<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Default values of the JDBC driver</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>PostgreSQL ODBC driver: configuration dialog</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>PostgreSQL ODBC driver: advanced configuration (Page 1)</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>PostgreSQL ODBC driver: advanced configuration (Page 2)</td>
<td>7</td>
</tr>
<tr>
<td>Figure 5</td>
<td>LIST syntax</td>
<td>16</td>
</tr>
<tr>
<td>Figure 6</td>
<td>DESO syntax</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Asynchronous iterator client code</td>
<td>21</td>
</tr>
<tr>
<td>Figure 8</td>
<td>General structure of the proprietary API</td>
<td>22</td>
</tr>
<tr>
<td>Figure 9</td>
<td>VDBDataSource configuration</td>
<td>24</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Superclass of all the partial results</td>
<td>25</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Hierarchy of partial results</td>
<td>27</td>
</tr>
<tr>
<td>Figure 12</td>
<td>General structure of a result assembler</td>
<td>28</td>
</tr>
<tr>
<td>Figure 13</td>
<td>ConsolePrinter code</td>
<td>32</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Code of a synchronous client</td>
<td>33</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Output schema of the DESO command depending on its parameters</td>
<td>38</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Configuration of the EasySoft ODBC/JDBC Gateway</td>
<td>40</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Configuration of OpenLink ODBC/JDBC Lite (1)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Configuration of OpenLink ODBC/JDBC Lite (2)</td>
<td>42</td>
</tr>
</tbody>
</table>
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:

- Presents the basic tasks required so that a client application can access Virtual DataPort.

- Describes how to access a Virtual DataPort server using the standard Java API for accessing JDBC Databases.

- Describes how to access a Virtual DataPort server through ODBC.

- Describes how to access Virtual DataPort views published as Web Services from client applications.

- Describes the generic API based on Web Services for executing any VQL statement on a Virtual DataPort server.

- Details all the peculiarities of the Denodo Virtual DataPort native development API.

- Includes or lists various examples of client development that uses Virtual DataPort both synchronously (where the client is blocked until the complete response is received from the server) and asynchronously (where the client controls the communication flow with the server and can process statement results as they are obtained from the sources).
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 (Virtual Query Language), a SQL-like language used to query and define data views (see the Administration Guide [1] and the VQL Advanced Guide [2]).

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 [1] 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:

- Access through JDBC (Java DataBase Connectivity) [5]. Virtual DataPort incorporates a JDBC driver that client applications can use for this purpose.
- Access through ODBC (Open DataBase Connectivity) [10]. Virtual DataPort provides an ODBC interface for ODBC clients.
- Access to views published as Web Services. The Virtual DataPort administration tool (see the Administration’s Guide [1]) allows any data view defined in DataPort to be published as a Web Service.
- Access through a generic Web Service interface. DataPort includes a generic Web Service interface to run VQL statements on a DataPort server. The WSDL [7] file is provided that can be used to generate the necessary stubs to access this interface.
- Access through the native API. DataPort also includes a native JAVA API for interaction with the server.

In the DENODO_HOME\samples\vdp\vdp-clients\ path you can find a README file with useful information about the different Virtual DataPort access interfaces.

This guide presents the basic guidelines for using each of these access modes and some examples of use are indicated. Reading the Javadoc documentation [3] is recommended for greater detail on classes, attributes and operations [3].

1.1 EXAMPLES AND HELP FILES

There are various examples of implementation of client applications for VDP in the path DENODO_HOME\samples\vdp\vdp-clients, using the different access methods mentioned in this document. For each example, source code and scripts are included for compilation and execution. See the file DENODO_HOME\samples\vdp\vdp-clients\README for more information on the examples.
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 JDBC 3.0 [5]. These are some of the features of the JDBC specification supported by the Virtual DataPort JDBC driver:

- The data types supported are those defined in the VQL Advanced Guide [2] (includes support for all basic types and for fields of the type array and register).
- Execution of view creation, query, insertion, update and deletion statements [2].
- Support for metadata description statements and listing of server catalog elements.
- Support for PreparedStatement, so the parameters of a VQL statement can be delegated to the Virtual DataPort server.
- Support for canceling the current statement execution by using the cancel() method of the java.sql.Statement class. When a query is cancelled, the 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 cancelling were effective. Therefore, the query cancellation does not imply closing the ResultSet that is being used.
- Support for querying wrappers by using the QUERY WRAPPER statement [2].
- Invocation of stored procedures using the CALL statement [2].
- Support for submitting batches of commands.

The JDBC driver for Virtual DataPort is a type 2 driver, whereby the library with the native DataPort API must be available in the machine that executes the client application.

Four basic elements are required for JDBC connection to a DataPort server:
- Class that implements the JDBC driver [com.denodo.vdp.jdbc.Driver].
- Connection string, which indicates the machine, port and database name (jdbc:vdb://<hostName>:<port>/<databaseName>).
- User name to be used when connecting to the server.
- User password for accessing the server.

For instance:
jdbc:vdb://localhost:9999/admin?queryTimeout=100000&chunkTimeout=95000

Optionally, a list of specific configuration parameters of Virtual DataPort can be added. The figure below, lists the parameters of the driver along with its default values.
Figure 1  Default values of the JDBC driver

The complete syntax for the JDBC connection URL is the following:

```
jdbc:vdb://<hostName>:<port>/<databaseName>?paramName1=paramValue1&... &paramNameN=paramValueN
```

Section 5.4 describes each of these parameters, except `reuseRegistrySocket`, `pingQuery` and `pingQueryTimeOut` that are valid for the JDBC driver but not for the Virtual DataPort API.

- **reuseRegistrySocket**. In a cluster environment with a load balancer, set this property to false. If false, because of how RMI works, the connections will be more evenly distributed through the Virtual DataPort servers of the cluster, than if the property is left to its default value (false).

  Load balancing is performed on a connection level, not on a query level. Therefore, when a client establishes a connection with a server of the cluster, all the queries of this connection are executed in the same server. In an environment with a load-balancer, if you use a connections pool, you must define a Ping Query and a Ping Query Timeout. When your application requests a connection to the pool, the pool executes this query. If the query does not finish or takes longer than the Timeout, the pool considers the connection invalid. If your connections pool does not support Ping Query Timeout, use the properties `pingQuery` and `pingQueryTimeOut` of the JDBC driver.

- **pingQuery**. Query that will be used to check the load of Virtual DataPort.

- **pingQueryTimeOut**. When a client executes the `pingQuery` and its execution takes more milliseconds than `pingQueryTimeOut`, the driver will throw a SQLException.

Sample scenario:

- You are using a Connections Pool that does not support Ping Query Timeout.
- The Ping Query of the Connection Pool is SELECT * FROM Table and you know that this query should not take longer than 2 seconds.

The URI of the JDBC driver to connect to the Virtual DataPort server should be like:

```
jdbc:vdb://acme:9999/admin?pingQuery=SELECT * FROM Table&pingQueryTimeOut=2000
```

With this URI, every time the pool executes this query, the driver checks that it does not take more than 2 seconds (pingQueryTimeOut). Otherwise, the driver throws a SQLException. Because of this, the pool will consider the connection invalid.

**NOTE:** The JDBC driver uses the following external libraries: Jakarta commons-collections, Jakarta commons-pool, Jakarta commons-logging, Jakarta commons-lang and Apache log4j. In the same path, another version of the driver is provided without these external libraries (denodo-vdp-
jdbcdriver-basic.jar}. This version can be used in applications already using other compatible versions of the same libraries for other purposes. It is important to take into account that the libraries employed must be compatible with the versions included with the product. Otherwise, the JDBC driver could work incorrectly.

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).
3 ACCESS THROUGH ODBC

ODBC (Open Database Connectivity [10]) is a standard API specification for using database management systems. Virtual DataPort provides access to ODBC clients through the PostgreSQL ODBC driver. By using an ODBC driver, we can use applications such as Excel to query Virtual DataPort.

Note: the ODBC driver has to be installed in all servers and workstations that need to access to Virtual DataPort through ODBC.

3.1 CONFIGURATION OF THE ODBC DRIVER IN WINDOWS

The PostgreSQL driver is not bundled with the Denodo Platform, so we have to download it from the PostgreSQL website. As any other ODBC drivers, we have to install it in the machine where the client application is running. Denodo Virtual Dataport has been successfully tested with the versions 8.0 and 9.0 of the driver.

3.1.1 Install the ODBC Driver

Follow these steps:

   Note: 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.

2. Extract the contents of the ZIP file and execute the MSI file in it.

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

The installation is now complete.

3.1.2 Set up a DSN (Data Source Name)

Ensure that you have logged in using an account with administrative rights.

Follow these steps:

1. Open the “ODBC Data Sources” applet of the “Administrative Tools” (Control Panel). To configure 32 bits DSNs in a 64 bits Windows, run %SystemRoot%\SysWOW64\odbcad32.exe instead.

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.

3. Select the “PostgreSQL Unicode” driver and click on “Finish”

4. In the configuration dialog fill in the following information:
   - Database: database in Virtual DataPort. E.g. admin.
- **Server** and **Port**: host name and port of the server that runs Virtual DataPort. The default ODBC port is 9996.

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

I.e.:

![PostgreSQL ODBC driver: configuration dialog](image)

**Figure 2** PostgreSQL ODBC driver: configuration dialog

5. Click on the button “Datasource” to open the “Advanced options” dialog:
   - Select “Use Declare/Fetch”.
   - Deselect “KSQO (Keyset Query Optimization)”

I.e.:
6. Click on "Page 2" and in the area 'Level of rollback on errors', select 'Transaction'. See Figure 4:

7. Click on the button “Test” to test the connection to Virtual DataPort.
8. Click “Ok”.

The DSN is now configured and ready to be used.

## 3.2 Configuration of the ODBC Driver in Linux and Other UNIX

### 3.2.1 Install the ODBC Driver

The installation of the ODBC driver in non-Windows platforms is more complex than in Windows because we have to download the source code and compile it.


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

   ```bash
   $> tar -zxf psqlodbc-xx.yy.zzzz.tar.gz
   $> cd psqlodbc-xx.yy.zzzz
   $> ./configure --without-libpq
   $> make
   ```

3. Execute the following command as user root to install the driver:

   ```bash
   $> make install
   ```

The installation is now complete.

### 3.2.2 Set up a DSN in Linux and Other UNIX

Linux does not provide an ODBC driver manager, so we have to install one. This section explains how to install and configure unixODBC [13]. 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](http://www.unixodbc.org/download.html)

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

   ```bash
   $> tar -zxf unixODBC*.tar.gz
   $> cd unixODBC*
   $> ./configure
   $> make
   ```

3. Execute the following command as user root:

   ```bash
   $> make install
   ```

#### 3.2.2.2 Register the PostgreSQL ODBC Driver in UnixODBC

Follow these steps:
1. Edit the file `/etc/odbcinst.ini` and add the following:

```
[postgresqlDriver]
Description=PostgreSQL 09.00.0200 driver
Driver=/usr/lib/psqlodbcw.so
UsageCount=1
```

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

2. Create a new file `postgresqlDriver.template` with the data that we have added to `/etc/odbcinst.ini` in step 1.

3. Execute this to register the PostgreSQL driver in the Driver Manager:

```
$> odbcinst -i -d -f postgresQLDriver.template
```

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

```
$> odbcinst -q -d
```

To uninstall the driver, execute:

```
$> odbcinst -u -d -n postgresQLDriver
```

3.2.2.3 Configure the DSN in UnixODBC

Follow these steps:

1. Edit the file `/etc/odbc.ini` and add the following:
2. Create a file “newODBCDriver.template” that contains what we have added to `odbc.ini` in step 1.

3. Execute this to create a new DSN:

```bash
$> odbcinst -i -s -f newODBCDriver.template
```

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

```bash
$> odbcinst -q -s
```

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

```bash
$> odbcinst -u -s -n <DSN name>
```
4 WEB SERVICE INTERFACE

4.1 ACCESS TO VIEWS PUBLISHED AS WEB SERVICES

The Virtual DataPort administration tool allows publishing any data view or stored procedure defined in Virtual DataPort as a Web Service. See the Administration's Guide[1] for information on this process.

These published views allow executing queries on views. If the exported view is updatable, insert, update and delete operations can also be exported.

Using the Denodo Virtual DataPort administration tool [1], the published web services can be deployed in the Web service container embedded in the Denodo Platform. Therefore, no external application server is needed. Alternatively, it is also possible to generate a .war file that can be installed in an external J2EE Web container such as Apache Tomcat [8].

The Virtual DataPort administration tool also allows generating a .wsdl file which can be used with a utility for Web Service programming (such as those included in Apache Axis [9]) to generate the necessary stubs to implement a client program accessing the Web Service.

The generated .war and .wsdl files are available for download in the /export path of the Web service container embedded in the Denodo Platform. For instance, if the default port is used and the access is made from the host where DataPort is installed, the path would be [http://localhost:9090/export](http://localhost:9090/export).

Examples of client programs accessing a view published as a Web Service can be found in the path DENODO_HOME\samples\vdp\vdp-clients (see the README file in this path which explains how to generate and publish the views accessed by the sample clients).

4.2 GENERIC WEB SERVICE INTERFACE

Virtual DataPort may also be accessed through a generic Web Service interface that allows any statement written in VQL to be run.

Said Web Service is deployed during installation in the Web Service container embedded in the Denodo Platform. The name of the web application is denodo-vdp-wsclient and it can be accessed at: [http://localhost:9090/denodo-vdp-wsclient/services](http://localhost:9090/denodo-vdp-wsclient/services).


A .war file is also provided, should it be desired to install the service in an external JEE application server [denodo-vdp-wsclient.war file situated in the path DENODO_HOME\webapps\vdp-wsclient].

Two types of client applications can be constructed that use the generic Web Service interface:

- **Stateless Client**. In this case, the client establishes a session with the server each time a statement is issued, closing it once execution has concluded.

- **Stateful Client**. In this mode, the client establishes a session with the server, uses said session to issue statements to same and closes it, when it has concluded. This avoids the cost of establishing a session for each statement. Besides, if stateful clients are used, it is also possible to request the results to the DataPort server in an asynchronous way. This means that it will not be necessary to wait for DataPort to retrieve all results from a query before starting their processing in the client.
The following subsections describe the manner in which the client applications of each of these types can execute statements on a DataPort server. The format in which the response to said statements is obtained is then described. Finally, how to configure the connection pool used by the Web service to execute statements on DataPort is described.

Examples of both client types can be found in the path `DENODO_HOME\samples\vdp\vdp-clients` (see README file in said path).

### 4.3 EXECUTING STATEMENTS IN STATELESS CLIENTS

This type of client can directly use the `executeQuery` and `executeQueryByConfig` operations to run queries on a DataPort server. To execute insert, update or delete statements the analogous `executeUpdate` and `executeUpdateByConfig` operations can be used. `executeQuery` and `executeUpdate` receive the following input parameters:

- VQL statement to run on the server (`String`-type element).
- URI connecting to the DataPort server (`String`).
- Login ID with which the connection to the DataPort server will be made (`String`).
- Password associated with the specified user (`String`).
- Maximum time (in milliseconds) the client is willing to wait for the statement to complete (`int`-type element). This parameter is optional. If not indicated (or the value 0 is indicated), then it waits until running is complete.

In addition, `executeQuery` has the following additional parameters:

- Maximum time (in milliseconds) the client is willing to wait for a partial result to arrive (`int`). Where this time is exceeded, Virtual DataPort returns the new results available (or an empty partial result if there are not new results available. This is an optional parameter. If not specified (or the value 0 is received), the server returns all the results together at the end of the statement run.
- Number of results that make up a partial result (`int`). When Virtual DataPort has available this number of results, they are returned to the client as a partial result even though the timeout specified in the previous parameter has not been exceeded. This is an optional parameter. If not specified (or the value 0 is received), the server returns all the results together at the end of the statement run.

The last two parameters provide control over how the server and the client exchange results returned by the statement. Allowing the server to return results to the client as they become available can influence the execution time.

The `executeQueryByConfig` and `executeUpdateByConfig` operations allow the same function but grouping the input parameters together as follows:

- A `String`-type element containing the VQL statement to run on the server.
- A `WSConfigVO`-type element (`WSUpdateConfigVO` in the case of `executeUpdateByConfig`) that contains the data required to locate the server and configure the connection.

`WSConfigVO`-type elements contain the following subelements:

- `vdpURL`. Should contain the connection URI to the DataPort server.
- `vdpUser`. User identifier with which the connection to the DataPort server will be made.
- `vdpPassword`. Password associated with the specified user.
• **vdpQueryTimeout.** Maximum time (in milliseconds) the client is prepared to wait for the statement to complete. This parameter is optional. If not indicated (or the value 0 is received), then it waits until the execution has concluded.

• **vdpChunkTimeout.** Where this time is exceeded, Virtual DataPort returns the new results available (or an empty partial result if there are not new results available). This is an optional parameter. If not specified (or the value 0 is received), the server returns all the results together at the end of the statement run.

• **vdpChunkSize.** When Virtual DataPort has available this number of results, they are returned to the client as a partial result even though the timeout specified in the previous parameter has not been exceeded. This is an optional parameter. If not specified (or the value 0 is received), the server returns all the results together at the end of the statement run.

The **WSUpdateConfigVO** elements are identical to the **WSConfigVO** elements but they do not have the last two sub-elements.

**4.4 EXECUTING STATEMENTS IN STATEFUL CLIENTS**

In this mode, the client establishes a session with the server and uses it to issue statements until it decides to close it.

• The **openConnection** operation allows establishing a session with the Virtual DataPort server. A **WSConfigVO** object that contains the data required to locate the server and configure the connection (see previous section) is given as a parameter. A **String**-type session identifier is returned.

• Once the session has been established, the **executeQuery** and **executeUpdate** operations can be used to issue, respectively, query statements and insert, update, delete statements on the DataPort server. In this client type these operations receive two **String**-type parameters:
  
  o The VQL statement to execute.
  o The session identifier returned by **openConnection**.

• Alternatively, and only for query statements, the **executeAsyncQueryBySession** operation may also be used to execute the query in an asynchronous way. This way, it will not be necessary for the DataPort server to finish the query execution before the client starts processing the results. This operation receives the same parameters than the previous one, but the manner in which results are processed is different (see section 4.5.2).

• The client may cancel the execution of a statement while it is still running, by using the **cancelQuery** method. This method receives the connection identifier returned by **openConnection** as input parameter. When canceling the query, the DataPort server will cancel all current accesses to the data sources and to the cache. After invoking **cancelQuery**, it is still possible for the server to return some results, if these were retrieved before the source access cancellation was effective. Therefore, even though a query may have been cancelled, the client might still request more results to the server.

• The client can close the session invoking the **closeConnection** operation which receives the session identifier to be closed as a parameter.

**4.5 PROCESSING THE RESULTS OF A STATEMENT**

Depending on the method use to execute the statement, the processing of the retrieved results will be performed either **synchronously** (results will not be processed until the execution has finished in the server) or **asynchronously** (results can be processed by the client whenever the DataPort server retrieves them from the sources). Section 4.5.1
Virtual DataPort 4.6 Developer Guide

Web Service Interface

4.5.1 Synchronous Processing

The `executeQuery` and `executeQueryBySession` methods return a `WSTableVO`-type element that encapsulates all the data returned by the query execution process. It contains the following subelements:

- **executionError.** If no error occurs, it takes NULL as value. Where an error does occur, it will contain a `WSErrorVO`-type element with two sub-elements:
  - **errorCode.** Contains the code of the error. The error codes are the same used by the DataPort native JAVA API (see the Javadoc documentation [3] of the class `com.denodo.vdb.vdbinterface.common.clientResult.ErrorCodes` for details on the possible error codes [3]).
  - **errorMessage.** Contains a message explaining the error that has occurred.

- **schema.** Element that contains metadata about the schema of the results returned by the statement. It is an array of `WSColumnMetadataVO`-type elements, where each element represents the name and the type of an attribute of the results.

- **data.** Contains the tuples obtained as a response to the statement. This is an array of `WSRowVO` elements, where each element represents a tuple of the statement result. Each `WSRowVO` element is comprised in turn of an array of `WSValueVO` elements. Each `WSValueVO` element represents the value of an attribute of the tuple.

It is important to remember that DataPort uses a data model that supports compound types: register types and array types. Array-type compound attributes always contain a register on an internal level that defines the schema of each of the elements of the array (see [1] for more information).

Therefore, the `WSValueVO` objects used to represent the value of an attribute in a tuple of the result can be of three subtypes:

- **WSSimpleValueVO.** Represents a value of simple-type data. There is one subtype for each of the simple DataPort types available (`WSIntValueVO`, `WSLongValueVO`, `WFloatValueVO`, `WSDoubleValueVO`, `WSTextValueVO`, `WSBlobValueVO`, `WSBooleanValueVO`, `WSDateValueVO`, `WSMoneyValueVO`, `WSMoneyEnumeratedValueVO`). All the subtypes define a subelement called `value` of the corresponding basic XML data type (`int` in the case of `WSIntValueVO`, `string` in the case of `WSTextValueVO`, etc.).

- **WSRegisterValueVO.** Represents a value of the register data type. Contains a `fieldValues` element, which is an array of `WSRegisterFieldValueVO` elements. Each `WSRegisterFieldValueVO` element represents the value of one of the fields of the data register and has two subelements:
  - **name.** A `String` that contains the field name.
  - **value.** A `WSValueVO`-type element that contains the field value.

- **WSArrayValueVO.** Represents a value of the array data type. Contains a `values` element, which is an array of `WSRegisterValueVO` elements. Each of these represents a value of the data array.

Likewise, the `WSColumnMetadataVO` elements that represent the metadata of the schema of an attribute returned in the query result contain two subelements:

- **columnName.** A `String` that contains the name of the attribute.
• **columnName**: Contains a **WSTypeVO**-type element that describes the attribute type. The **WSTypeVO** elements contain a **typeName** subelement that indicates the name of the data type.

The **WSTypeVO** elements can also be one of these subtypes:

• **WSSimpleTypeVO**: Represents an attribute of a simple type (e.g., int, float, String, etc.) and does not have additional subelements.

• **WSRegisterTypeVO**: Represents a register-type attribute. Contains a **fields** subelement that is an array of **WSRegisterFieldMetadataVO** elements. Each of the elements of this array contains the metadata of a register field and is formed of two subelements:
  • **name**: A **String** that contains the name of the field.
  • **type**: Contains a **WSTypeVO**-type element that describes the field type.

• **WSArrayTypeVO**: Represents an array-type attribute. Contains a **basetype** subelement of the type **WSRegisterTypeVO** that describes the type of elements in the array.

• **WSEnumeratedTypeVO**: Represents an enumerated-type attribute. Contains a **values** subelement (of array of strings type) with one element for each possible value of the enumerated type.

The **executeUpdate** and **executeUpdateBySession** operations return a **WSAffectedRowsVO** element. This element contains one integer-type sub-element called **affectedRows** indicating the number of tuples affected by the operation, and one sub-element of **WSErrorVO** type containing the same information as in the **executeQuery** and **executeQueryBySession** operations.

### 4.5.2 Asynchronous Processing

The **executeAsyncQueryBySession** method returns an object of type **WSQueryInfoVO**. Objects of this type are identical to the **WSTableVO** objects described in the previous section, but without containing the data sub-element. Therefore, the object may be used to obtain the schema of the data which will be returned by the statement, and to gather information about possible returned errors. However, it will not store reply data.

To obtain the statement results, the **getNextResults** method is required. This method accepts two input parameters:

• **maxResults**: It is an integer-type parameter that lets the user define the number of rows that are requested to the server. Whenever the server has available that number of results, it will return them to the client. The client will have to invoke **getNextResults** to obtain more results. If, while invoking **getNextResults**, the query finishes before reaching the requested number of results, the server will also return the results.

• **sessionID**: This is the session identifier returned by **openConnection**.

The result of invoking **getNextResults** is a **WSPartialResultVO**-type object. It is comprised of the following three sub-elements:

• **data**: It stores all rows retrieved after a statement was executed. It is identical to the **data** sub-element of the **WSTableVO** object described in the previous section.

• **executionError**: Identical to the **executionError** sub-element of the **WSTableVO** object described in the previous section.

• **endResult**: Boolean value that lets the user know whether the query has finished (true) or not (false).
4.6 INFORMATION ABOUT OBJECTS OF THE DATABASE

Besides using the JDBC, ODBC and Web Service interfaces to execute queries to obtain and modify data from the Virtual DataPort server, it is also possible to execute queries to describe the objects of the database:

- Execute LIST sentences to obtain a list of the existing views, data sources, users, etc. The syntax for this command is:

```
LIST { DATABASES | USERS | I18NS }
LIST TYPES [ ENUMERATED | ARRAY | REGISTER ]
LIST VIEWS [ BASE | ALL ]
LIST OPERATORS [ <data_type:identifier> ]
LIST WRAPPERS <wrapper type>
LIST DATASOURCES <datasource type> | LDAP ALL
LIST MAPS { I18N | SIMPLE }
LIST PROCEDURES
LIST WEBSERVICES
LIST WIDGETS
LIST JARS

<datasource type> ::= { ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | WS | XML }
<wrapper type> ::= <datasource type> | ITP
```

Figure 5 LIST syntax

Section 6.1 details the output schema of the LIST command.

- Execute DESC sentences to obtain information about a certain view, data source, user, etc (the information returned depends on the type of the element) The syntax for this command is:

```
DESC { DATABASE | USER | TYPE | PROCEDURE | VIEW } <name:identifier>
DESC DATASOURCE <datasource type> <name:identifier>
DESC MAP { I18N | SIMPLE } <name:identifier>
DESC SESSION
DESC WEBSERVICE <name:identifier>
DESC WIDGET <name:identifier>
DESC WRAPPER <wrapper type> <name:identifier>

<datasource type> ::= { ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | WS | XML }
<wrapper type> ::= <datasource type> | ITP
```

Figure 6 DESC syntax

Section 6.1 details the output schema of each DESC command (DESC WEBSERVICE has a different output schema than DESC DATABASE)

- Manage the embedded Web Container:
WEBCONTAINER START : starts the Web Container.
WEBCONTAINER STOP : stops the Web Container.
WEBCONTAINER STATUS : obtain the status of the Web applications deployed: WebContainerStarted = { true | false }, WSName, ContextPath.

4.7 CONNECTION POOL CONFIGURATION

Each time that a Web service operation is invoked, it is translated into a statement and executed against the DataPort server to obtain the required results. It is possible to configure certain connection parameters used by the Web service to access DataPort.

The most important aspect that can be configured is the use of a connection pool to allow the connections opened by the Web service with DataPort to be reused. In production environments, the use of the connection pool is strongly recommended.

The configuration of the connection parameters is made on the web.xml file at the DENODO_HOME/resources/apache-tomcat/webapps/denodo-vdp-wsclient/WEB-INF path. The configurable parameters are:

- poolEnabled. true or false to respectively use the pool or not.
- poolInitSize. Initial number of connections in the pool.
- poolMaxActive. Maximum number of connections in the pool.

In the web.xml file, each parameter appears as an environment variable for the Web service. The property name appears as the value of the env-entry-name element of the variable and the value must be set on env-entry-value. For instance, the next fragment shows the poolEnabled parameter configured with the true parameter:

```
<env-entry>
  <env-entry-name>poolEnabled</env-entry-name>
  <env-entry-value>true</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```
5 VIRTUAL DATAPORT DEVELOPMENT API

Denodo Virtual DataPort also offers a Java API that allows executing any VQL statement on the server.

Two fundamental stages exist when executing a statement using Virtual DataPort. In the first stage, the statement is sent to the server and, in the second, the results are requested, received and interpreted.

When the server receives a request for the results of a statement, it returns all the tuples of the result processed to that time.

Optionally, it is possible to specify the maximum number of elements the client wishes to receive in every partial result ("chunk") exchanged between the server and the client. This last characteristic is recommended only in the case where the quantity of results available to the server is very high and the probability that the client will use all the results is very low.

Likewise, the client can receive various types of messages as the response of a query:

- Messages with tuples. Contain part of or all the tuples that form the query result.
- Empty messages. Indicate that the server does not have any more responses for the query as yet.
- End messages. Indicate that query execution has ended in the server.
- Error messages. These are sent, if an error occurs, when the query is being executed.

There are four fundamental classes in the development of a Virtual DataPort client:

- `com.denodo.vdb.vdbinterface.common.datasource.VDBDataSource`. Defines the data required to connect to the server.
- `com.denodo.vdb.vdbinterface.common.connection.VDBConnection`. Front-end to which the clients make their queries. Represents a connection with the server.
- `com.denodo.vdb.vdbinterface.common.clientResult.PartialResult`. Parent class of all the message types that the client can receive.
- `com.denodo.vdb.vdbinterface.client.printer.Printer`. Responsible for converting results issued by the server to the format required by a specific client type.

5.1 CONFIGURATION OF CLIENT APPLICATIONS

The necessary .jar files in the application CLASSPATH to use the Virtual DataPort Client API are as follows:

1. In Windows systems:
   a. `%DENODO_HOME%\lib\contrib\commons-collections.jar`

\(^1\) Using this option can increase the communication required between client-server, whereby it is advisable that it only be used when necessary.
b. `%DENODO_HOME%/lib/contrib/commons-logging.jar`
c. `%DENODO_HOME%/lib/contrib/commons-pool.jar`
d. `%DENODO_HOME%/lib/contrib/denodo-util.jar`
e. `%DENODO_HOME%/lib/contrib/log4j.jar`
f. `%DENODO_HOME%/lib/vdp-client-core/denodo-vdp-client-base.jar`
g. `%DENODO_HOME%/lib/vdp-client-core/denodo-vdp-client-ext.jar`

2. In Unix systems:
   a. `$DENODO_HOME/lib/contrib/commons-collections.jar`
   b. `$DENODO_HOME/lib/contrib/commons-logging.jar`
   c. `$DENODO_HOME/lib/contrib/commons-pool.jar`
   d. `$DENODO_HOME/lib/contrib/denodo-util.jar`
   e. `$DENODO_HOME/lib/contrib/log4j.jar`
   f. `$DENODO_HOME/lib/vdp-client-core/denodo-vdp-client-base.jar`
   g. `$DENODO_HOME/lib/vdp-client-core/denodo-vdp-client-ext.jar`

The Apache Log4j API package is used for log files management. A configuration file would have to be included in the CLASSPATH to configure the trace levels. A sample file can be found in the `%DENODO_HOME%/conf/vdp-admin/log4j.xml` path.

The values of some configuration parameters can be altered using a configuration file that must be included in the CLASSPATH. A sample file can be found in `%DENODO_HOME%/conf/vdp-admin/VDBAdminConfiguration.properties`.

The following section provides:

- An example of a statement execution using the Virtual DataPort API.
- A global view of the Virtual DataPort API and details about its classes.
- A few samples of how to develop a Virtual DataPort client using its API. These samples explain the two possible approaches when developing a Virtual DataPort client:
  1. **Synchronous** approach: the client is blocked until it receives the complete response from the server.
  2. **Asynchronous** approach: the client controls the communication flow with the server and can start processing results before receiving all of them.

### 5.2 EXAMPLE

Before providing a detailed description of the Virtual DataPort API, we will see a sample application that sends a query to the server and retrieve the results asynchronously. The section 5.7.2 contains more details about creating asynchronous clients.

```java
import com.denodo.vdb.vdbinterface.client.printer.standard.StandardRowVO;
import com.denodo.vdb.vdbinterface.client.util.VQLStatement;
import com.denodo.vdb.vdbinterface.client.util.datasource.DataSourceLocator;
import com.denodo.vdb.vdbinterface.client.util.iterator.VDBResultSetIterator;
import com.denodo.vdb.vdbinterface.common.clientResult.MetadataField;
import com.denodo.vdb.vdbinterface.common.clientResult.vo.descriptions.type.TypeVO;
import com.denodo.vdb.vdbinterface.common.clientResult.vo.sentences.ValueVO;
```
import com.denodo.vdb.vdbinterface.common.connection.VDBConnection;
import com.denodo.vdb.vdbinterface.common.datasource.VDBDataSource;
import com.denodo.vdb.vdbinterface.common.connection.VDBConnectionException;
import java.util.*;

public class IteratorDemoClient {
    public static void main(String args[]) {
        String dsName = "clientDs";
        String sentence = "SELECT * from view";
        String url = args[0];
        String login = args[1];
        String passwd = args[2];
        try {
            VDBDataSource tempds = new VDBDataSource(dsName, "testDS", url, login, passwd);
            DataSourceLocator.addDataSource(tempds);
            VDBDataSource ds = DataSourceLocator.getDataSource(dsName);
            VDBConnection vdpConnection = ds.getConnection();
            VQLStatement vqlStatement = vdpConnection.createVQLStatement();
            vqlStatement.setPreparedStatement(sentence);
            VDBResultSetIterator resultSetIterator = vqlStatement.executeQuery();

            int numOfTuples = 0;
            while (resultSetIterator.hasNext()) {
                numOfTuples++;
                StandardRowVO tuple = (StandardRowVO)resultSetIterator.next();
                Map map = tuple.getValues();
                Iterator it = map.keySet().iterator();
                while (it.hasNext()) {
                    ValueVO valueVO = (ValueVO)map.get(it.next());
                    System.out.print(valueVO + "	");
                }
            }

            System.out.println("Messages:");
            System.out.println(resultSetIterator.getMessages());

            System.out.println("Schema:");
            MetadataField[] schema = resultSetIterator.getSchema();
            if (schema != null) {
                for (int i=0; i<schema.length; i++) {
                    TypeVO type = schema[i].getType();
                    System.out.println(schema[i].getName() + ": " + type);
                }
                System.out.println("\n");
            }
        } catch (Exception e) {
            System.err.println("Error trying to access the server ... ");
        }
    } finally {
        try {
            if (vdpConnection != null) {
                vdpConnection.close();
            }
        } finally {
            if (vdpConnection != null) {
                vdpConnection.close();
            }
        }
    }
}
vdpConnection.close();
) catch (VDBConnectionException e) { System.err.println("Error closing connection ... "); e.printStackTrace();
)
}
}

Figure 7  Asynchronous iterator client code

The different steps taken to execute the statement are shown below:

- (1) The first step is to configure the connection parameters to access the DataPort server. For this task a VDBDataSource object can be defined using the connection parameters required, or the DataSourceLocator class can be used to locate these data from a name (see 5.4.1).
- (2) Once the data source has been created or located with the DataSourceLocator, a connection is made to support the communication with the server (see 5.4.2).
- (3) A VQLStatement object is obtained from the connection, which is then prepared with the statement to be executed.
- (4) Where the statement presents a variable parameter, VQLStatement defines a method to associate specific values with the parameters.
- (5) The statement is executed, obtaining an asynchronous iterator with the results. The asynchronous iterator also provides additional data relative to the statement executed (see 5.7.2).
- (6) Once statement execution has started, the object returned allows iterating on the results returned by the server. Using this type of client by default, each tuple is returned enclosed in a StandardRowVO object.
- (7) It is also possible to obtain informative or error messages sent by the server.
- (8) The server also provides data relative to the schema of the data returned.
- (9) The connection is closed in order to be returned to the connection pool.

5.3  GENERAL STRUCTURE OF THE VIRTUAL DATAPORT DEVELOPMENT API

Figure 8 shows the general structure of the DataPort API. The essential element is the VDBConnection class. To establish a connection with the management system a VDBDataSource object must be obtained from the metadata that connects to the server. Obtaining the VDBDataSource for a specific DataPort server from a name is simplified using the DataSourceLocator utility class.
Figure 8  General structure of the proprietary API

The VDBConnection class provides direct mechanisms for executing statements on the server, also indicating a class (that will extend Printer) to process the results obtained. To simplify the statement execution process the VQLStatement class can be used, which allows:

a) Use of a VQL statement containing parameters that will later be rewritten with specific values (in a manner similar to a JDBC PreparedStatement).
b) Obtaining an object that allows asynchronously iterating on the results returned by the server.
c) Use of the default implementation of a result-converter object (called StandardPrinter, which will return each tuple as a StandardRowVO-type object). See section 5.7.2 for more details.

5.4 VIRTUAL DATAPORT DATA SOURCES

5.4.1 Data Sources

A data source identifies a Virtual DataPort server. This is represented an object of the VDBDataSource class.

To define a VDBDataSource the following parameters must be specified:

- dataSourceName: A data source name.
- description: A data source description.
- dbURI: A connection string for the server (\//host:port/databaseName).
- userName: A user name to make a connection.
- userPwd: Password for the user.
- queryTimeout: Maximum time (in milliseconds) the user application will wait for the statement to end. This parameter is optional. If it is not indicated or its value is 0, then it waits the time necessary for the statement execution to end.
- **chunkTimeout**: Maximum time the user application will wait for a partial result to arrive. This is optional. If this time is exceeded, Virtual DataPort returns an empty partial result. If it is not specified (or 0), the server returns the full set of results together.

- **chunkSize**: Number of results that make up a partial result. This factor, together with the preceding one, defines the partial result return behavior of the manager. If Virtual DataPort has available this number of results, they will be returned to the client, even if chunkTimeout has not been exceeded. If not specified or 0, all the results of the statement are returned in one sole partial result.

The connection objects can be obtained from a `VDBDataSource` object by using the `getConnection` method. To reduce the time required to create connections when executing statements on the Virtual DataPort server the `VDBDataSource` class allows specifying the use of reusable connection pools. If a pool is to be used, its configuration can be indicated with the `PoolConfig` class specifying:

- **initSize**: Number of connections with which the pool will be initialized. A number of connections is established and created in "idle" state, ready to be used.

- **maxActive**: Maximum number of active connections that the pool can manage at the same time. (zero implies no limit)

- **maxIdle**: Maximum number of active connections that can remain idle in the pool without having to clear additional connections. (zero implies there is no limit)

For more information, see Javadoc documentation for the developer API [3].

### 5.4.1.1 Definition and Location of Data Sources

To simplify client creation Virtual DataPort provides a utility class, `com.denodo.vdb.vdbinterface.client.util.datasource.DataSourceLocator`, which allows a `VDBDataSource` to be obtained for a DataPort server from its name.

```java
VDBDataSource ds = DataSourceLocator.getDataSource(DATASOURCE_NAME);
```

In order to locate the `VDBDataSource`, it must have been registered first. There are three ways to do this:

- The `DataSourceLocator` class defines an `addDataSource` method that allows `VDBDataSource` objects created directly by the application to be registered.

- The system automatically registers those data sources which properties are accessible in a `.properties` file called `ConfigurationParameters.properties` (or any other name, if it is specified through the `-DconfFile` property of the JAVA virtual machine).

- The system automatically registers those data sources which properties are accessible through JNDI [6] in the format `vdbDataSource.<dataSourceName>.<propertyName>`. In particular, the properties must be exported in a JNDI server from the path `java:comp/env/`. This allows, for example, the data sources to be defined in the `web.xml` file of a J2EE Web application.

Figure 9 shows the name of all the properties along with their usual values.
# Data source configuration used by provided clients
to connect to Virtual DataPort.
vdbDataSource.clientDs.dbURI=localhost:9999/vdb
vdbDataSource.clientDs.userName=demo
vdbDataSource.clientDs.userPswd=demo
vdbDataSource.clientDs.queryTimeout=80000
vdbDataSource.clientDs.chunkTimeout=80000
vdbDataSource.clientDs.chunkSize=100

# Pool parameters. Non Mandatory.
vdbDataSource.clientDs.initSize=0
vdbDataSource.clientDs.maxActive=30
vdbDataSource.clientDs.maxIdle=20

# Uncomment to disable pool.
#vdbDataSource.clientDs.poolEnabled=false

Figure 9  VDBDataSource configuration

## 5.4.2 Virtual DataPort Connections

The VDBConnection class encapsulates a connection with a Virtual DataPort server. It represents a proxy to "Virtual DataPort" (for more information see the design pattern Proxy[4]). This makes a simple front-end available to clients to access the server.

The main methods of this class are:

- `exec(query, Printer)`: Executes a statement or query in the server. The second argument is a Printer class. This class, which will be explained later, will format the results received from the server according to the specific requirements of each client.

- `execDeferred(query, printer)`: this method also executes a statement using Virtual DataPort. The difference with the `exec(query, printer)` method is that `execDeferred(query, printer)` simply executes the statement and returns the control to the client without having requested any of the results. To request the results from the server, execute the method `getNextResult(printer)`.

- `getNextResult(Printer)`: Obtains the next partial result from the server. Again the Printer class will be in charge of assembling the result that the client will receive from the data returned from the server.

- `openClientConnection(user,pwd,url)`: Opens a connection with Virtual DataPort. This method throws an exception if any of the parameters is incorrect (e.g. the username or password are incorrect or a connection cannot be established with the server). This method is useful for clients that only want to validate the credentials typed in by the user (e.g. to check a login process).

- `cancel()`: Stops current execution of a statement. When canceling a query, the DataPort server will cancel all current accesses to the data sources and to the cache. After invoking `cancel`, it is possible that the server returns some results, if these were retrieved before the source access cancellation was effective. Therefore, even though a query might have been canceled, the client can still request more results to the server.
• getConnectionTime(): Provides the connection time.

• isCached(): Returns true if the connection has been obtained from the connections pool. False, if the connection has been created.

• close(): Closes the connection with the server (or returns it to the pool, if defined).

• createVQLStatement(): Creates a VQLStatement object associated with this connection object. This class allows executing parameterized statements and obtains the results by using an asynchronous iterator (see section 5.7.2).

There are other methods in this class that are useful for obtaining and establishing the configuration parameters of the Virtual DataPort server, i.e. get() and set() methods for the **userName**, **password**, **dbURI**, **queryTimeout**, **chunkTimeout** and **chunkSize**.

### 5.5 PROTOCOL OF COMMUNICATIONS WITH THE SERVER

Another relevant element for communicating with the server is the hierarchy of partial results ('chunks' received from the server as a response to the statement) that encapsulates the data exchanged.

![PartialResult](image)

**Figure 10** Superclass of all the partial results

The types of partial results a client can receive are the following:

• **EmptyResult**: partial result without any data. Issued by the server, when the client requests the results before any result not yet returned becomes available.

• **EndResult**: partial result issued by the server when a statement has ended. After receiving a message of this type, the client should not request more results of the statement to the server.

• **ErrorResult**: partial result issued by the server when an error occurs during a statement execution. This is also an end message. After receiving a message of this type, the client should not request more results of the statement to the server.

• **MessageResult**: informative message issued by the server. This is not an end message, and it may contain statement execution data. From the 4.0 version, a new message subtype called **WarningResult** is included. It is used to notify incidences occurred during the execution of a sentence.
• **MetadataResult**: message with the metadata of a SELECT, QUERY WRAPPER, or CALL statement issued using Virtual DataPort. The metadata of a query includes the name of each column together with its type (MetadataField object).

• **ChunkResult**: message that contains tuples resulting from a SELECT, QUERY WRAPPER or CALL statement issued through the server.

Likewise, the protocol established between the client and the server for sending and receiving messages follows the rules below:

• The number of fragments or partial results that can be returned as response to a statement execution is not fixed.

• The statement ends when an end message or error message arrives.

• A message can encapsulate another message (according to the design pattern decorator [4]). This minimizes the number of communications between the client and the server. This rule is not valid for error messages and empty messages.

• When the query executed in the server is a query against a view (SELECT), a query against a wrapper (QUERY WRAPPER) or the invocation of a stored procedure (CALL), the first non-empty message contains the metadata referring to the statement results (it indicates the columns that form the result together with their associated data types). Another informative message is sent (MessageResult) by decorating the metadata message, which encapsulates the statement internationalization configuration properties (I18nVO). A list of the internationalization property names it encapsulates can be found in the VQL Advanced Guide [2].

• An error message may contain an explanation message and a code that indicate the error type that has occurred (e.g. an internal error in the server, a user authentication error, etc. See the Javadoc documentation [3] of the class com.denodo.vdb.vdbinterface.common.clientResult.ErrorCodes for details of the possible error codes [3]).

The following diagram gives a more detailed description of the hierarchy of partial results.
As already mentioned, when a query statement (SELECT) has been issued to Virtual DataPort, the client obtains a ChunkResult object. Inside a ChunkResult object there is a group of tuples (ChunkRow), and each tuple contains a group of values (ValueVO). A value can be simple (SimpleVO), an array (ArrayVO) or a register (RegisterVO). The simple values (SimpleVO) can be of each of the data types supported (see [2]). For example: integers (IntVO), date (DateVO), etc.

Besides, a MessageResult can also encapsulate the response to a DESC or LIST statement (see [2]). In these cases, it will encapsulate an object of the type MessageVO that can respectively be a description of some element in the catalog (VDBObject), or a list of elements of the catalog (VDBObjectList). From the 4.0 version, a new message subtype called WarningResult is included. It is used to notify incidences occurred during the execution of a sentence.

A VDBObject object exists for each element in the Virtual DataPort catalog (wrapper, base or derived view, stored procedure, operator, type, etc. See [1] for more details on these elements) a description of which can be requested from the server using a DESC statement. There is also a VDBObject object to represent the execution trace of a statement (see section 5.7.3).

For more insight into the structure of partial results a client may receive see Javadoc documentation [3].

### 5.6 PROCESSING STATEMENT RESULTS

The requirements of the different client applications regarding the format required to receive the results of a statement on DataPort can differ considerably.

Therefore, the solution adopted in DataPort separates the communication logic from the assembly of partial results that are returned from the server. A new class hierarchy with the Printer class as superclass will be in charge of this task. This way, when a new client appears, it is enough to provide a new Printer subclass which transforms the results coming from the server.
A developer may create his/her own Printer classes or use some of the implementations included in Virtual DataPort which are suitable for most cases (com.denodo.vdb.vdbinterface.client.printer package):

- **ConsolePrinter**: Textually formats the results and sends them to an output stream (like, for example, the standard output).

- **standard.StandardPrinter**: Printer class that assembles the results of any statement issued using Virtual DataPort. The results and the associated execution plan are returned encapsulated in the standard.StandardTableVO value object. A StandardTableVO object defines methods for obtaining the result tuples from the manager (getTuples()), the schema of the data returned (getSchema()), the messages included in the response as a string (getMessage()) and the event trace associated with the query (getTrace()). Each tuple of the result will be returned as a StandardRowVO object that will contain a field for each attribute present in the response.

Please refer to Javadoc documentation [3] to obtain detailed information on the structure of the predefined Printer classes.
5.6.1 Implementation of New Result Assemblers

Figure 12 shows the methods that should be implemented to create a new Printer. All the message types that can be processed should first be defined and formatted: this is carried out by implementing a series of methods of the PartialResultVisitor interface. The following methods of the Printer class should also be implemented or redefined:

- **isFinished()**: returns true, when the printer determines that the statement has ended. Generally, this happens when an error message or end message arrives; although there is nothing which prevents a printer from ending a statement using other criteria.

- **setFinished(boolean)**: allows a statement to be ended manually. An example of use of this characteristic is clicking on a cancel button in a graphic tool window.

- **getclientResult()**: returns the result of the statement adapted to the format required by the client.

- **log(String)**: logs a message. Subclasses should redefine this method to redirect the messages to the appropriate output for each client. For example, a command line client should redirect these messages to the standard output and a graphic client should redirect the messages to a graphic window.

- **getErrorCode()**: if an error occurs during the execution of a statement, this method returns the type of the error. See Javadoc documentation [3] to obtain a list of the error codes and their description.

- **reset()**: restores class state to begin the processing of a new statement.

- **format(Exception)**: formats the errors that are not controlled and thus those that generate an exception. Exceptions can also require different processing depending on the client type. It is the printer that should be in charge of processing them.

As example, Figure 13 shows the implementation of the predefined Printer implementation ConsolePrinter that textually formats the results obtained and writes them in an output stream (the standard output by default).

```java
// Import needed for the log
import java.io.PrintStream;
import java.util.Iterator;
import com.denodo.vdb.vdbinterface.client.printer.Printer;
import com.denodo.vdb.vdbinterface.common.clientResult.ChunkResult;
import com.denodo.vdb.vdbinterface.common.clientResult.EmptyResult;
import com.denodo.vdb.vdbinterface.common.clientResult.EndResult;
import com.denodo.vdb.vdbinterface.common.clientResult.ErrorResult;
import com.denodo.vdb.vdbinterface.common.clientResult.MessageResult;
import com.denodo.vdb.vdbinterface.common.clientResult.MetadataField;
import com.denodo.vdb.vdbinterface.common.clientResult.MetadataResult;
import com.denodo.vdb.vdbinterface.common.clientResult.VQLException;
import com.denodo.vdb.vdbinterface.common.clientResult.vo.sentences.ChunkRow;
import com.denodo.vdb.vdbinterface.common.clientResult.vo.sentences.ValueVO;
```
public class ConsolePrinter extends Printer{
  private PrintStream _out=System.out;
  private String _metadata=null;
  public ConsolePrinter() {}
  public ConsolePrinter(PrintStream out){_out=out;}
  public void format(EmptyResult result) throws VQLException {
    log("Info: received new empty result");
  }
  public void format(EndResult result) throws VQLException {
    _out.println("_________________________");
    _out.println("Info: received end result");
    setFinished(true);
    if (result.getDecorated()!=null) {
      result.getDecorated().accept(this);
    }
  }
  public void format(ChunkResult result) throws VQLException {
    log("Info: received new chunk result");
    Iterator chunkRows=result.getResults().iterator();
while (chunkRows.hasNext()) {
    _out.print("| ");
    ChunkRow singleRow=(ChunkRow)chunkRows.next();
    Iterator values=singleRow.getValues().iterator();
    while (values.hasNext()) {
        _out.print((ValueVO)values.next() + " | ");
    }
    _out.println();
}
if (result.getDecorated()!=null) {
    result.getDecorated().accept(this);
}
/**
* Defines the behavior when a message with the metadata from a query arrives from the server.
* @param result Message with the metadata.
*/
public void format(MetadataResult result) throws VQLException {
    log("Info: received metadata result");
    _metadata=new String("[ ");
    Iterator fields=result.getFields().iterator();
    while (fields.hasNext()) {
        MetadataField field=(MetadataField)fields.next();
        _metadata=_metadata+(field.getName()) + " ";
    }
    _metadata=_metadata+"]\n";
    log(_metadata);
    if (result.getDecorated()!=null) {
        result.getDecorated().accept(this);
    }
}
/**
* Defines the behavior when an informative message arrives from the server.
* @param result Informative message.
*/
public void format(MessageResult result) throws VQLException {
    log("Info: received message result");
    if(result.getMessage()!=null)log(result.getMessage());
    if(result.getObject()!=null)log(result.getObject().toString());
    if (result.getDecorated()!=null) {
        result.getDecorated().accept(this);
    }
}
/**
* Defines the behavior when an error message arrives.
* @param result Error message.
*/
public void format(ErrorResult result) throws VQLException {
    setFinished(true);
    log(result.getMessage());
    if (result.getDecorated()!=null) {
        result.getDecorated().accept(this);
    }
}
* Defines the behavior when an exception is generated during a query.
* @param exception generated exception.
*/
public Object format(Exception exception) throws VQLException{
    log("error: "+ exception.getMessage());
    return null;
}
/**
* Forwards the message to the output stream.
*/
public void log(String message){
    _out.println(message);
}

Figure 13 ConsolePrinter code

5.7 GENERAL CLIENT OPERATING MODES
This section describes how to develop clients for the Virtual DataPort server using the JAVA native API.

There are two possible types of Virtual DataPort clients:

- **Synchronous client.** The client, after issuing a statement, remains blocked until all the results are obtained.

- **Asynchronous iterator client.** **Recommended option.** The client sends the VQL statement to the server and does not have to wait until the execution has finished.

Details of each of these client types are described in the following subsections.

5.7.1 Synchronous Clients
In this case, the client, after issuing a statement, remains blocked until it obtains all the results in a single block. These client types can be created very easily through the two actions below:

a) Implement the Printer that assembles the results (or use one of those already provided with Virtual DataPort. See section 5.6).

b) Invoke the exec(query,printer) method of the VDBConnection. This method sends the statement to the server and returns the results of the statement. An important aspect is that the application does not regain control until the communication process has completely finished.

An example of the steps to be followed to execute a statement against Virtual DataPort following this method is shown in Figure 14 (the predefined Printer ConsolePrinter is used):

```
// Sentence to execute against the server
String sentence = "SELECT * FROM @ecb";

// The printer for assembling query results must be chosen
Printer assembler = new ConsolePrinter(System.out);

try {
    // We get a connection to Virtual DataPort
```
VDBDataSource ds = DataSourceLocator.getDataSource("clientDs");
VDBConnection connection = ds.getConnection();

Object result = null;
// Query is executed. The format of the final result is determined by
// the printer.
result = connection.exec(sentence, assembler);
}
} catch (Exception ex){
ex.printStackTrace();
} finally {
// The connection must be closed.
if (vdpConnection != null) {
try {
 vdpConnection.close();
} catch (VDBConnectionException e) {
 System.err.println("Error closing connection ... ");
 e.printStackTrace();
}
}

Figure 14 Code of a synchronous client

5.7.2 Asynchronous Iterator Client

This type of client sends the VQL statement to the server and doesn’t have to wait until the execution of the
statement has finished. Instead, it obtains the results asynchronously. This is the recommended option to create a
DataPort client using the JAVA API.

This type of client obtains the results asynchronously and composes them through the assembler
StandardPrinter. A StandardTableVO object is generated internally with the results.

To develop a client that iterates over the results asynchronously, follow these steps:

1. Obtain a new VQLStatement from the current VDBConnection that encapsulates the
connections with the Virtual DataPort server: VDBConnection.createVQLStatement()

2. Invoke the method VQLStatement.setPreparedStatement(String) to set the VQL
query of the statement.

The VQLStatement can execute parameterized VQL statements and allows specifying their parameters
like the PreparedStatement class of the JDBC API [5]. These parameters are specified in the VQL
query with ?<parameterIdentifier>

If the values of the parameters are filled into the statement with the method
setParameters(List), the parameters have to be enumerated in the statement, starting with the
index 1. For example:
SELECT * FROM xyz WHERE a > ?1 and b < ?2.

If the values of the parameters are filled into the statement with the method setParameters(Map),
then they are replaced by the occurrences of ?<parameterIdentifier> in the statement.

3. Invoke the method VQLStatement.executeQuery().

This method returns a VDBResultSetIterator object that implements the interface
java.util.Iterator.
Each tuple is represented as a `StandardRowVO` object with a field for each attribute of the query output schema. The `hasNext()` method of the `VDBResultSetIterator` object must be used to iterate over the results. This is due to the fact that the asynchronous wait is implemented into this method, so that it waits until the data is available. The `next()` method of the same class launches a `NoSuchElementException` exception if there is no data available. Therefore, the `hasNext()` method must be used before.

Figure 7 shows an example of how this client type is fully implemented.

### 5.7.3 Information on Statement Execution Trace

The trace of a statement allows a detailed examination of its execution plan. This plan is displayed to the administrator in the form of a tree, where each node represents an intermediate view involved in the execution of a query or access to a source via a wrapper. The most relevant parameters are displayed for each node on the query execution tree. These parameters include:

- **Node type.** If the node is a view, this indicates the type of view (base view, union, join, projection, etc.). If it is an access to a source (wrapper), this indicates the type of data source (JDBC, Web Service, Web, etc.).
- **Execution time.** Time spent completely executing the node and all its children.
- **Start time.** The exact moment at which node processing begins in the execution plan.
- **End of query time.** The exact moment at which node processing (and that of all its children) ends in the execution plan.
- **Time obtaining the first tuple of results.** Time spent until the node receives the first tuple to be processed.
- **Number of tuples processed.** Number of tuples processed by the node.
- **Status.** This indicates whether the node was correctly executed or whether an error occurred.
- **Advanced parameters.** These provide further details on each node type. For example:
  - In the case of wrapper-type nodes, it indicates the exact sub-queries executed on each data source and the connection data used to access them.
  - For each view-type node, it is indicated whether the cache has been used, whether swapping has been necessary, etc.

One of the main utilities of the trace function is debugging. The Virtual DataPort administration tool can be used to access these data graphically (see the Virtual DataPort Administration Guide [1] for further information).

The Virtual DataPort API provides access to the execution plan trace information. To do so, an instance of the `com.denodo.vdb.vdbinterface.common.clientResult.vo.descriptions.queryplan.ExecutionPlanVO` class must be obtained using the public `ExecutionPlanVO getExecutionPlan ()` method of the `com.denodo.vdb.vdbinterface.client.util.iterator.VDBResultSetIterator` class.

This class includes a series of methods to access the execution plan information such as the name of the view being accessed, the number of tuples that this specific view has returned, whether the view has completed its data extraction process, etc., as previously indicated (for further information, see the Javadoc documentation [3]).

The `List getSubExecutionPlans ()` method returns the child nodes of the corresponding parent node, which allows you to browse the tree structure of the execution plan to the data sources themselves (data sources do not have children, and therefore this method returns the null value when invoked on data source nodes).

It is important to remember that, for the `getExecutionPlan ()` method to return an instance of the class encapsulating the execution plan, the VQL query must include the “TRACE” clause (e.g. “select * from view trace”).
Depending on the node type (view, wrapper, stored procedure, etc.) there are different subclasses of `ExecutionPlanVO`, which include additional information parameters (for further information, see the description of the classes in the JavaDoc documentation [3]):

- `com.denodo.vdb.vdbinterface.common.clientResult.vo.descriptions.queryplan.vdb.IDUAccessPlanVO` for insert, update and/or delete plans,
- `com.denodo.vdb.vdbinterface.common.clientResult.vo.descriptions.queryplan.StoredProcedurePlanVO` for stored procedures,
- `com.denodo.vdb.vdbinterface.common.clientResult.vo.descriptions.queryplan.vdb.VDBAccessPlanVO` for views,
6  APPENDIX

6.1  OUTPUT SCHEMA OF THE LIST COMMAND

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.

6.2  OUTPUT SCHEMA OF THE DESC COMMAND DEPENDING ON ITS PARAMETERS

<table>
<thead>
<tr>
<th>Command</th>
<th>Result column names</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESC DATABASE</td>
<td>name&lt;br&gt;description</td>
</tr>
<tr>
<td>DESC DATASOURCE ARN</td>
<td>name&lt;br&gt;aracne server route&lt;br&gt;user name</td>
</tr>
<tr>
<td>DESC DATASOURCE CUSTOM</td>
<td>name&lt;br&gt;class name&lt;br&gt;classpath&lt;br&gt;jars</td>
</tr>
<tr>
<td>DESC DATASOURCE DF</td>
<td>name&lt;br&gt;data delimiter&lt;br&gt;end delimiter&lt;br&gt;column delimiter&lt;br&gt;end of line delimiter&lt;br&gt;header pattern&lt;br&gt;tuple pattern&lt;br&gt;data route</td>
</tr>
<tr>
<td>DESC DATASOURCE GS</td>
<td>name&lt;br&gt;google mini server&lt;br&gt;proxy host&lt;br&gt;proxy port&lt;br&gt;proxy user</td>
</tr>
<tr>
<td>DESC DATASOURCE JDBC</td>
<td>name&lt;br&gt;database uri&lt;br&gt;driver class name&lt;br&gt;user name&lt;br&gt;classpath&lt;br&gt;database name&lt;br&gt;database version&lt;br&gt;ping query&lt;br&gt;initial size&lt;br&gt;max active</td>
</tr>
<tr>
<td>DESC DATASOURCE JSON</td>
<td>name&lt;br&gt;data route</td>
</tr>
<tr>
<td>DESC DATASOURCE LDAP</td>
<td>DESCRIPITION</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>uri</td>
<td></td>
</tr>
<tr>
<td>login</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC DATASOURCE ODBC</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>dsn</td>
<td></td>
</tr>
<tr>
<td>database uri</td>
<td></td>
</tr>
<tr>
<td>driver class name</td>
<td></td>
</tr>
<tr>
<td>user name</td>
<td></td>
</tr>
<tr>
<td>classpath</td>
<td></td>
</tr>
<tr>
<td>database name</td>
<td></td>
</tr>
<tr>
<td>database version</td>
<td></td>
</tr>
<tr>
<td>ping query</td>
<td></td>
</tr>
<tr>
<td>initial size</td>
<td></td>
</tr>
<tr>
<td>max active</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC DATASOURCE WS</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>wsdl route</td>
<td></td>
</tr>
<tr>
<td>authentication user</td>
<td></td>
</tr>
<tr>
<td>authentication type:</td>
<td></td>
</tr>
<tr>
<td>{ No authentication - 'Authentication user’ is empty, 1 = HTTP Basic authentication, 2 = ‘HTTP Digest authentication’, 3 = ‘NTLM authentication’, 10 = ‘WSS Basic authentication’, 11 = ‘WSS Digest authentication’ }</td>
<td></td>
</tr>
<tr>
<td>proxy host</td>
<td></td>
</tr>
<tr>
<td>proxy port</td>
<td></td>
</tr>
<tr>
<td>proxy user</td>
<td></td>
</tr>
</tbody>
</table>

Passwords are not included for security purposes.

<table>
<thead>
<tr>
<th>DESC DATASOURCE XML</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>data route</td>
<td></td>
</tr>
<tr>
<td>schema route</td>
<td></td>
</tr>
<tr>
<td>dtd route</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC MAP</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
</tbody>
</table>

A row for entry of the map

<table>
<thead>
<tr>
<th>DESC PROCEDURE</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>type (parameter type)</td>
<td></td>
</tr>
<tr>
<td>direction = { IN</td>
<td>OUT } : input or output parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC SESSION</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC TYPE</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
</tbody>
</table>

If the type is complex it returns a row for each component of the type. If it is simple {ej: text}, only the Name of the type.

<table>
<thead>
<tr>
<th>DESC USER</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td></td>
</tr>
<tr>
<td>admin (true if the user is an ‘Administrator’. False otherwise)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC VIEW</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldname</td>
<td></td>
</tr>
<tr>
<td>fieldtype (a row for each field of the view)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESC WEBSERVICE</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsname</td>
<td></td>
</tr>
<tr>
<td>wstypes</td>
<td></td>
</tr>
<tr>
<td>operationtype = { 1 = SELECT, 10 = INSERT, 11 = UPDATE, 12 = DELETE }</td>
<td></td>
</tr>
<tr>
<td>operationname</td>
<td></td>
</tr>
<tr>
<td>fieldname</td>
<td></td>
</tr>
<tr>
<td>fieldtype</td>
<td></td>
</tr>
<tr>
<td>fielddirection (input or output parameter)</td>
<td></td>
</tr>
</tbody>
</table>

Returns a row for each parameter of the Web Service.

<table>
<thead>
<tr>
<th>DESC WRAPPER ARN</th>
<th>DESCRIPITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>type = ‘arn’</td>
<td></td>
</tr>
<tr>
<td>datasource name</td>
<td></td>
</tr>
<tr>
<td>handler name</td>
<td></td>
</tr>
<tr>
<td>filter main terms</td>
<td></td>
</tr>
<tr>
<td>output schema</td>
<td></td>
</tr>
</tbody>
</table>
| DESC WRAPPER CUSTOM | name
|---|---
type = 'custom'
datasource name
parameters
output schema
| DESC WRAPPER DF | name
|---|---
type = 'df'
datasource name
output schema
| DESC WRAPPER GS | name
|---|---
type = 'gs'
datasource name
client
languages
site collections
number of key match
output schema
| DESC WRAPPER ITP | name
|---|---
type = 'itp'
creation date
maintenance
old
sequence
substitutions
model content
scanners
output schema
| DESC WRAPPER { JDBC | ODBC } | name
|---|---
type = { 'jdbc' | 'odbc' }
data source name
relation name
sql sentence
aliases
output schema
| DESC WRAPPER JSON | name
|---|---
type = 'json'
datasource name
tuple root
output schema
| DESC WRAPPER LDAP | name
|---|---
type = 'ldap'
datasource name
object classes
recursive search
ldap expression
output schema
| DESC WRAPPER WS | name
|---|---
type = 'ws'
datasource name
service name
port name
operation name
input message
output message
output schema
| DESC WRAPPER XML | name
|---|---
type = 'xml'
datasource name
tuple root
output schema

Figure 15  Output schema of the DESC command depending on its parameters
6.3 ACCESS THROUGH AN ODBC-JDBC BRIDGE

Virtual DataPort has a ODBC interface for ODBC-compliant applications (see section 3). But alternatively, ODBC applications can use an ODBC-JDBC bridge to access Virtual DataPort through the JDBC interface.

This section explains how to configure two of the most popular ODBC/JDBC bridges of the market: EasySoft ODBC/JDBC Gateway [11] (successfully tested with versions 1.3.1 and 2.1) and OpenLink ODBC/JDBC bridge [12] (successfully tested with version 6.0).

6.3.1 Access through Easysoft ODBC/JDBC Gateway

The steps to access Virtual DataPort using EasySoft ODBC/JDBC Gateway are:

1. Install EasySoft ODBC/JDBC Gateway (see instructions in [11]).

2. Create a new ODBC DSN (Data Source Name) using EasySoft ODBC/JDBC Gateway.
   a. Go to Administrative Tools in the Windows Control Panel.
   b. Select ‘ODBC data sources’.
   c. Click on the ‘System DSN’ panel and click ‘Add’.
   d. Select Easysoft ODBC/JDBC Gateway and click on “End”.

3. Complete the next window as indicated below (see Figure 16):
   a. Specify the user name and password to be used in the DataPort server in the ‘User Name’ and ‘Password’ fields.
   b. The fields ‘Driver Class’, ‘Classpath’ and ‘URL’ must be completed with the corresponding values from the Virtual DataPort JDBC driver (see section 2): JDBC driver class, the path to the Virtual DataPort JDBC driver and access URL to a Virtual DataPort server.

4. Click on ‘Test’ to check whether connection can be made to the DataPort server.

5. Finally, click on ‘Ok’ to complete the process.
6.3.2 Access through Openlink ODBC/JDBC Bridge

The steps to access Virtual DataPort using OpenLink ODBC/JDBC Bridge (called ODBC/JDBC-Lite) are:

1. Install OpenLink ODBC/JDBC-Lite (see instructions in [12]).
   a. The installation process may modify the present CLASSPATH and, therefore, it is advised to check it after installation. The DataPort JDBC driver should be included in the CLASSPATH.
   b. Make sure that the jvm.dll file (normally found in the path %JAVA_HOME%/jre/bin/server) is in the system PATH.

2. Create a new ODBC data source using OpenLink ODBC/JDBC-Lite.
   a. Go to Administrative Tools in the Control Panel.
   b. Select ‘ODBC data sources’.
   c. Click on the ‘System DSN’ panel and click ‘Add’.
   d. Select OpenLink ODBC/JDBC-Lite and click on end.

3. In the next window, enter the name and a description for the new data source and click on ‘Next’.

4. Complete the next window as indicated below (see Figure 17):
a. Specify the user name and password to be used in the DataPort server in the 'Login ID' and 'Password' fields.

b. The 'JDBC Driver' and 'URL string' fields must be completed with the corresponding values from the Virtual DataPort JDBC driver (see section 2): JDBC driver class and access URL to a DataPort server.

5. The next windows of the wizard allow configuring different use options for the data source. Mark the following options in the first window (see Figure 18):
   a. 'Drop Catalog Name from Database Metadata Calls'.
   b. 'Disable support of quoted identifier'

6. Once the configuration is complete, a window will be accessed that allows the new data source to be tested using the 'Test Data Source' button.

7. If the test is successful, click on 'Finish' to complete the process.

![Figure 17 Configuration of OpenLink ODBC/JDBC Lite (1)](image-url)
Figure 18 Configuration of OpenLink ODBC/JDBC Lite (2)
REFERENCES