions
- `<FunctionName>VdpAggregateFunction` for aggregation functions

Note: These conventions are case sensitive. That means that the name of the class has to be like `function1VdpFunction` and not `function1VDPFUNCTION`.

This way a Java class named `Concat_SampleVdpFunction` will be interpreted as a condition function named `Concat_Sample`; and a class named `Group_Concat_SampleVdpAggregateFunction`, as an aggregate function named `Group_Concat_Sample`.

All Java methods implementing the function signatures must have the name `execute`. The signature associated with each method will be extracted from its method parameters.

For example a class named `Concat_SampleVdpFunction` with a method `execute(valueA:String, valueB:String):String` will generate the function signature `CONCAT_SAMPLE(arg1:text, arg2:text)`.

The way to define an arity n in a custom function is with an array as the last parameter in the method. I.e. a class `Concat_SampleVdpFunction` with a method declared as `public String execute(String [] inputs)`.

A custom function has to define a method named `executeReturnType` with the same parameters as the associated `execute` method if:

- The return type of the function is an array or a register.
- Or, the return type of the function depends on the type of the input parameters.

See the section 6.1.4 for more details about this method.

6.1.3 Compound Types

In custom functions, compound types and compound values are represented with the following Java classes:

- Registers:
  - `com.denodo.common.custom.elements.CustomRecordType` represents the type of a register field (not a value of the register).
    Its method `getProperties()` returns a collection of name-type pairs. Each element of the collection holds the type of one of the fields of the register. The class of the `Object` returned by the method `getType()` of the interface `CustomRecordType.Property` depends on the type of the field:
    i. If the type of the field is basic, the method returns a `java.lang.Class`: `Long.class`, `Integer.class`, `String.class`, etc.
    ii. If the type of the field is a register, the method returns a `CustomRecordType` object.
    iii. If the type of the field is an array, the method returns a `CustomArrayType` object.
  - `com.denodo.common.custom.elements.CustomRecordValue` represents the value of register field.
    Its method `getProperties()` returns a collection of name-value pairs of the register. Each element of the collection holds the value of one of the fields of the register. The class of the `Object` returned by the method `getValue()` of the interface `CustomRecordValue.Property` depends on the type of the field:
    i. If the type of the value is basic, the method returns a basic Java object: `java.lang.String`, `java.lang.Integer`, `java.lang.String`, etc.
    ii. If the type of the field is a register, the method returns a `CustomRecordValue` object.
    iii. If the type of the field is an array, the method returns a `CustomArrayValue` object.
- Arrays:
  - `com.denodo.common.custom.elements.CustomArrayType` represents the data type of an array field (not a value of the array).
    It holds the name of the type and an instance of `CustomRecordType` with the type of the elements of the array (an array is always an array of registers)
  - `com.denodo.common.custom.elements.CustomArrayValue` represents the values of an array. It holds a list of `CustomRecordValue` objects.
- `com.denodo.common.custom.elements.CustomGroupValue` represents the list of values coming from a non-aggregation field in an aggregation function.

The class `CustomElementsUtil` provides methods to create array and register types and values.

6.1.4 Custom Function Return Type

The custom functions whose return type is an `array` or a `register`, or whose return type depends on the input values, have to implement an additional method that returns the type of the function based on the parameters of the function.

If the function has several signatures that meet one of these conditions, there must be an additional method for each signature.

In the custom functions developed with Java annotations, this additional method has to be annotated with `CustomExecutorReturnType`. If it is developed with name conventions, the name of the method has to be `executeReturnType`.

This additional method has to meet these rules:

1. It must have the same number of parameters as the "execute" method.
2. Each parameter of the additional method must have the same type or an equivalent one, as its respective parameter in the `execute` method:
    a. If `execute` returns a basic Java type, the additional method has to return the same basic Java class. For example, if `execute` returns a **`String`** object, the additional method has to return **`java.lang.String.class`**.
    b. If `execute` returns a **`CustomRecordValue`** object, the additional method has to return a **`CustomRecordType`** object.
    c. If `execute` returns a **`CustomArrayValue`** object, the additional method has to return a **`CustomArrayType`** object.

    See Table 4 to know the type that these return parameters must have in Virtual DataPort.
    d. If a parameter of `execute` is a **`CustomRecordValue`**, the type of the parameter in the additional method has to be **`CustomRecordType`**.
    e. If a parameter of `execute` is a **`CustomArrayValue`**, the type of the parameter in the additional method has to be **`CustomArrayType`**.
3. If the returned type is a compound data type, the type will be created in Virtual DataPort, unless it already exists. If the returned type does not have name, the type will be created with a random name.

At runtime, every time this function is invoked, Virtual DataPort will execute this additional method to know the return type of the function.

The following two sections contain an example of how to implement this additional method in a function that uses annotations and in a function that uses name conventions.

6.1.4.1 Function without Annotations with Return Type Depending on the Input.

Implementation of a function `SPLIT` which splits strings around matches of a given regular expression and returns the array of those substrings:

```
public class SplitVdpFunction {
    private static final String STRING_FIELD = "string";
    public CustomArrayValue execute(String regex, String value) {
        if (value == null || regex == null) {
            return null;
        }
        String[] result = value.split(regex);
        LinkedHashMap<String, Object> results =
                new LinkedHashMap<String, Object>(1);
        List<CustomRecordValue> arrayValues =
                new ArrayList<CustomRecordValue>(result.length);
        for (String string : result) {
            results.put(STRING_FIELD, string);
            CustomRecordValue recordValue =

CustomElementsUtil.createCustomRecordValue(results);
            arrayValues.add(recordValue);
        }
        return
CustomElementsUtil.createCustomArrayValue(arrayValues);
    }

    public CustomArrayType executeReturnType(String regex, String
value){
        LinkedHashMap<String, Object> props =
                new LinkedHashMap<String, Object>();
        props.put(STRING_FIELD, String.class);
        CustomRecordType record =
                CustomElementsUtil.createCustomRecordType(props);
        CustomArrayType array =
                CustomElementsUtil.createCustomArrayType(record);
        return array;
    }
}
```

Figure 13 Example of function without annotations with return type depending on the input

6.1.4.2 Aggregation Function Using Annotations

The following function returns the first value of a non group-by field for each group:

```
@CustomElement(
      type=CustomElementType.VDPAGGREGATEFUNCTION,
name="FIRST_RECORD")
public class FirstRecordFunction {

    @CustomExecutor
    public CustomRecordValue execute(
            @CustomGroup(groupType=CustomRecordValue.class,
name="records")
                CustomGroupValue<CustomRecordValue> records) {

        if(records == null) {
            return null;
        }
        if(records.size() == 0) {
            return null;
        }

        return records.getValue(0);
    }

    @CustomExecutorReturnType
    public CustomRecordType executeReturnType(
            CustomRecordType recordType) {

        return recordType;
    }
}
```

Figure 14 Example of aggregation function using annotations

```
@CustomElement(
    type=CustomElementType.VDPAGGREGATEFUNCTION,
name="FUNCTION_F1")
public class FirstRecordFunction {

    @CustomExecutor
    public Long execute(
            @CustomGroup(groupType=Long.class, name="values")
                CustomGroupValue<Long> records) {

        ...
        ...
        return ...
    }

    @CustomExecutorReturnType
    public Class executeReturnType(Long values) {

        return Long.class;
    }
}
```

Figure 15 Example of aggregation function using annotations

6.1.5 Getting Information about the Context of the Query

Invoke the method `CustomElementsUtil.getQueryContext()` to obtain an instance of the class `QueryContext`. This class provides methods to obtain:

- The user name that executes the query that uses this function.
- The set of roles granted to this user.
- The Denodo database to which the user is connected when executing the query that uses this function.

Open the Javadoc of the Denodo API to obtain more details about this class. I.e. `<DENODO_HOME>/docs/vdp/api/index.html`.

6.2 DEVELOPING STORED PROCEDURES

Virtual DataPort provides an API to develop custom stored procedures written in Java.

After developing a stored procedure, you have to import it into the Virtual DataPort server. The section "Importing Stored Procedures" of the Administration Guide [ADMIN_GUIDE] explains how to do it.

The Denodo Platform provides examples of stored procedures and their source code. They are located in `DENODO_HOME/samples/vdp/storedProcedures`. The `README` file in this path contains instructions to compile and install them.

We strongly recommend using the Denodo4E plugin for Eclipse, to develop stored procedures. The file `README` in `<DENODO_HOME>/tools/denodo4e` explains how to install this plugin.

The classes and interfaces for developing stored procedures are located in the package `com.denodo.vdb.engine.storedprocedure`

This section describes briefly the use of its main classes. See the Javadoc documentation [VDP_API] for further details on these classes and their methods.

To create a stored procedure, create a new Java class that extends `com.denodo.vdb.engine.storedprocedure.AbstractStoredProcedure`

NOTE: Every time a stored procedure is invoked, the Execution Engine creates an instance of this class. Therefore, this class may have attributes that store the state of the procedure during its execution, such as the number of processed rows if the query processes a result set.

You have to override the following methods:

- `public String getName()`
  This method has to return the name of the stored procedure.
  It cannot return `NULL`.
- `public String getDescription()`
  This method has to return the description of the stored procedure.
  It cannot return `NULL`.
- `public StoredProcedureParameter[] getParameters().`
  This method is invoked once every time the procedure is invoked.

It has to return an array with the input and output parameters of the stored procedure. Each parameter is represented with a `StoredProcedureParameter` object. A `StoredProcedureParameter` object specifies the name, type, direction (input and/or output) and "nullability" (if accepts a `NULL` value or not) of a parameter.
If a parameter is a compound type, an array of `StoredProcedureParameter` objects must be specified to describe its fields.
This method cannot return `NULL`. If does not have input nor output parameters, it has to return an empty array.

Example: the Figure 16 contains a method `getParameters()` of a stored procedure that has the following parameters:

- An input parameter of type `text`
- An output parameter that is an array of registers. These registers have two fields: `field1` (`text`) and `field2` (`int`)

```
public StoredProcedureParameter[] getParameters() {

  return new StoredProcedureParameter[] {
            new StoredProcedureParameter(
                  "parameter1"
                , Types.VARCHAR
                , StoredProcedureParameter.DIRECTION_IN)

          , new StoredProcedureParameter(
                  "compound_field"
                , Types.ARRAY
                , StoredProcedureParameter.DIRECTION_OUT
                , true
                , new StoredProcedureParameter[] {
                     new StoredProcedureParameter(
                           "field1"
                         , Types.VARCHAR
                         , StoredProcedureParameter.DIRECTION_OUT)
                   , new StoredProcedureParameter(
                           "field2"
                         , Types.INTEGER
                         , StoredProcedureParameter.DIRECTION_OUT)
                })
  };
}
```

Figure 16 Definition of the parameters of a stored procedure with compound fields

- `public void doCall(Object[] inputValues)`
  The Execution Engine invokes this method when this procedure is called. If the procedure has to return results, invoke the method `getProcedureResultSet():StoredProcedureResultSet` of the superclass to obtain a reference to the list of rows that this procedure will return. Then invoke the method `addRow(…)` of `StoredProcedureResultSet` for each row you want to return.

  Example: let us say that the procedure has a single output parameter called

`compound_field` as the one defined in Figure 16. The following code snippet builds a row and adds it to the result set:

```
@Override
protected void doCall(Object[] inputValues) throws
            SynchronizeException, StoredProcedureException {
    ...
    ...
    ...
    Object[] row = new Object[1];
    List<Struct> compoundField = new
ArrayList<Struct>(values.size());
    List<String> fieldsNames = Arrays.asList("field1", "field2");
    /*
     * 'values' was generated before
     */
    for (Map.Entry<String, Integer> value : values.entrySet()) {

        List structValues = Arrays.asList(value.getKey()
                                          , value.getValue());
        Struct struct = super.createStruct(fieldsNames,
structValues);
        compoundField.add(struct);
    }

    row[0] = createArray(compoundField, Types.STRUCT);
    getProcedureResultSet().addRow(row);
}
```

Figure 17 Stored procedures: building a row of a compound type

Optionally, you can override the following methods:

- `public void initialize(DatabaseEnvironment environment)`
  The Execution Engine invokes this method once every time a query executes this stored procedure. This method can be overridden to perform initialization tasks.
  The object `DatabaseEnvironment` provides several methods that can be useful for the execution of the procedure. See the Denodo Javadoc [VDP_API] for further details about them:
  - Execute queries with the methods `executeQuery` and `executeUpdate`.

    To cancel all the queries executed from the stored procedure, execute this:

    ```
    ((DatabaseEnvironmentImpl) this.environment).cancelQueries()
    ```
  - To obtain a reference to other stored procedures, invoke `lookupProcedure(…)`.
  - To obtain a reference to functions, invoke `lookupFunction(…)`.
  - To create a new transaction, invoke `createTransaction(…)`.

- To add a stored procedure to the current transaction, invoke `joinTransaction(…)`.
- To write a message to the Server's log, invoke `log(…)`.
- To obtain the value of a Server's property, invoke `getDatabaseProperty(…)`
  The properties that can requested are `CURRENT_USER` (user name of the current user) and `CURRENT_DATABASE` (current database).

You can obtain a reference to `DatabaseEnvironment` from other methods by invoking `super.getEnvironment()`.

- `public boolean stop()`
  The Execution Engine invokes this method when a query involving this stored procedure is cancelled. The class `AbstractStoredProcedure` provides a default implementation of this method that does not do anything and only returns `false`.
  If the tasks executed by this procedure can be cancelled, override this method and cancel them. If this procedure opens any connection to other systems, opens files, etc., close these resources from this method.
  If this method returns `true`, the procedure must guarantee that it will finish after this method is invoked. If the procedure will not finish after invoking this procedure, return `false`.
  If this procedure does not overwrite this method, the Execution Engine will try to interrupt the execution of this procedure and the queries started by it. Therefore, overwriting this method is not mandatory, although recommended.

- `public void prepare()`
  The Execution Engine invokes this method when it is about to begin a transaction involving this procedure.

- `public void commit()`
  The Execution Engine invokes this method to confirm the current transaction.

- `public void rollback()`
  The Execution Engine invokes this method to undo the current transaction.

- `public boolean caseSensitiveParameters()`
  If the name of the input and output parameters defined by the stored procedure are case sensitive, override this method and return `true`.

- `public void log(level, message)`
  Log a message to the Virtual DataPort logging system. The message will be added to the log category with the name of the class of this procedure. I.e. if the class of the procedure is `com.acme.procedure1`, the message is added to the category `com.acme.procedure1`.
  If this log category is enabled, the message will be logged to `<DENODO_HOME>/logs/vdp/vdp.log`.

  To enable a log category, modify the `<DENODO_HOME>/conf/vdp/log4j.xml` or invoke the `LOGCONTROLLER` stored procedure. See more about this in the section "Configuring the Logging Engine" of the Administration Guide.

`AbstractStoredProcedure` provides other useful methods:

- `static java.sql.Struct createStruct(Collection values, int type)` This method creates a *struct* SQL-type object. Invoke this method to create a register of elements. See an example in Figure 17.

- `static java.sql.Array createArray(Collection values, int type)` This method returns an array. Invoke this method to create an array of elements. The elements of the list have to be of the type `java.sql.Struct`. You can create them by invoking the method `createStruct(…)`. See an example in Figure 17.

6.2.1 Required Libraries to Develop Stored Procedures

To develop a stored procedure, add the following jar files to the CLASSPATH of your environment:

- `<DENODO_HOME>/lib/vdp-server-core/denodo-vdp-server.jar`

- `<DENODO_HOME>/lib/contrib/commons-logging.jar`

- `<DENODO_HOME>/lib/contrib/jta.jar`

- `<DENODO_HOME>/lib/contrib/denodo-commons-util.jar`

Note: if the stored procedure relies on external jars, do the following:

- Copy the jars to the directory `<DENODO_HOME>/extensions/thirdparty/lib`

- Or, copy the contents of the required jars into the jar that contains the stored procedure. You have to copy the contents of the required jars, not the jars themselves.

- Or, import the external jars (see section *Importing Extensions* of the Administration Guide [ADMIN_GUIDE]) and when importing the new stored procedure, select the jar with the stored procedure and also the external jars (see section *Importing Stored Procedures* of the Administration Guide)

6.3 DEVELOPING CUSTOM WRAPPERS

Virtual DataPort provides an API to develop custom wrappers to retrieve data from sources that are not supported.

We strongly recommend using the Denodo4E plugin for Eclipse, to develop custom wrappers (see the file `README` in `<DENODO_HOME>/tools/denodo4e`).

To create a new Custom wrapper, also called Custom data source, you have to extend the Java abstract class `AbstractCustomWrapper` (`com.denodo.vdb.engine.customwrapper`). This abstract class provides a default implementation of the interface `CustomWrapper` (`com.denodo.vdb.engine.customwrapper`). You should *not* implement the interface `CustomWrapper`.

The following sections explain how to extend the `AbstractCustomWrapper` class.

Virtual DataPort includes a sample Custom wrapper that retrieves data from a Salesforce account. This example is at `<DENODO_HOME>/samples/vdp/customWrappers`. There is a `README` file in this directory that explains how to compile, install and use this Custom wrapper.

6.3.1 Required Libraries to Develop Custom Wrappers

To develop custom wrappers for Virtual DataPort, add the following `jar` files to the CLASSPATH of your environment:

- `<DENODO_HOME>/lib/vdp-server-core/denodo-vdp-server.jar`
- `<DENODO_HOME>/lib/contrib/commons-lang.jar`
- `<DENODO_HOME>/lib/contrib/commons-logging.jar`
- `<DENODO_HOME>/lib/contrib/denodo-commons-interpolator.jar`
- `<DENODO_HOME>/lib/contrib/denodo-commons-util.jar`

If the custom wrapper that you are going to develop relies on third-party jars, you have to do one of the following steps:

- Copy the required jars to the directory `<DENODO_HOME>/extensions/thirdparty/lib`.
- Or, copy the contents of the required jars into the jar that contains the custom wrapper. You have to copy the contents of the required jars, not the jars themselves.
- Or, import the external jars into Virtual DataPort (see section "Importing Extensions" of the Administration Guide). Then, when you create the new custom data source, select the jar with the custom data source that you have developed *and* the external jars (see section "Custom Sources" of the Administration Guide).

6.3.2 Extending AbstractCustomWrapper

The simplest way to create a new Custom wrapper is to implement the following two methods of the abstract class `com.denodo.vdb.engine.customwrapper.AbstractCustomWrapper`:

- `public CustomWrapperSchemaParameter[] getSchemaParameters(Map<String, String> inputValues)`
  This method has to return the output schema of the Custom wrapper, which is the schema of the data obtained by querying the wrapper.
  You can develop Custom wrappers whose output schema depends on the values of their input parameters. If you want to do this, you have to implement the method `getInputParameters` (see section 6.3.3) to define the input parameters of the wrapper. Then, the parameter `inputValues` will contain the values for these parameters.
  The output schema is represented as an array of

`CustomWrapperSchemaParameters` objects, which represent fields of the schema. A `CustomWrapperSchemaParameter` has a name, a type and several other properties (as mandatoriness, nullability, etc.), and an optional array of other `CustomWrapperSchemaParameters` in case the represented field is compound.

- `public void run( CustomWrapperConditionHolder condition, List<CustomWrapperFieldExpression> projectedFields, CustomWrapperResult result, Map<String, String> inputValues`
  Virtual DataPort invokes this method when a user queries the wrapper. Depending on the wrapper's configuration (see section 6.3.6), the `condition` and `projectedFields` arguments may be taken into account (conditions will be explained in section 6.3.4). These two parameters encapsulate the conditions and the list of projected fields of the query to the wrapper.
  `inputValues` contains the input parameters of the wrapper. It only contains a textual representation of the values and does not contain information about their type. The method `getInputParameterValue(String name)` returns an instance of `CustomWrapperInputParameterValue` that provides full information about the parameter value and its type.

  Usually, an implementation of the method `run` involves analyzing the passed conditions, projected fields and input values, querying the wrapper's data source and returning the retrieved data to Virtual DataPort. This is done by invoking the method `addRow` of the `result` argument. The arguments of `addRow` are an array of `Objects` and a list of projected fields. The array passed to `addRow` must contain a series of `Objects` matching the list of projected fields. Also, the types of the Objects must match the schema defined in the method `getSchemaParameters` (see the Javadoc [VDP_API] for the method `getParameterClass` of the `CustomWrapperSchemaParameter` class to check the appropriate Java class for a `CustomWrapperSchemaParameter` according to its type).

By implementing these two methods, you can create a Custom wrapper. However, in some scenarios you may need to override some methods of the `AbstractCustomWrapper` class to access more advanced features. The next section lists these methods and their default behavior.

There are other useful methods in `AbstractCustomWrapper` like `log(int level, String logMessage)` to log information to the server logging files or `getCustomWrapperPlan()` to access the execution plan and add some information to the trace (see section 6.3.7).

6.3.3 Overriding AbstractCustomWrapper

The following methods may be overridden when extending `AbstractCustomWrapper`:

- `public CustomWrapperInputParameter[] getInputParameters()`.
  This method defines a series of input parameters accepted by the Custom wrapper, represented as an array of objects `CustomWrapperInputParameter`. The default implementation of this method returns an empty array.
  Figure 18 shows an example implementation of this method.
  The `CustomWrapperInputParameter` objects have the following properties:

- `name`: the name of the parameter.
- `mandatory`: if `true`, you have to provide this parameter when querying the wrapper. If `false`, the parameter is optional.
- `description`: description of the parameter. The Administration Tool will show this description as a tooltip in the Custom Data Source wizard.
- `type`: type of the input parameter. To instantiate an object of the class `CustomWrapperInputParameterType` invoke the appropriate method of the `CustomWrapperInputParameterTypeFactory` class. This factory has the following methods:
  - `booleanType(…)` to create a boolean parameter.
  - `integerType()`, `longType()`, `floatType()` and `doubleType()` to create a number parameter.
  - `stringType()`, `longStringType()` to create normal and long text parameters.
  - `enumStringType(…)` to create an enumeration parameter. An input parameter of this type can only have the values of the enumeration. The Administration Tool displays a drop-down list so the user can select a valid value.
  - `hiddenStringType()` to create a text parameter that contains sensitive information that cannot be written to the Virtual DataPort log or displayed in the Administration Tool. The Administration Tool hides the values of this type of parameters.
  - `routeType(…)` to create a parameter that stores a path to a file. The Administration Tool provides a wizard to build valid routes for this type of parameters.
  - `loginType()` and `passwordType()`. If the custom wrapper has an input parameter created with `loginType()` and another one created with `passwordType()`, the Administration Tool allows the user to enable pass-through credentials when creating a base view over the wrapper. In that case, when a user queries the base view, the values of these two parameters will be the credentials of user that executed the query.

```
@Override
    public CustomWrapperInputParameter[] getInputParameters() {
        return new CustomWrapperInputParameter[] {
            new CustomWrapperInputParameter(STRING_PARAM,
                    "A mandatory parameter of type string",
                    true,
                    CustomWrapperInputParameterTypeFactory.
                        stringType()),
            new CustomWrapperInputParameter(BOOLEAN_PARAM,
                    "A mandatory parameter of type boolean with
'false'" +
                    "as the default value",
                    true,
                    CustomWrapperInputParameterTypeFactory.
                        booleanType(false)),
            new CustomWrapperInputParameter(INTEGER_PARAM,
                    "An optional parameter of type integer",
                    false,
                    CustomWrapperInputParameterTypeFactory.
                        integerType()),
            new CustomWrapperInputParameter(ROUTE_PARAM,
                    "An optional parameter of type route",
                    false,
                    CustomWrapperInputParameterTypeFactory.
                        routeType(RouteType.values())) };
    }
```

Figure 18 Example implementation of the method `getInputParameters`

- `public CustomWrapperConfiguration getConfiguration()`
  This method defines the CUSTOM wrapper's configuration (details on how to configure a CUSTOM wrapper are given in section 6.3.6). The default implementation of this method returns an instance of `CustomWrapperConfiguration` with all the available configuration parameters set to their default values.

- `public boolean stop()`
  The Execution Engine invokes this method when a query involving this custom wrapper is cancelled. The class `AbstractCustomWrapper` provides a default implementation of this method that does not do anything and only returns `false`.
  If the tasks executed by this wrapper can be cancelled, override this method and cancel them. If this wrapper opens any connection to other systems, opens files, etc., close these resources from this method.
  If this method returns `true`, the wrapper must guarantee that it will finish after this method is invoked. If the wrapper will not finish after invoking this wrapper, return `false`.
  If this wrapper does not overwrite this method, the Execution Engine will try to interrupt its execution. Therefore, overwriting this method is not mandatory, although recommended.

Custom wrappers can provide support for Insert, Delete and Update operations. By implementing/overriding the appropriate methods, the CUSTOM wrapper will be automatically configured so DataPort knows that it has insert, delete or update capabilities. The following methods may be overridden to provide support for IDU operations:

- `public int insert(`
  `Map<CustomWrapperFieldExpression, Object> insertValues`
  `, Map<String, String> inputValues)`
  `throws CustomWrapperException.`
  This method defines how the CUSTOM wrapper inserts data in its associated data source. The wrapper's input parameters' values are passed as an argument to be used if necessary. The data to be inserted is provided as a map between `CustomWrapperFieldExpressions` and `Objects`. A `CustomWrapperFieldExpression` has a name and an optional list of sub-fields, in case the field is compound. The method `getStringRepresentation` of `CustomWrapperFieldExpression` provides a default text version of a field. That can be just the field's name, or something more elaborate like `myfield.myarray[10].myinteger` (this example represents a compound field `myfield` with an array-type sub-field `myarray` with a sub-field `myinteger`). This method returns the number of successfully inserted values. The default implementation does nothing and returns `0`.

- `public int delete(`
  `CustomWrapperConditionHolder condition`
  `, Map<String, String> inputValues)`
  `throws CustomWrapperException.`
  This method defines how the CUSTOM wrapper deletes data from its associated data source. Deletion conditions and input parameters' values are passed as arguments (see section 6.3.4 for details about dealing with conditions). The number of successfully deleted values is returned. The default implementation of this method does nothing and returns `0`.

- `public int update(`
  `Map<CustomWrapperFieldExpression, Object> updateValues`
  `, CustomWrapperConditionHolder condition`
  `, Map<String, String> inputValues)`
  `throws CustomWrapperException.`
  This method defines how the CUSTOM wrapper updates data in its associated data source. Update conditions, update values and input parameters' values are provided as arguments (see section 6.3.4 for details about dealing with conditions and see the explanation of the insert method in this section for details on update values). The number of successfully updated values is returned. The default implementation of this method does nothing and returns `0`.

CUSTOM wrappers can provide support for distributed transactions. By implementing/overriding the appropriate methods, the CUSTOM wrapper will be automatically configured so DataPort knows that it has transactional capabilities. The following three methods must be overridden for the CUSTOM wrapper to support distributed transactions:

- `public void prepare()`
  This method defines how the CUSTOM wrapper performs a *prepare* operation in the context of a distributed transaction. The default implementation does nothing.

- `public void commit()`
  This method defines how the CUSTOM wrapper performs a *commit* operation in the context of a distributed transaction. The default implementation does nothing.

- `public void rollback()`
  This method defines how the CUSTOM wrapper performs a *rollback* operation in the context of a distributed transaction. The default implementation does nothing.

6.3.4 Dealing with Conditions

Conditions passed to a Custom wrapper as arguments for its run, delete and update methods (see sections 6.3.2 and 6.3.3) come encapsulated in instances of `CustomWrapperConditionHolder`. The objects of this class contain two versions of the conditions passed to the Custom wrapper:

- A simplified version, available by calling the `getConditionMap` method. This version consists on an association between `CustomWrapperFieldExpressions` and `Objects`. For example, if we have a condition map like `{ FIELD1 - 100; FIELD2 - 'foo' }`, it means that the condition passed to the CUSTOM wrapper is `FIELD1 = 100 AND FIELD2 = 'foo'`. For a condition to be available as a map, it must match the pattern `FIELD_1 = value [AND FIELD_N = value]*`, in other case the `getConditionMap` method will return null. Conversion to map can be forced by calling the `getConditionMap(boolean force)` method, passing true as the value for force. Take into account that in this case the returned map may not be equivalent to the original condition.

- A `CustomWrapperCondition` instance. This version is available by calling the `getComplexCondition` method.

`CustomWrapperCondition` is the superclass of all the types of conditions that can be supported by a Custom wrapper:

- `CustomWrapperSimpleCondition` represents a simple condition. It holds the left expression (a `CustomWrapperExpression` object), an operator and the right expression (an array of `CustomWrapperExpression` objects).
  The right expression is stored in array, which usually contains only one expression but may contain more for operators such as `CONTAINSAND` or `CONTAINSOR` that may have several parameters.

- `CustomWrapperAndCondition` represents a series of conditions joined by the AND operator. It holds a list of `CustomWrapperCondition` objects.

- `CustomWrapperOrCondition` represents a series of conditions joined by the OR operator. It holds a list of `CustomWrapperCondition` objects.

- `CustomWrapperNotCondition` represents a negated condition. It holds a `CustomWrapperCondition`.

`CustomWrapperExpression` is the superclass of all the types of expressions supported by a Custom wrapper (for details, see the Javadoc documentation [VDP_API] for the following classes):

- `CustomWrapperFieldExpression`. This is the most common type of expression in a condition's left side. See section 6.3.3 for details.

- `CustomWrapperSimpleExpression`. This kind of expression has a type (one of the types defined in `java.sql.Types`) and a value.

- `CustomWrapperFunctionExpression`. Represents a function with parameters. This type of expression has a name, an optional modifier (`ALL` or `DISTINCT`), a list of parameters (instances of `CustomWrapperFunctionParameter`) and the property of being an aggregation function or not. A `CustomWrapperFunctionParameter` contains a list of `CustomWrapperExpressions`.

- `CustomWrapperConditionExpression`. Represents a condition parameter in a `CASE` function. Contains a `CustomWrapperCondition`.

- `CustomWrapperContainsExpression`. Represents an expression with the `CONTAINS` operator. Contains a literal and a search expression.

- `CustomWrapperArrayExpression`. Contains a list of `CustomWrapperExpressions`, all of the same kind.

- `CustomWrapperRegisterExpression`. Contains a list of `CustomWrapperExpressions` of any kind.

6.3.5 Dealing with the ORDER BY Clause

If the custom wrapper declares in the `CustomWrapperConfiguration` that it supports `ORDER BY` delegations (see section 6.3.6), the developer can invoke the method `getOrderByExpressions()` to obtain the expressions in the `ORDER BY` clause that is delegated to the custom wrapper. This method returns a list of `CustomWrapperOrderByExpression` objects that have the following attributes:

- `field`: the field that the rows should be sorted by.

- `order`: the order (ASC or DESC) of the sorting.

6.3.6 Configuring a Custom Wrapper

A CUSTOM wrapper can be configured with the `getConfiguration` method. This method must return an instance of the `CustomWrapperConfiguration` class, which encapsulates the following configuration parameters:

- `delegateProjections` (true by default). Defines whether a CUSTOM wrapper can deal with projected fields when being queried.

- `delegateCompoundFieldProjections` (`true` by default). Defines whether a CUSTOM wrapper can deal with compound projected fields when being queried.

- `delegateOrConditions` (`false` by default). Defines whether a CUSTOM wrapper can deal with `OR` conditions, as in `WHERE F1 = 1 OR F1 = 2` in SQL.

- `delegateNotConditions` (`false` by default). Defines whether a CUSTOM wrapper can deal with `NOT` conditions, as in `WHERE NOT (F1 = 1)` in SQL.

- `delegateArrayLiterals` (`false` by default). Defines whether a CUSTOM wrapper can deal with conditions containing arrays (as in `MY_INT_ARRAY = { ROW( 1 ), ROW( 2 ) }`).

- `delegateRegisterLiterals` (`false` by default). Defines whether a CUSTOM wrapper can deal with conditions containing registers (as in `MY_REGISTER = ROW( 1, 'A' )`).

- `delegateLeftLiterals` (false by default). Defines whether a CUSTOM wrapper can deal with conditions with literals in their left side (as in `100 = FIELD`).

- `delegateRightFields` (false by default). Defines whether a CUSTOM wrapper can deal with conditions with fields in their right side (as in `FIELD1 = FIELD2`).

- `delegateRightLiterals` (true by default). Defines whether a CUSTOM wrapper can deal with conditions with literals in their right side (as in `FIELD1 = 100`).

- `delegateOrderBy` (false by default). Defines whether a CUSTOM wrapper can deal with ORDER BY expressions.

- `allowedOperators` (by default, an array containing the operator '='). Defines which operators are supported in the conditions passed to the CUSTOM wrapper. In the Javadoc documentation [VDP_API] for the `setAllowedOperators` method of the `CustomWrapperConfiguration` class there is a list with all the possible operators.

The values for all of these properties can be obtained and defined by means of the appropriate getters and setters.

6.3.7 Updating the Custom Wrapper Plan

The method `getCustomWrapperPlan()` returns a `CustomWrapperPlan` object that represents the current wrapper execution plan. This object allows adding information to the wrapper plan that will be displayed in the execution trace. To add information to the wrapper plan invoke the method `addPlanEntry(String title, String entry)`. For example, if the custom wrapper queries a database, it could be useful to add to the wrapper plan information about the query executed in the database.

6.4 DEVELOPING CUSTOM INPUT FILTERS

When creating a DF, JSON or XML data source, you can select an input filter that preprocesses the data retrieved from the source, before the Execution Engine processes it. Besides providing several out of the box input filters, Virtual DataPort provides a Java API that allows you to develop filters that preprocess the data in any way you need.

Virtual DataPort includes a sample custom filter that reads the data from the source and replaces one character with another one. This example is at the folder `<DENODO_HOME>/samples/vdp/customConnectionFilter`. The `README` file in this directory explains how to compile, install and use this custom filter.

6.4.1 Required Libraries to Develop Custom Filters

To develop a custom filter, add the following jar files to the CLASSPATH of your environment:

- `<DENODO_HOME>/lib/contrib/denodo-commons-connection-util.jar`
- `<DENODO_HOME>/lib/contrib/denodo-commons-util.jar`

Note: if the custom filter relies on external jars, do the following:

- Copy the jars to the directory `<DENODO_HOME>/extensions/thirdparty/lib`
- Or, copy the contents of the required jars into the jar that contains the stored procedure. You have to copy the contents of the required jars, not the jars themselves.
- Or, import the external jars (see section *Importing Extensions* of the Administration Guide [ADMIN_GUIDE]) and when importing the new custom filter, select the jar with the custom filter and also the external jars.

6.4.2 Developing Custom Filters

To develop a custom filter, create a new class that extends the class `CustomConnectionFilter` (`com.denodo.parser.connection.filter` package).

This class has to implement the method `execute(InputStream is):InputStream`.

A custom connection filter may have input parameters. They can be useful if you want the behavior of the custom filter to be easily customizable. To retrieve the parameters entered by the user when assigning the filter to a data source, invoke the method `getParameters():Map<String, Object>` from the `execute(…)` method.

The folder `<DENODO_HOME>/samples/vdp/customConnectionFilter` contains an example of a custom filter.

After developing the custom filter, generate its jar and import it into Virtual DataPort (see section ”Importing Extensions” of the Administration Guide).

7 CUSTOM POLICIES

Custom policies are query interceptors, which are invoked before the Virtual DataPort server executes a query over a view. They are similar to row restrictions with the benefit that can be customized.

When a user queries a view with a custom policy assigned, the policy can take one of the following actions:

- Reject the query.
- Accept the query without restrictions.
- Or, accept the query but imposing restrictions such as limit the rows returned by the query, add a filter condition, etc.

To select one of these actions, custom policies have access to several parameters of the queries' context to decide how to proceed:

- The query the user wants to execute
- The user name and their privileges
- A JMX connection to the Server that the policy can use to access any Virtual DataPort data via JMX
- …

Custom policies are reusable, which means the following:

- As they are similar to row restrictions, they are assigned in a similar manner. Therefore, you can assign the same custom policy over several views for a user or a role.
- They can define configuration parameters. When a policy is assigned to a user or a role over a view, you can customize its behavior with these parameters. Thanks to this, the behavior of a policy can be customized depending on the user or role you assign it to.

  For example, if you develop a policy to limit the number of queries over a view, which the users of a role can execute at the same time, this number can be a parameter of the policy. That way, you can set a limit when assigning the policy to the role "developer" and another limit to the role "application".

When a user queries a view and this user has custom policies assigned over this view, the policies are evaluated in the following way:

- Custom policies are not taken into account when the user executing the query is an administrator or an administrator of the database.

- If the user does not have any role and she has custom policies assigned over the view, the Server evaluates the policies one by one. If one of the policies rejects the query, the query is rejected.

- If the user has one or more roles assigned (these roles may have other roles assigned), the evaluation of custom policies is performed in groups. For each role, there is a group formed by the custom policies assigned directly to this role and another group with the custom policies directly assigned to the user.

  A group rejects a query when at least one of the policies of the group rejects the query.
  A group accepts a query when all the policies of the group accept the query.

  The query is accepted if at least one group accepts the query.

For example, let us say that there is a user with two roles: *R1* and *R2*. The user has two policies assigned over the view *V*: *P1* and *P2*. The role *R1* of the user has another two policies assigned over the view *V*: *P3* and *P4*. The role *R2* has another two policies assigned over the view *V*: *P5* and *P6*.

When the user queries the view *V*, Virtual DataPort evaluates the policy *P1*. If *P1* accepts the query, it evaluates *P2*. If *P2* also accepts the query, the Server does not evaluate more policies and executes the query.

If *P1* rejects the query, the Server does not evaluate the other policies of the user and begins evaluating the policies of the role *R1*: *P3* and *P4*. If *P3* accepts the query, it evaluates *P4*. If *P4* also accepts the query, the Server does not evaluate more policies and executes the query.

If *P3* rejects the query, the Server does not evaluate the other policies of the role *R1* and begins evaluating the policies of the role *R2*: *P5* and *P6*. If *P5* accepts the query, the Server evaluates *P6*. If *P6* also accepts the query, the Server executes the query.

If *P5* rejects the query, the Server does not execute the query.

7.1 DEVELOPING A CUSTOM POLICY

A custom policy is a Java class with some annotations that mark the class as a custom policy and indicate which method the Server has to execute to intercept the query before executing it. Every time a custom policy is executed, the Server creates a new instance of the class.

We strongly recommend using the Denodo4E plugin for Eclipse, to develop custom policies (see the file `README` in `<DENODO_HOME>/tools/denodo4e`).

To develop a custom policy, add the `<DENODO_HOME>/lib/contrib/denodo-commons-commons-custom.jar` file to the Classpath of your environment.

You can find the Javadoc of the required classes and annotations in the `<DENODO_HOME>/docs/vdp/api` directory.

There is a sample custom policy in the directory
`<DENODO_HOME>/samples/vdp/customPolicies/`

This custom policy limits the number of concurrent queries that a user/role can execute over the same view or stored procedure. This custom policy has one input parameter called "Limit", which sets the maximum number of concurrent queries this user/role can execute.

The `README` file of this directory explains how to compile the example and import it into Virtual DataPort.

To develop a custom policy, create a new Java class and annotate it with the annotation `com.denodo.common.custom.annotations.CustomElement`. This annotation has the following parameters:

- `type`: it has to be `com.denodo.common.custom.annotations.CustomElementType.VDPCUSTOMPOLICY`
- `name`: name of the custom policy. The Administration Tool displays this value in the list of custom policies.

To access to the context of the query, add an attribute of the class `com.denodo.common.custom.policy.CustomRestrictionPolicyContext` and annotate it with `com.denodo.common.custom.annotations.CustomContext`.

At runtime, this attribute will hold the context of the query. That is:

- The query the user wants to execute: `getQuery()`.
- The fields involved in the query. That is, all the field in the `SELECT`, `WHERE`, `GROUP BY` and `HAVING` clauses: `getFieldsInQuery()`.
- The user who executes the query and her roles: `getCurrentUserName()` and `getCurrentUserRoles()`.
- Database where the query is executed: `getCurrentDatabaseName()`.
- User / role to whom the custom policy was assigned: `getPolicyCredentialsName()` and `getPolicyCredentialsType()`. The latter method returns if the policy is assigned to a user or a role.
- View / stored procedure that the custom policy was assigned to: `getElementType()` and `getElementName()`.
- Properties of the query. Invoke `getProperty(…)` to obtain the value of the property and `setProperty(…)` to change it. The available properties are the constants defined in the `CustomRestrictionPolicyContext` class: `I18N_PROPERTY`, `SWAP_PROPERTY`, etc.
- Provides a JMX connection to the Virtual DataPort server that the custom policy can use to retrieve any data via JMX: `getJmxConnection()`.
- Provides a method to log a message in the Server's logging system: `log(…)`.

When a custom policy is executed, the Server will execute the method marked with the annotation `com.denodo.common.custom.annotations.CustomExecutor`.

The Java class of the custom policy must have one and only one method marked with the annotation `CustomExecutor`. This method has to return a `com.denodo.common.custom.policy.CustomRestrictionPolicyValue` object.

If you want to add parameters to the custom policy, add a parameter to this method and annotate it with `com.denodo.common.custom.annotations.CustomParam`. This annotation has two parameters:

- `name`: the Administration Tool uses this parameter to display information about the custom policy to the users.
- `mandatory`: boolean value that indicates if this parameter is optional.

The value of these parameters is set when assigning the custom policy to a Virtual DataPort user or role.

The class `CustomRestrictionPolicyValue` (class of the objects returned by the policy) has two constructors:

- `CustomRestrictionPolicyValue(CustomRestrictionPolicyType policyType)`
  Constructs a `CustomRestrictionPolicyValue` without imposing any restriction.
- ```
  CustomRestrictionPolicyValue(
      CustomRestrictionPolicyType policyType,
      CustomRestrictionPolicyFilterType filterType,
      String condition,
      Set<String> sensitiveFields)
  ```
  Constructs a `CustomRestrictionPolicyValue`, which may impose a restriction.

`CustomRestrictionPolicyType` is an `enum` with the following fields:

- `REJECT`: it means that the policy rejects the query.
- `ACCEPT`: it means that the policy accepts the query and the Server will execute it.
- `ACCEPT_WITH_FILTER`: it means that the policy accepts the query but it sets some restrictions, which are determined by the fields of the second constructor: `filterType`, `condition` and `sensitiveFields`.

If you want the custom policy to accept (`ACCEPT`) or reject (`REJECT`) the query, instance the object using the first constructor.

If you want the custom policy to accept the query with some restrictions (`ACCEPT_WITH_FILTER`), use the second constructor and provide a non-null value for `filterType`, `condition` and `sensitiveFields`. In this case, the custom policy works in the same way as a restriction, so you have to define a condition, a set of sensitive fields and a type of filter.

The object `CustomRestrictionPolicyFilterType` tells the Server what to do when the query returns a row that does *not* verify the `condition`. The filter can be:

- `REJECT_ROW`: the Server will only include in the result of the query, the rows that verify the condition set by the parameter `condition`.
- `REJECT_ROW_IF_ANY_SENSITIVE_FIELDS_USED`: if the query uses at least one field of `sensitiveFields`, the Server will only return the rows that verify
  ```

`condition`.
If the query uses none of the `sensitiveFields`, the Server will not filter any row.

- `REJECT_ROW_IF_ALL_SENSITIVE_FIELDS_USED`: if the query uses *all* the fields of `sensitiveFields`, the Server will only return the rows that verify `condition`.
  If the query does not use *all* the fields in `sensitiveFields`, the Server will not filter any row.

- `MASK_SENSITIVE_FIELDS_IF_ANY_USED`: if the query uses at least one field of `sensitiveFields`, the Server will set to `NULL` the fields in the `Set sensitiveFields` of the rows that do *not* verify `condition`.
  If the query does not use any field in `sensitiveFields`, the Server will not mask any field.

- `MASK_SENSITIVE_FIELDS_IF_ALL_USED`: if the query uses *all* the fields of `sensitiveFields`, the Server will set to `NULL` the fields in the `Set sensitiveFields` of the rows that do *not* verify `condition`.
  If the query does not use *all* the fields in `sensitiveFields`, the Server will not mask any field.

8 APPENDICES

8.1 OUTPUT SCHEMA OF THE LIST COMMAND

This section lists the output of the of the `LIST` commands when executed from a JDBC or an ODBC client.

Note: when these commands are executed from the "VQL Shell" of the Administration Tool, they return information that should only be used for debugging purposes and may change in the future.

The output schema of all the `LIST` commands (`LIST WRAPPERS`, `LIST DATASOURCES` …) has only one column: `name`, except for the commands

- `LIST JARS`: returns the columns: `JAR_NAME`, `FUNCTIONS_TYPE`, `FUNCTIONS`. The values of the columns `FUNCTIONS_TYPE`, `FUNCTIONS` are `NULL` if the jar does not contain a custom function. Otherwise, the output contains a row for each type of function of the jar:
  - `FUNCTIONS_TYPE`: indicates the type of the function: `CONDITION` or `AGGREGATE`.
  - `FUNCTIONS`: names of the functions contained in the jar.
- `LIST FUNCTIONS CUSTOM`: contains the columns `NAME`, `TYPE` and `SYNTAX`. The command returns a row for each signature of each function. The `TYPE` column indicates if the function is an aggregation function or a condition function.

8.2 OUTPUT SCHEMA OF THE DESC COMMANDS

This section lists the output of the of the `DESC` commands when executed from a JDBC or an ODBC client.

When these commands are executed from the "VQL Shell" of the Administration Tool, they return information that should only be used for debugging purposes and may change in the future.

The output of these commands does not include any password for security purposes.

Command	Result Column Names
`DESC DATABASE <database name>`	`name` `description`

`DESC DATASOURCE ARN <data source name>`	`name`
	`aracne server route`
	`user name`

DESC DATASOURCE CUSTOM <data source name>	name class name classpath jars
DESC DATASOURCE DF <data source name>	name begin delimiter end delimiter column delimiter end of line delimiter header pattern tuple pattern data route
DESC DATASOURCE ESSBASE <data source name>	name database version uri user name
DESC DATASOURCE GS <data source name>	name google search server route proxy host proxy port proxy user
DESC DATASOURCE JDBC <data source name>	name database uri driver class name user name classpath database name database version ping query initial size max active
DESC DATASOURCE JSON <data source name>	name data route
DESC DATASOURCE LDAP <data source name>	name uri login

DESC DATASOURCE ODBC <data source name>	name dsn database uri driver class name user name classpath database name database version ping query initial size max active
DESC DATASOURCE OLAP <data source name>	name database name database version xmla uri user name max active initial size
DESC DATASOURCE SAPBW <data source name>	name database name database version system name xmla uri language user name
DESC DATASOURCE SAPBWBAPI <data source name>	name system name host name client id system number user name read block size

DESC DATASOURCE SAPERP <data source name>	name system name host name client id system number user name
DESC DATASOURCE WS <data source name>	name wsdl route authentication user authentication type: authentication user is empty = No authentication 1 = HTTP Basic authentication 2 = HTTP Digest authentication 3 = NTLM authentication 10 = WSS Basic authentication 11 = WSS Digest authentication proxy host proxy port proxy user Passwords are not included for security purposes.
DESC DATASOURCE XML <data source name>	name data route schema route dtd route
DESC MAP SIMPLE <map name> DESC MAP I18N <map name>	key value A row for entry of the map
DESC PROCEDURE <procedure name>	name type (parameter type) direction = { IN | OUT } : input or output parameter.
DESC ROLE <role name>	description roles roles is the name of a role assigned to this role. If a role has other rows assigned, this command returns a row for each role assigned to this role.

DESC SESSION	database user i18n activeTransaction true if this connection has started a transaction. false Otherwise.
DESC TYPE <type name>	field type If the type is complex, it returns a row for each component of the type. If it is simple (e.g. text), only the name of the type.
DESC USER <user name>	name description admin (true if the user is an Administrator. false otherwise)
DESC VIEW <view name>	fieldname fieldtype (a row for each field of the view) fieldTypeCode fieldPrecision fieldDecimals fieldRadix The last four fields are the values of the "Source type properties" of the fields of the view. Their value correspond with the properties SOURCETYPEID, SOURCETYPESIZE, SOURCETYPEDECIMALS and SOURCETYPERADIX respectively of the CREATE TABLE / VIEW statement that created the view. See more about these properties in the section "Viewing the Schema of a Base View" of the Administration Guide and in the section "JDBC Wrappers" of the Advanced VQL Guide.
DESC [SOAP | REST] WEBSERVICE <Web service name>	wsname wstypes operationtype = { 1 = SELECT, 10 = INSERT, 11 = UPDATE, 12 = DELETE } operationname fieldname fieldtype fielddirection (input or output parameter) Returns a row for each parameter of the Web Service.

DESC WRAPPER ARN <wrapper name>	name type = "arn" datasource name handler name filter main terms output schema
DESC WRAPPER CUSTOM <wrapper name>	name type = "custom" datasource name parameters output schema
DESC WRAPPER DF <wrapper name>	name type = "df" datasource name output schema
DESC WRAPPER ESSBASE <wrapper name>	name type datasource name server name application name cube name MDX sentence output schema
DESC WRAPPER GS <wrapper name>	name type = "gs" datasource name client languages site collections number of key match output schema

DESC WRAPPER ITP <wrapper name>	name type = "itp" creation date maintenance old sequence substitutions model content scanners output schema
DESC WRAPPER { JDBC | ODBC } <wrapper name>	name type = { "jdbc" | "odbc" } data source name relation name sql sentence aliases output schema
DESC WRAPPER JSON <wrapper name>	name type = "json" datasource name tuple root output schema
DESC WRAPPER LDAP <wrapper name>	name type = "ldap" datasource name object classes recursive search ldap expression output schema
DESC WRAPPER OLAP <wrapper name>	name type = "olap" datasource name catalog name schema name cube name mdx sentence output schema

DESC WRAPPER SAPBW <wrapper name>	name type = "sapbw" schema name cube name mdx sentence output schema
DESC WRAPPER SAPBWBAPI <wrapper name>	name type = "sapbwbapi" schema name cube name mdx sentence output schema
DESC WRAPPER SAPERP <wrapper name>	name type = "saperp" schema name bapi name output schema
DESC WRAPPER WS <wrapper name>	name type = "ws" datasource name service name port name operation name input message output message output schema
DESC WRAPPER XML <wrapper name>	name type = "xml" datasource name tuple root output schema

Table 5 Output schema of the DESC command depending on its parameters

8.3 ERROR CODES RETURNED BY VIRTUAL DATAPORT

The following table lists the error codes returned by the JDBC API of Virtual DataPort.

Error Type	Error Codes	Meaning
Authentication	20 21 22 23 Range from 600 to 700	The Server cannot authenticate the user. Usually, it means that the credentials of the user are not valid or that the user does not have enough privileges to connect to the database. If the user is trying to connect to an LDAP-authenticated database, it could there be an error establishing a connection to the LDAP server.
Parsing	1 19 Range from 1100 to 1199	There was an error parsing the query. It usually means that some clause of the query is misspelled or some parameter is missing.
Connection	Range from 10000 to 19999	The Server could not open a connection to a data source. This happens when the source is down, in case of JDBC data sources, the JDBC driver cannot be loaded, there was an error creating an instance of a Custom wrapper, etc.
Security	11 12 Range from 20000 to 29999	The user does not have enough privileges to execute the request. For example, when a user with READ privileges tries to execute an `INSERT` query.
Compute capabilities	2 9 Range from 30000 to 39999	There was an error while creating a view or preparing the execution plan of a query because: • The schema of the view or the query cannot be calculated. E.g., the query tries to project a field that does not exist. • Or, the restrictions of one or more sources used in the query are not met. E.g., the query does not provide a value for the mandatory fields.

Metadata management	5 8 10 17 18 24 Range from 40000 to 49999	There was an error loading/storing the metadata of an object (data source, wrapper, view, etc.). This may happen when there is already an object with the same name, the name of the new object is not valid, etc.
Execution	4 6 7 13 14 15 16 25 Range from 50000 to 59999	There was an error during the execution of the query.

Table 6 Error codes returned by the Denodo JDBC API

REFERENCES

[ADMIN_GUIDE] Virtual DataPort Administration Guide. Denodo Technologies.

[AXIS] Apache Axis. *http://ws.apache.org/axis/*

[INTELLISOFT] Intellisoft. *http://www.pgoledb.com/*

[JDBC] Java Database Connectivity (JDBC).
http://www.oracle.com/technetwork/java/javase/jdbc/

[NPGSQL] .Net Data Provider for PostgreSQL. *http://npgsql.projects.postgresql.org*

[ODBC] Microsoft Open DataBase Connectivity (ODBC).
http://support.microsoft.com/kb/110093

[TOM] Apache Tomcat. *http://tomcat.apache.org/*

[UX_ODBC] unixODBC. *http://www.unixodbc.org/*

[VDP_API] Javadoc documentation of the Developer API.
<DENODO_HOME>/docs/vdp/api/index.html

[VQL] Virtual DataPort VQL Advanced Guide. Denodo Technologies.

[WSDL] Web Services Description Language (WSDL). *http://www.w3.org/TR/wsdl*