NOTE
This document is confidential and is the property of Denodo Technologies (hereinafter Denodo).
No part of the document may be copied, photographed, transmitted electronically, stored in a document management system or reproduced by any other means without prior written permission from Denodo.
# INDEX

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>I</td>
</tr>
<tr>
<td>SCOPE</td>
<td>I</td>
</tr>
<tr>
<td>WHO SHOULD USE THIS DOCUMENT</td>
<td>I</td>
</tr>
<tr>
<td>SUMMARY OF CONTENTS</td>
<td>I</td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>2</td>
</tr>
<tr>
<td>2  GENERAL OVERVIEW OF VIRTUAL DATAPORT</td>
<td>3</td>
</tr>
<tr>
<td>2.1 CREATING OR DEFINING DATA</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Defining Base Relations</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 Defining Data Sources and Wrappers</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Defining the Views of the Global Schema</td>
<td>6</td>
</tr>
<tr>
<td>2.2 EXECUTING STATEMENTS</td>
<td>7</td>
</tr>
<tr>
<td>3  LANGUAGE FOR DEFINING AND PROCESSING DATA: VQL</td>
<td>8</td>
</tr>
<tr>
<td>3.1 DATA TYPES</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Internationalization</td>
<td>9</td>
</tr>
<tr>
<td>3.2 STATEMENTS</td>
<td>9</td>
</tr>
<tr>
<td>3.3 SELECT STATEMENT: CLAUSES</td>
<td>10</td>
</tr>
<tr>
<td>3.4 INSERT / UPDATE /DELETE: CLAUSES</td>
<td>11</td>
</tr>
<tr>
<td>3.5 LOGICAL OPERATORS</td>
<td>11</td>
</tr>
<tr>
<td>3.6 COMPARISON OPERATORS</td>
<td>12</td>
</tr>
<tr>
<td>3.7 FUNCTIONS FOR CONDITIONS AND DERIVED ATTRIBUTES</td>
<td>15</td>
</tr>
<tr>
<td>3.7.1 Arithmetic Functions</td>
<td>16</td>
</tr>
<tr>
<td>3.7.2 Text Processing Functions</td>
<td>17</td>
</tr>
<tr>
<td>3.7.3 Date Processing Functions</td>
<td>20</td>
</tr>
<tr>
<td>3.7.4 Type Conversion Functions</td>
<td>22</td>
</tr>
<tr>
<td>3.7.5 XML Processing Functions</td>
<td>22</td>
</tr>
<tr>
<td>3.7.6 Other Functions</td>
<td>24</td>
</tr>
<tr>
<td>3.7.7 Aggregation Functions</td>
<td>26</td>
</tr>
<tr>
<td>3.8 SYNTAX CONVENTIONS</td>
<td>27</td>
</tr>
<tr>
<td>3.8.1 Syntax of Functions and Condition Values</td>
<td>28</td>
</tr>
<tr>
<td>4  CREATING A BASE RELATION (OR BASE VIEW)</td>
<td>31</td>
</tr>
<tr>
<td>4.1 MODIFYING A BASE VIEW</td>
<td>31</td>
</tr>
<tr>
<td>4.2 QUERY CAPABILITIES: SEARCH METHODS AND WRAPPERS</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1 Query Constraints</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2 Assigning Wrappers to Search Methods</td>
<td>34</td>
</tr>
<tr>
<td>4.2.3 Example of How a Search Method is Created</td>
<td>34</td>
</tr>
<tr>
<td>5  QUERIES: SELECT STATEMENT</td>
<td>36</td>
</tr>
<tr>
<td>5.1 FROM CLAUSE</td>
<td>38</td>
</tr>
<tr>
<td>5.1.1 Join Operations</td>
<td>39</td>
</tr>
<tr>
<td>5.1.2 Flatten View (Flattening Data Structures)</td>
<td>40</td>
</tr>
<tr>
<td>5.2 SELECT CLAUSE</td>
<td>41</td>
</tr>
<tr>
<td>5.2.1 Derived Attributes</td>
<td>41</td>
</tr>
<tr>
<td>5.3 WHERE CLAUSE</td>
<td>41</td>
</tr>
<tr>
<td>5.3.1 Conditions with Compound Values</td>
<td>42</td>
</tr>
</tbody>
</table>
# Table of Contents

5.4 **GROUP BY CLAUSE** .................................................................................................................... 42  
5.4.1 Use of Aggregation Functions.............................................................................................. 43  
5.5 **HAVING CLAUSE** ..................................................................................................................... 43  
5.6 **UNION CLAUSE** .......................................................................................................................... 43  
5.6.1 Specifying Projections in UNION Queries ............................................................................. 44  
5.7 **ORDER BY CLAUSE** ................................................................................................................... 44  
5.8 **OFFSET AND FETCH** .............................................................................................................. 44  
5.9 **CONTEXT CLAUSE** ................................................................................................................... 45  
5.10 **TRACE CLAUSE** ....................................................................................................................... 47  
5.11 **CASE CLAUSE** .......................................................................................................................... 50  

6 **DEFINING A DERIVED VIEW** .......................................................................................................... 51  
6.1 Modifying a Derived View ........................................................................................................... 52  

7 **INSERTIONS, UPDATES AND DELETION OF VIEWS** .................................................................. 53  
7.1 **INSERT STATEMENT** .................................................................................................................. 53  
7.2 **UPDATE STATEMENT** ............................................................................................................... 54  
7.3 **DELETE STATEMENT** ................................................................................................................. 55  
7.4 **USE OF THE WITH CHECK OPTION** ....................................................................................... 55  

8 **TRANSACTIONS IN VIRTUAL DATAPORT** ................................................................................... 57  

9 **STORED PROCEDURES** ................................................................................................................. 58  
9.1 Importing a Stored Procedure ..................................................................................................... 58  
9.2 Use of Stored Procedures ............................................................................................................. 58  
9.3 Pre-defined Procedures .................................................................................................................. 59  

10 **DEFINING OTHER ELEMENTS OF THE CATALOG** ..................................................................... 62  
10.1 Defining a Data Type .................................................................................................................. 62  
10.2 Defining a Map ........................................................................................................................... 63  
10.3 Defining .JAR Extensions ........................................................................................................... 64  
10.4 Defining JMS Listeners ............................................................................................................... 64  

11 **CREATING DATABASES, USERS AND ACCESS LEVELS** ............................................................ 67  
11.1 Databases in Virtual Dataport ..................................................................................................... 67  
11.2 User and Access Structures in Virtual Dataport .......................................................................... 67  
11.2.1 Types of Users ....................................................................................................................... 67  
11.2.2 Types of Access Rights ......................................................................................................... 67  
11.3 VQL Statements of Databases, Users and Privileges .................................................................. 69  
11.3.1 Creating Databases ............................................................................................................... 69  
11.3.2 Modifying and Deleting Databases ....................................................................................... 69  
11.3.3 Creating Users ....................................................................................................................... 70  
11.3.4 Modifying and Deleting Users ............................................................................................. 71  
11.3.5 Changing the Active Database ............................................................................................. 72  
11.3.6 Modifying the Privileges of a User ....................................................................................... 72  

12 **DESCRIPTING CATALOG ELEMENTS** .......................................................................................... 76  
12.1 Exporting Metadata .................................................................................................................... 78  
12.2 Importing Metadata .................................................................................................................... 79
13 LISTING ELEMENTS IN THE CATALOG ................................................................. 81

14 REMOVING ELEMENTS FROM THE CATALOG ...................................................... 83

15 PUBLICATION OF WEB SERVICES ........................................................................ 85
   15.1 CREATION OF NEW WEB SERVICES ............................................................... 85
   15.2 WEB SERVICES AUTHENTICATION ................................................................. 89
      15.2.1 Basic and Digest ......................................................................................... 89
      15.2.2 Basic LDAP ............................................................................................... 89
      15.2.3 WSS ........................................................................................................ 90
      15.2.4 VDP ........................................................................................................ 90
   15.3 EMBEDDED WEB CONTAINER MANAGEMENT .................................................. 90
   15.4 DEPLOYMENT AND EXPORTING OF WEB SERVICES ...................................... 90

16 PUBLICATION OF WIDGETS ................................................................................ 92
   16.1 CREATE NEW WIDGETS .................................................................................. 92
   16.2 EXPORT A WIDGET ....................................................................................... 93
   16.3 DEPLOYMENT AND EXPORT OF AUXILIARY WEB SERVICES ......................... 94

17 HELP COMMAND .................................................................................................. 96

18 GENERATING WRAPPERS AND DATA SOURCES ............................................... 98
   18.1 VALID CONVERSIONS BETWEEN TYPES IN WRAPPERS AND VDP TYPES ........ 99
      18.1.1 Native-type Conversions of a Wrapper to Java Types .................................. 100
   18.2 SPECIFYING PATHS IN VIRTUAL DATAPORT ................................................... 103
      18.2.1 Filters ..................................................................................................... 104
   18.3 CREATING DATA SOURCES .......................................................................... 105
      18.3.1 JDBC Data Sources .................................................................................. 105
      18.3.2 ODBC Data Sources ................................................................................. 107
      18.3.3 Multidimensional Data Sources ............................................................... 109
      18.3.4 Data Sources for Web Services ............................................................... 110
      18.3.5 XML Data Sources .................................................................................. 113
      18.3.6 JSON Data Sources ................................................................................ 115
      18.3.7 DF Data Sources ..................................................................................... 117
      18.3.8 Denodo Aracne Data Sources ................................................................. 119
      18.3.9 Google Mini Data Sources ..................................................................... 120
      18.3.10 LDAP Data Sources ............................................................................. 121
      18.3.11 BAPI Data Sources ................................................................................ 121
      18.3.12 Custom Data Sources ............................................................................ 122
      18.3.13 Data Source Configuration Properties ................................................... 123
   18.4 CREATING WRAPPERS .................................................................................. 128
      18.4.1 Execution Context and Interpolation Strings ............................................. 128
      18.4.2 Wrapper Metadata ................................................................................... 128
      18.4.3 JDBC Wrappers ....................................................................................... 129
      18.4.4 Multidimensional Databases Wrappers ................................................... 132
      18.4.5 ODBC Wrappers .................................................................................... 133
      18.4.6 WWW Wrappers .................................................................................... 134
      18.4.7 Web Services Wrappers ......................................................................... 139
      18.4.8 XML Wrappers ....................................................................................... 141
      18.4.9 JSON Wrappers ..................................................................................... 142
      18.4.10 DF wrappers .......................................................................................... 143
      18.4.11 Denodo Aracne Wrappers ...................................................................... 145
FIGURES

Figure 1  Search form of a virtual bookshop on the Internet ................................................................. 4
Figure 2  Search method for a bookshop ............................................................................................... 5
Figure 3  Basic primitives for specifying VQL statements .................................................................... 28
Figure 4  Syntax of the statement CREATE TABLE .............................................................................. 29
Figure 5  Syntax of the statement ALTER TABLE .............................................................................. 31
Figure 6  Example of creating a base view ........................................................................................... 31
Figure 7  Example of how a search method is created with ALTER TABLE ........................................ 34
Figure 8  Syntax of the SELECT statement ......................................................................................... 38
Figure 9  Syntax of the DEPLOY, UNDEPLOY and EXPORT statements .................................................. 91
Figure 10  Syntax of the WEBCONTAINER statement ........................................................................ 90
Figure 11  Syntax of the CREATE WEBSERVICE statement ............................................................... 87
Figure 12  Syntax of the CREATE TYPE syntax .................................................................................. 74
Figure 13  Syntax of the CALL statement ........................................................................................... 59
Figure 14  Syntax of the statement ALTER VIEW .............................................................................. 52
Figure 15  Syntax of the CREATE MAP .................................................................................................. 63
Figure 16  Syntax of the ALTER USER statement .............................................................................. 71
Figure 17  Syntax of the ALTER DATABASE statement ........................................................................ 69
Figure 18  Syntax for a projection of the result of an union ................................................................... 44
Figure 19  Syntax of the SELECT statement ......................................................................................... 38
Figure 20  Syntax of the INSERT statement .......................................................................................... 53
Figure 21  Syntax of the DELETE statement .......................................................................................... 55
Figure 22  Syntax of the CREATE PROCEDURE statement ................................................................. 58
Figure 23  Syntax of the CREATE JAR statement ............................................................................... 64
Figure 24  Syntax of the CREATE DATABASE statement ................................................................. 69
Figure 25  Syntax of how a view is defined in accordance with others .................................................. 51
Figure 26  Syntax of the CONTEXT clause ......................................................................................... 46
Figure 27  Syntax of the EXECUTE clause ......................................................................................... 49
Figure 28  Syntax of the UPDATE statement ........................................................................................ 54
Figure 29  Syntax of the procedure CREATE VIEW ............................................................................ 31
Figure 30  Syntax of the statement CREATE VIEW ............................................................................. 31
Figure 31  Syntax of the procedure CREATE DATABASE ..................................................................... 69
Figure 32  Syntax of the procedure CREATE USER statement ........................................................... 71
Figure 33  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 34  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 35  Example of how a search method is created with ALTER TABLE ........................................ 34
Figure 36  Syntax of the procedure ALTER DATABASE statement .................................................... 69
Figure 37  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 38  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 39  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 40  Syntax of how a search method is created with ALTER TABLE ........................................... 34
Figure 41  Syntax of the procedure CREATE DATABASE statement .................................................. 69
Figure 42  Syntax of the procedure CREATE USER statement ............................................................ 71
Figure 43  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 44  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 45  Syntax of the procedure CREATE DATABASE statement .................................................. 69
Figure 46  Syntax of the procedure CREATE USER statement ............................................................ 71
Figure 47  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 48  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 49  Syntax of the procedure CREATE DATABASE statement .................................................. 69
Figure 50  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 51  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 52  Syntax of the procedure CREATE DATABASE statement .................................................. 69
Figure 53  Syntax of the procedure ALTER DATABASE statement .................................................... 70
Figure 54  Syntax of the procedure ALTER USER statement ............................................................ 71
Figure 112 Syntax for creating an LDAP wrapper........................................................................... 152
Figure 113 Syntax for modifying an LDAP wrapper................................................................. 152
Figure 114 Syntax of the command to create BAPI wrappers: CREATE WRAPPER SAPERP .................. 153
Figure 115 Syntax of the command to modify BAPI wrappers: ALTER WRAPPER SAPERP ................. 153
Figure 116 Syntax of the CUSTOM wrapper creation statement.............................................. 153
Figure 117 Example of creating a CUSTOM............................................................................. 154
Figure 118 Syntax of the CUSTOM wrapper update ............................................................... 154
Figure 119 DF Wrapper configuration example with SOURCECONFIGURATION.............................. 156
Figure 120 Syntax of the QUERY WRAPPER statements......................................................... 156
Figure 121 Syntax of the WWW QUERY WRAPPER statements ............................................. 156
Figure 122 Trees of compound elements ............................................................................... 159
Figure 123 Tuple with compound elements .......................................................................... 160
Figure 124 Tree of Compound-type values ........................................................................... 160
Figure 125 Creating a base relation with compound types.................................................... 160
Figure 126 Creating a wrapper with compound types ............................................................ 160
Figure 127 Adding a search method with compound types ..................................................... 161
Figure 128 QUERYPLAN syntax............................................................................................... 164
Figure 129 Definition tree for view V5.................................................................................. 165
Figure 130 Example of function without annotations with return type depending on the input ...... 172
Figure 131 Example of aggregation function using annotations ............................................ 172
Figure 132 Internationalization configuration es_euro .......................................................... 180
Figure 133 Syntax of the function GETVAR............................................................................ 182
Figure 134 Definition of a view with a variable in the selection condition (GETVAR) .................. 182
Figure 135 Invoking a view defined with a variable in the selection condition............................ 182
Figure 136 Syntax of the function SETVAR............................................................................ 182
Figure 137 Invoking a view defining a variable in the selection condition .................................... 182
Figure 138 Definition of a view with a variable in the selection condition (GETVAR) .................. 184
Figure 139 Invoking a view defined with a variable in the selection condition............................ 184
Figure 140 Invoking a view defining a variable in the selection condition .................................... 185
TABLES

Type conversions permitted with the CAST function ......................................................... 22
Available authentication methods for the Denodo Web services ........................................ 89
Automatic conversions between JAVA types and Virtual DataPort types ........................... 99
Other valid conversions between JAVA types and Virtual DataPort types ......................... 99
Type Conversion Tables for JDBC Wrappers .................................................................. 100
ITPilot-type conversions ............................................................................................... 101
Type Conversion Table for Web Services Wrappers ..................................................... 101
Type Conversion Table for XML Wrappers ................................................................... 102
Equivalency between Java and Virtual DataPort data types ....................................... 169
Reserved Characters for Date Format ............................................................................ 179
Java Date and time patterns used in VirtualDataPort .................................................... 237
PREFACE

SCOPE
This document provides an overview of Virtual DataPort from the perspective of the experienced administrator.

WHO SHOULD USE THIS DOCUMENT
This document is aimed at administrators and developers that require an in-depth knowledge of how all the administration activities of a Virtual-DataPort-based integration solution are executed. It incorporates a description of activities such as wrapper definition, the creation of relational views using base relations or map specification on integrated fields. The detailed information required to install the system or develop applications using APIs is provided in other manuals that will be referenced when needed.

SUMMARY OF CONTENTS
More specifically, this document:

- Describes some important characteristics of Virtual DataPort which the reader must be aware of in order to understand the rest of the document.

- Provides a general overview of the VQL language.

- Gives a detailed description of how to execute the different operation tasks on the Virtual DataPort server, i.e. how the catalog elements are defined and modified and how queries and updates are made to the server.
1 INTRODUCTION

Virtual DataPort is a global solution for heterogeneous and dispersed data source integration in real time.

Virtual DataPort uses VQL (Virtual Query Language) as Data Definition and Data Manipulation Language. VQL allows creating and updating the elements that constitute the system catalog, as well as querying and updating the unified information views built through using DataPort. VQL is highly compatible with SQL but it also includes specific constructions to deal with the particularities of a virtual information integration system in real time.

This document describes VQL and how to use it to perform Virtual DataPort administration tasks. It is important to remark that it is not usually needed to write VQL scripts manually. The Virtual DataPort Administration Guide [ADMIN_GUIDE] describes how to use the graphical tools to perform the most usual administration tasks.
2 GENERAL OVERVIEW OF VIRTUAL DATAPORT

This section briefly introduces the stages involved in creating an information integration application using Virtual DataPort.

In this section we will assume that the application will be created by manually writing VQL sentences. This is not the usual case: the Virtual DataPort Administration Guide [ADMIN_GUIDE] describes how to use the graphical tools to perform the most usual administration tasks. **Using the Graphical administration tool mode is strongly recommended for administration tasks.**

Two phases can be distinguished in the Virtual DataPort operation: a data creation or definition phase and a query and/or update execution phase. In the first phase the system unified data model is defined: data sources are imported and unified views combining source information are created. The second phase, the query and/or update execution one, constitutes normal operation of the system in which statements on views expressed in VQL are accepted and resolved extracting and combining data from the sources, and modifying information of them.

The next sub-sections deal, respectively, with each one of these phases.

2.1 CREATING OR DEFINING DATA

This section describes the data creation phase where the objective is defining the views or relations which constitute the global schema of Virtual DataPort. All the tasks involved in the creation and administration of schemas within the DataPort server are described in detail in subsequent sections of this document. Only a general overview of the process is provided in this section.

To define data, it is necessary to first define the **base relations** that are to represent the external sources supplying the data. To do so, a **wrapper** must be specified for the base relation, the purpose of which will be to extract data from the source and interpret the results it returns. Depending on the type of source used, a **data source** may have to be created prior to the **wrappers**. The **data sources** encapsulate data to access a specific source so that it can be reused by the different **wrappers** acting on it.

Once the base relations can access the source data, the views comprising the global schema will be defined through the composition and combination of the base relations.

Below is a brief description of these operations.

2.1.1 Defining Base Relations

Each source in the system is modeled as a series of base relations exported by same. Each base relation is composed of a series of attributes in a manner similar to a table in a conventional relational database.

Each attribute of a relation belongs to a **data type**. The type of a specific attribute delimits what query operators can be applied to it and certain constraints that the elements of this type should comply with. Some types of normal data supported by Virtual DataPort are: character strings, integers, currencies, dates, etc. Also supported are array-type data (to represent multi-valued data) and register (to represent register-type data). By combining these two data types, hierarchical data structures can also be easily represented in the unified data model.

In addition, each base relation explicitly describes its query capabilities through the aforementioned **search methods**. This is necessary because some data sources (e.g. web sources or web services) do not allow any query to be made.
on its data, instead limited interfaces are presented for these purposes (e.g. HTML forms or operations defined through a Web Service). If a relation has no search method, no query can be made on it.

Each search method is composed of a series of 5-uples. Each 5-uple represents a constraint that a specific query should comply with so that it can be executed on the source using this search method.

The format of a 5-uple is \( (\text{attribute}, \text{operators}, \text{obligatoriness}, \text{multiplicity}, \text{possible_values}) \) where:

- **attribute** is an attribute of the relation.
- **operators** is the group of operators that can be used in the queries on this source and with this search method. \( \text{ANY} \) represents any operator allowed by the attribute data type.
- **obligatoriness** can have three values: \( \text{OBL} \) indicates that the attribute should obligatorily appear in any query on the source. \( \text{OPT} \) indicates that the attribute may or may not appear in the query (it is optional) and \( \text{NOS} \) indicates that the queries for this attribute are not allowed in the source.
- **multiplicity** indicates how many values can be included in the source query for the attribute and the given operator. If it is not possible to make queries for this attribute ('\( \text{NOS} \)' value in the obligatoriness field), the value is necessarily 0. '+' indicates a number of values greater than 0 but without an upper limit.
- **possible_values** is the list of values that can be used to query the attribute. If the value '\( \text{ANY} \)' is contained in it, this means that the search range is not limited (within the range associated with the attribute data type) and the attribute can be queried using any value. If the obligatoriness field is fixed in the 5-uple to the '\( \text{NOS} \)' value, then it necessarily takes the value of an empty group.

Any query will normally be permitted in sources such as relational databases or XML documents. In this case, the base relations created based on these sources will typically use a unique search method with a 5-tuple for each relation attribute. Each 5-tuple will indicate that the attribute may appear or not in the queries ('\( \text{OPT} \)' value in \( \text{obligatoriness} \)) and with any multiplicity ('\( \text{ANY} \)' value in \( \text{multiplicity} \)).

The situation may be different in other sources. Observe the following example.

**Example:** Consider the example of a virtual bookshop on the Internet the search form of which is like that shown in Figure 1.

![Figure 1 Search form of a virtual bookshop on the Internet](image-url)
The form obliges the user to specify a value for the TITLE attribute and gives the option to set a value for the AUTHOR attribute and for the FORMAT attribute (restricted to a group of values). The searches by title and author are searches by keyword (operator like). A search by exact phrase (operator ‘=’) is indicated by selecting the box next to the field search box. For each attribute a simultaneous search is allowed using one value only. In addition to the fields TITLE, AUTHOR and FORMAT, we can expect that the shop outputs a PRICE attribute, which cannot be queried directly using the form.

Let us model this source as a relation \( R=\{\text{TITLE, AUTHOR, FORMAT, PRICE}\} \) with a search method containing the 5-uples shown in Figure 2.

\[
\begin{align*}
(TITLE, \{\text{like, =}\}, \text{OBL, 1, Any}) \\
(AUTHOR, \{\text{like, =}\}, \text{OPT, 1, Any}) \\
(FORMAT, \{=\}, \text{OPT, 1, \{'All formats’,'Hardcover’, 'eBooks’, 'Paperback’\}}) \\
(PRICE, {}, \text{NOS, 0, \{\}})
\end{align*}
\]

Figure 2  Search method for a bookshop

Note that the first 5-tuple has the value \{like, =\} in the OPERATORS field and OBL in the OBLIGATORINESS field; this does not mean that it is obligatory to query the TITLE attribute with both operators, but that it is obligatory to query it at least with one of them. In order to have the TITLE attribute appear obligatorily in the query with both operators (this is not possible in the form in the example), this should be done with two different 5-uples for the TITLE attribute, one for each operator:

\[
\{(\text{TITLE}, \{\text{like}\}, \text{OBL, 1, ANY}) \ (\text{TITLE}, \{\text{=\}}, \text{OBL, 1, ANY})\}\)
\]

Thus, as can be seen, when you want to differentiate the treatment of a specific attribute according to the operator with which it is used, more than one 5-tuple can exist for each attribute.

2.1.2 Defining Data Sources and Wrappers

A wrapper must be created and assigned to a base relation in order to be able to obtain data from a specific source. Each wrapper should provide access to the data forming a base relation in a source, so that they are structured in a manner similar to a Relational Database table with regard to the DataPort server. More specifically, each wrapper should provide an overall view of the source according to the base relation model provided in the previous section.

The process of generating wrappers for source types such as JDBC/ODBC Databases, SOAP and REST Web Services, LDAP servers, XML documents, delimited text files or unstructured information indices may be undertaken quickly and simply with the help of the Denodo Virtual DataPort administration tool (see [ADMIN_GUIDE]). It is normally necessary to previously create a data source for the source that will encapsulate data to access it that is reusable by the different wrappers acting on it. Wrappers can be created for semi-structured Web sources (WWW) in a fully graphic manner with Denodo ITPIlot (see [ITPILOT]). It is also possible to create wrappers for specific applications using the CUSTOM wrapper type.

Should you wish to create data sources and wrappers directly using VQL, without the help of the administration tool, section 18 of this document describes how to do so. In general, use of the administration tool is strongly recommended for these tasks, as the process is much simpler and automatic.

Once the wrapper has been created for a source, all that needs to be done is link it to the adequate search method or methods from the base relation that represents said data in the system (see section 4). You can now make queries on the base relation.
### 2.1.3 Defining the Views of the Global Schema

Once the base relations have been defined and their corresponding wrappers constructed, each relation of the global schema is defined through a query on the base relations in a manner similar to that used to define views on a conventional database.

It is important to highlight that when defining a view, in addition to the base relations, previously defined views can also be used.

**Example:** Take three base relations

A = \{TITLE, AUTHOR, FORMAT, PRICE\},
B = \{TITLE, AUTHOR, FORMAT, PRICE\} and
C = \{TITLE, AUTHOR, AVERAGE_RELEVANCE\}.

A and B represent two electronic bookshops on the Internet. C represents a source in which users review books and the system allows the average review for a specific book to be searched for. Imagine that we want to obtain a relation of the global schema

\[ R = \{\text{TITLE, AUTHOR, PRICE, AVERAGE RELEVANCE}\} \]

that contains all the books of A and B, together with the average review according to C and the minimum value found for the \text{PRICE} attribute, amongst the occurrences of the book found in both sources. \( R \) can be defined in the two following steps:

1. Creating the view \text{bookview} like the union of A and B.

   ```sql
   CREATE VIEW BOOKVIEW AS
   SELECT * FROM @A
   UNION
   SELECT * FROM @B;
   ```

2. Creating the view \( R \) as the join of \text{bookview} and C by applying an operation \text{groupby} to select the minimum price for each book.

   ```sql
   CREATE VIEW R AS
   SELECT TITLE, AUTHOR, AVERAGE_RELEVANCE, MIN(PRICE) AS MINIMUM
   FROM bookview JOIN C ON bookview.TITLE = C.TITLE
   AND bookview.AUTHOR = C.AUTHOR
   GROUP BY TITLE, AUTHOR, AVERAGE_RELEVANCE;
   ```

As mentioned above, the base relations can present limitations in their query capabilities, which are expressed through search methods. When creating views Virtual DataPort can automatically calculate its search methods from those of the base relations and from the statement used to define the view. This allows the system to know \textit{a priori} if a specific query can be answered.

#### 2.1.3.1 Post-processing

When considering the query capabilities of a source, it is also important to bear in mind that the DataPort server can carry out post-processing operations on the results obtained from said source. From the query constraints of a source it is possible to obtain its list of capabilities as a superset of same by applying post-processing. This task is carried out automatically by the server.
2.2 EXECUTING STATEMENTS

Once the creation stage has been completed, DataPort is ready to accept queries and/or updates (updates can only be executed on "updatable views" as defined by the SQL 92 standard). Details on how external applications can send and execute statements on the Virtual Database are provided in the Developer Guide [DEVELOPER_GUIDE]. This section provides only a general overview of the statement execution process from an internal point of view.

When a VQL query is received, the Virtual DataPort query interpreter starts by checking if the query capabilities of the views involved allow obtaining an answer to the query. If the query cannot be answered to, the user is informed of this. If it can, the process continues.

Subsequently, the Plan Generator creates the possible execution plans for the query. The plans normally differ in aspects such as the algorithm used to execute the joins or the specific search methods selected on the sources.

The optimizer module is responsible for obtaining the cost of each of the plans, according to different parameters, so that the best can be selected. This process, among other tasks, is responsible for optimally distributing processing between the DataPort server and the sources, delegating operations such as groupby, selection conditions, joins or unions, where possible. Hence, data transfer on the network can be minimized. This stage is also responsible for tasks such as choosing the most suitable method for implementing join operators, for establishing the swapping to disk strategy for very large results or for managing use of the cache module. See section 19.2 for more details.

Once the optimum plan has been selected, the Execution Engine puts it into practice. Execution of a plan assumes execution of a series of sub-queries expressed in terms of the base relations only. These sub-queries will be executed by the wrapper of the corresponding source. It is remarkable how Virtual DataPort is capable of making maximum use of parallelism, whereby sub-queries are normally executed in parallel.

Finally, the execution engine combines the results returned by the sources in accordance with that specified for each plan, thus obtaining the final response to the query.

It is important to highlight that the system operates in an asynchronous manner. This means that as the results of the sources become available, the system begins to process them even if the sources have not yet issued a complete response. This considerably speeds up the times for obtaining the first tuples of the final result. Another important aspect is that the system is capable of processing partial results, i.e. it can process the query even if some of the sources are temporarily inaccessible, providing the results that can be obtained with the remaining sources while still informing the client application of the errors.

Another fundamental aspect is that, optionally, as mentioned previously, all or part of the source data can be pre-loaded in the server cache. In this case, the system will check if the sub-query received can be resolved with the data contained in the cache. If this is the case, the response is obtained directly from same instead of querying the source.

Virtual DataPort also supports the executing of updating statements (INSERT / UPDATE / DELETE) on views, provided these can be updated according to the standard definition in SQL-92. See section 7 for further details.
3 LANGUAGE FOR DEFINING AND PROCESSING DATA: VQL

The SQL Structured Query Language is a standardized database language supported by most relational database managers available on the market. Virtual DataPort provides a language called Denodo VQL (Denodo Virtual Query Language) which extends SQL with the required capabilities in a distributed information integration environment.

VQL, like SQL, is composed of commands, clauses, operators and aggregation functions. These elements are combined in the instructions to create, update and manipulate databases. This section describes the commands, clauses, operators and syntax of VQL.

3.1 DATA TYPES

The Virtual DataPort catalog includes a group of predefined data types. These types can be divided into two groups: basic types and compound types.

The basic data types supported are:

- `int`. Represents an integer number in the range -2147483648 to 2147483647.
- `long`. Represents an integer number in the range -9223372036854775808 to 9223372036854775807.
- `float`. Represents a real number in the range 1.4E-45 to 3.4028235E38.
- `double`. Represents a real number in the range 4.9E-324 to 1.7976931348623157E308.
- `boolean`. Represents a logical value, true or false.
- `text`. Represents a character string.
- `date`. Encapsulates a date.
- `money`. Represents a currency value.
- `blob`. Represents a binary data element. Blob data type values cannot take part in query conditions.
- `xml`. Represents an XML document (or a fragment of an XML document).

Compound data types are as follows:

- `enumerated`. Attributes of this data type can take as value one of a set of character strings defined by the data type.
- **register**: Data type that serves to represent data with an internal and heterogeneous structure, i.e. the fields into which the data are subdivided are not all the same type.

- **array**: Represents a list of elements of the same register type – therefore, the order of the elements matters –.

As can be seen below (see sections 10.1 and 19.1), Virtual DataPort allows defining specific compound data types. In this way, hierarchical data elements, such as those normally used in data sources such as Web Services or XML documents, can be naturally modeled.

### 3.1.1 Internationalization

Virtual DataPort incorporates support for the integration of data sources from different countries or geographic areas, also expressing the output data in the formats expected by the country in question.

For example, Virtual DataPort incorporates support to compare monetary amounts expressed in different currencies through automatic conversions. In a similar manner, DataPort offers support to display query results in a specific time zone independent of the zone used by the data sources (e.g. in Spain, although data are extracted from US sources, the results can display the currencies, times and dates corresponding to Spain).

For this, it is needed to have an internationalization configuration for each of the countries/locations from which data handled by the DataPort server can come, represented by a map of the type i18n (see map construction in Section 10.2). Virtual DataPort includes maps already created for the most common configurations of many countries. The name of those configurations uses the standard prefix defined in the standard ISO-3166 [COUNTRY_ISO] (e.g. Spain (es_euro), Great Britain (gb), France (fr), United States (us), etc.).

New internationalization configurations can also be added very easily. See section 19.3 for more information.

Lastly, it is important to bear in mind that the default format to be used to write date, money and double constants in the queries on a view is established by the internationalization configuration being used. See section 19.3 for more information about the parameters of an internationalization configuration and section 12 to know how to obtain the parameters assigned to a certain configuration. Section 3.7.3 describes different functions to process date values that may be useful to express them in the required format.

### 3.2 STATEMENTS

Two types of statements exist in VQL:

- **DDL (Data Definition Language)** statements that allow new relations, wrappers, etc. to be created and defined. The DDL commands are:
  
  - **CREATE**: Creates or replaces new tables (base relations), views, stored procedures, wrappers, data sources, published web services, maps, types, databases and users.
  
  - **DROP**: Eliminates elements such as tables (base relations), views, stored procedures, wrappers, data sources, published web services, maps, types, databases and users.
  
  - **ALTER**: Modifies specific properties of a table (or base relation) such as its internationalization configuration, cache, swapping configuration, and so on. It also allows modifying database and user permissions.
• DESC: Shows the description of data types, views, stored procedures, adapters, maps, operators, wrappers, data sources, published web services, databases and users defined in the server. It also allows to obtain a hierarchical description of how a specific view is constructed (views which define it along with the relational operators involved), and the VQL sentences required to rebuild a catalog element.

• LIST: Enumerates the different elements of the catalog (data types, views, etc.)

• GRANT and REVOKE: Allow to establish or revoke user permissions over databases, stored procedures and/or views.

• DML (Data Manipulation Language) statements, which enable to query and update data. Virtual DataPort provides the following DML statements:
  
  • SELECT, used to execute queries to the server.
  
  • INSERT, UPDATE and DELETE for inserting, updating and deleting, respectively.
  
  • BEGIN, COMMIT, ROLLBACK for beginning, committing and rolling back a transaction, respectively.
  
  • CALL, to call up stored procedures.

3.3 SELECT STATEMENT: CLAUSES

The SELECT statement is used to execute queries and to define new views. It is comprised of a series of clauses. The clauses supported by the language are:

• FROM: Specifies the relation or relations from which the data is selected. It is possible to specify subqueries. It is also possible to specify the invocation of a stored procedure.

• WHERE: Specifies the conditions that the data to be selected should fulfill.

• UNION: Performs the union of two SELECT statements.

• GROUP BY: Used to group the results obtained as the response to a query according to the specified aggregation fields.

• HAVING: The HAVING clause is used to filter the registers returned by a query using the GROUP BY clause.

• ORDER BY: Used to order the selected data according to the indicated attributes.

• CONTEXT: Used to modify certain configuration options to execute a specific query.
• **TRACE**: Provides the execution plan of a statement.

### 3.4 **INSERT / UPDATE / DELETE CLAUSES**

**INSERT / UPDATE / DELETE** statements allow for the tuples of a view to be inserted, updated and deleted, respectively, directly updating the data source. These statements can only be executed on views created using databases or CUSTOM-type sources. Furthermore, it must be possible to update views according to the definition of standard SQL-92 (see section 7).

The **INSERT** statement allows for a new tuple of data to be inserted in a view, updating the data source directly. It supports the following clauses:

- **INTO**: This indicates the view on which the data is to be inserted and its attributes.
- **VALUES**: This indicates the value for each attribute of the view of the new tuple inserted.
- **SET**: Alternative syntax to the use of the **VALUES** clause to specify the value of each attribute of the new tuple.

The **UPDATE** statement allows for one or several tuples of data of a view to be altered, updating the data source directly. It supports the following clauses:

- **UPDATE**: Indicates the view where tuples will be updated.
- **SET**: Indicates the attributes of the view that will be modified by the operation as well as the new values to be taken by each one.
- **WHERE**: Specifies the condition to be met by the tuples to be updated.

The **DELETE** statement allows for one or several tuples of a view to be deleted, updating the data source directly. It supports the following clauses:

- **FROM**: Indicates the view where tuples will be updated.
- **WHERE**: Specifies the condition to be met by the tuples to be deleted.

All statements mentioned also support the following clauses:

- **CONTEXT**: Used to modify certain configuration options to run a statement.
- **TRACE**: Provides the execution plan of a statement.

### 3.5 **LOGICAL OPERATORS**

Logical operators are used to create Boolean expressions (which are evaluated as *true* or *false*) typically used in a **WHERE** clause. The logical operators supported are:
• **AND**: Is the logical "and". Evaluates two conditions and returns a true value only if both are correct.

• **OR**: Is the logical "or". Evaluates two conditions and returns a true value, if one of the two is correct.

• **NOT**: Is the logical negation. It is applied to a condition and negates its value.

### 3.6 COMPARISON OPERATORS

An operator of this type returns the logical value *true* or *false* according to the evaluation result of two or more operands. Depending on the nature of the operator the operands should be of a specific data type. When the right operand of an operator can accept more than one value, these must be introduced separated by commas (see section 3.8).

The operators supported are:

- **’<’**: Receives two operands that can be of the types: int, long, float, double, date, money. Evaluated as *true* if the first operand is less than the second.

- **’<=’**: Applied to two operands of the same type as in the operator ’<’ and is evaluated as true if the first operand is less than or equal to the second.

- **’>’**: Receives two operands that can be of the types: int, long, float, double, date, money. Checks if the first operand is greater than the second.

- **’>=’**: Applied to two operands of the same types as the operator ’>’ and is evaluated as true if the first operand is greater than or equal to the second.

- **’=’**: Receives two operands that can be of the types: int, long, float, double, boolean, text, enumerated, date and money. Evaluates the equality of the two operands.

- **’<>’**: Applied to two operands of the same types as the operator ’=’ and is evaluated as true if the first operand is not equal to the second.

- **‘like’**: Accepts one text-type element and one or more SQL LIKE expressions as operands. It checks if the character string matches all the expressions received. Each expression must follow standard SQL format for the expressions used with the SQL like operator:
  
  - The character ‘%’ represents a segment of any length within a character string.
  - The character ‘_’ represents a segment of length 1.

For example, the expression ‘%commerce_’ matches any string ending with the substring ‘commerce’ followed by a character. If the characters ‘%’ or ‘_’ are included as part of a constant substring, they must be escaped by prefixing them with the character ‘$’.

If the escape character is included, it must be escaped as well (e.g. ‘$ $’).
**Examples**: The first query returns tuples from the view `internet_inc` with a summary attribute containing the text 'adsl'. The second query requires that they also contain the text 'error'.

```sql
SELECT * FROM internet_inc WHERE summary like '%adsl%'
SELECT * FROM internet_inc WHERE summary like '%adsl%','%error%'
```

- `'regexp_like'`: It has two parameters: a text-type parameter and a regular expression [REGEXP]. It checks if the text-type parameter matches the regular expression.

**Examples**: Consider the following view `PRODUCTS`:

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJ00</td>
<td>Product A</td>
</tr>
<tr>
<td>AJ17</td>
<td>Product B</td>
</tr>
<tr>
<td>AJ1A8</td>
<td>Product C</td>
</tr>
<tr>
<td>PQ983</td>
<td>Product D</td>
</tr>
<tr>
<td>PQ900</td>
<td>Product E</td>
</tr>
</tbody>
</table>

The query

```sql
SELECT * FROM products WHERE identifier regexp_like 'AJ\d+'
```

returns the rows:

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJ00</td>
<td>Product A</td>
</tr>
<tr>
<td>AJ17</td>
<td>Product B</td>
</tr>
</tbody>
</table>

- `'contains'`: Accepts two text-type elements as operands. The first operand will be a text-type attribute in a view created from a searchable external index on non-structured data (indexes are typically imported using Aracne and/or Google Mini data sources. See sections 18.3.8 and 18.3.9). The second operand will be a Boolean search expression written in the search language on non-structured data supported by DataPort (see section 20.2).

The syntax of the search language on non-structured data is described in section 20.2. However, bear in mind that the search options available depend on the capacities natively provided by the data source. For example, Google Mini does not support different characteristics of the search language such as proximity searches. Therefore, when the contains operator is used with attributes from Google Mini sources, these capacities will not be available. Section 20.3 provides exact details as to the search capacities supported for Google Mini sources and Aracne sources. The Custom-type wrappers allowing access to other data sources can specify the search language capacities for contains that are supported through Configuration Properties (see section 18.3.13.1).

In the case of derived views, the search capacities supported for an attribute are calculated by DataPort depending on the capacities of their base view attributes. It is possible to view the capacities of each
attribute by using the `DESC VIEW` statement to query the value of its Configuration Properties (see sections 18.3.8 and 18.3.13.1).

**Examples**: The following query returns the tuples from the `aracnaview` view, where the `searchablecontent` attribute contains the words ‘acme’ and ‘incorporated’:

```
SELECT * FROM aracnaview WHERE searchablecontent contains 'acme AND incorporated'
```

The following query returns the tuples from the `aracnaview` view where the `searchablecontent` attribute contains the exact words ‘acme incorporated’ and some other word starting with ‘product’:

```
SELECT * from aracnaview WHERE searchablecontent contains "acme incorporated "AND product*"
```

- ‘containsor’: Accepts 2 or more text-type elements as operands. It checks if the first string contains at least one of the other strings received.

- ‘isContained’: Accepts 2 or more text-type elements as operands. It checks whether the first string is contained in all the other strings received.

- ‘is not NULL’: Applied to one operand, which can belong to the following data types: `int`, `long`, `float`, `double`, `boolean`, `text`, `enumerated`, `date`, `money` and `link`. Checks if the value is not null, i.e. if it has any value.

- ‘is NULL’: Receives an operand that can belong to one of the following data types: `int`, `long`, `float`, `double`, `boolean`, `text`, `enumerated`, `date`, `money` and `link`. Evaluates if the value is null, i.e. if it does not have any value.

- ‘is TRUE’: Applied to one operand of the type `boolean`. It returns the logical value of the operand (i.e. true if - and only if - its value is true; otherwise false).

- ‘is FALSE’: Receives an operand of the type `boolean`. It returns the negation of the logical value of the operand (i.e. true if the operand is evaluated as false; otherwise false).

- ‘in’: Receives a list of operands that can belong to one of the following data types: `int`, `long`, `float`, `double`, `text`, `enumerated`, `date` and `money`. Returns true if the operand on the left side is included in the list of operands on the right side. The list of operands may or may not be between brackets.

**Example**: The following two statements produce the same result: They select tuples from the view `internet_inc` for which their value for the `taxid` attribute is the same as the value ‘B78596011’ or ‘B78596012’:
SELECT * FROM internet_inc WHERE taxid in ('B78596011','B78596012')

SELECT * FROM internet_inc WHERE taxid in 'B78596011','B78596012'

• ‘between’: Applied to three operands that can belong to one of the following data types: int, long, float, double, date and money. Returns true if the operand on the left side is found in the range specified by the other two operands, including the limit values. As an alternative syntax, the operands limiting the range may be separated by the word AND.

Example: The following two statements produce the same result: They select tuples from the view internet_inc for which their value for the iinc_id attribute is within the range of 2 and 4 (inclusive):

SELECT * FROM internet_inc WHERE iinc_id between 2 AND 4

SELECT * FROM internet_inc WHERE iinc_id between 2,4

• ‘~’. The evaluation of this operator returns a value between 0 and 1 that estimates the similarity between the two text-type operands using a variety of similarity algorithms. In addition to the operands to compare, the similarity operator receives the similarity algorithm to use and a minimum similarity threshold as parameters. Where the similarity between character strings reaches or exceeds the threshold, the condition is assessed as true. Where this is not the case, it is assessed as false. The left-hand (text-type) operand is one of the character strings to compare. The second specifies the minimum similarity threshold (a value of between 0 and 1) and the third (optional) specifies the similarity algorithm to be used. The algorithms available are the same as for the similarity function (see section 3.7.2).

Example: The following query returns tuples for which their customername field has a similarity of over 0.7 with the ‘General Motors Inc’ string, using the Jaro Winkler editing distance algorithm between strings:

SELECT * FROM internet_inc_cname WHERE customer_name ~ 'General Motors Inc','0.7','JaroWinkler'

3.7 FUNCTIONS FOR CONDITIONS AND DERIVED ATTRIBUTES

Derived attribute functions are used to generate new attributes in the schema of a view, applying an expression in function of the other attributes of the view, constants and/or the result of evaluating other functions. Expressions using these functions can also be used as operands in the conditions.

A function is defined as an identifier and a list of arguments that can in turn be constants, fields or other functions.

Virtual DataPort provides a series of predefined functions that can be grouped into different types based on the data type to which they can be applied:
- Arithmetic functions
- Functions for text processing
- Functions for date processing
- Type conversion functions.
- Functions for processing XML-type elements.
- Other functions.

The functions supported by the system are described in the following paragraphs. See section 20.1 for detailed examples of the use of each function.

Additional functions can be added to the server by uploading Jars containing custom functions (see section 19.3.1) to the server.

**NOTE:** Functions are generally represented in prefix notation, i.e. an identifier is indicated followed by a list of parameters in brackets and separated by commas. For some functions there is also an infix notation (for some arithmetic functions, for example).

### 3.7.1 Arithmetic Functions

Arithmetic functions are applied to attributes and constants of the following types: int, long, float, double and money. MIN and MAX functions can be also applied to date elements.

In general, if a function accepts numeric arguments of different types, it will return a result of the most generic type. For instance, the addition of an int-type value and a double-type value will return a double-type result. Regarding the money type, if a function receives a money-type argument and other numeric-type one, it will return a money-type element.

Appendix 20.1.1 contains examples of how to use these functions:

- **SUM**: This function receives a variable number of arguments (two or more) and returns the sum of these. The infix version of this function has two arguments and is represented by the symbol `+`.

- **SUBTRACT**: This function receives two arguments and returns the result of subtracting the value of the second argument from the first. The infix version of this function receives two arguments and is represented by the symbol `-`.

- **MULT**: This function receives a variable number of arguments (two or more) and returns the result of multiplying them. The infix version of this function receives two arguments and is represented by the symbol `*`.

- **DIV**: This function receives two numeric-type arguments and returns the result of dividing the first argument by the second. If both arguments are integers, the result of the division will also be an integer. The infix version of this function receives two arguments and is represented by the symbol `/`.

- **MIN**: This function receives a variable number of arguments (two or more) and returns the smallest argument of the list. This function also accepts date arguments.

- **MAX**: This function receives a variable number of arguments (two or more) and returns the greatest argument of the list. This function also accepts date arguments.

- **ABS**: This function receives one numeric-type argument and returns its absolute value.
- **MOD**: This function receives two arguments and returns the result of the module operation between the first argument and the second (the remainder of the full division of the first and second arguments). The infix version of this function receives two arguments and is represented by the symbol '%'.

- **CEIL**: This function receives a numeric argument and returns the smallest integer, greater than or equal to the argument, closest to the argument. If the argument has `int` type, it returns a value of `int` type. If the argument has type `long`, `float` or `double`, the returned value is of type `long`. If the argument has type `money`, the returned value has the same type.

- **FLOOR**: This function receives a numeric argument and returns the biggest integer, less than or equal to the argument, closest to the argument. If the argument has `int` type, it returns a value of `int` type. If the argument has type `long`, `float` or `double`, the returned value is of type `long`. If the argument has type `money`, the returned value has the same type.

- **ROUND**: This function receives a numeric argument and returns as a result the integer number closest to the argument. If the argument has `int` type, it returns a value of `int` type. If the argument has type `long`, `float` or `double`, the returned value is of type `long`. If the argument has type `money`, the returned value has the same type.

- **POWER**: This function is given two numeric arguments, the second of which must be an integer. It returns a `double`-type value result obtained through the exponentiation of the first argument with the second as the exponent.

- **SQRT**: This function is given a numeric argument and returns a `double`-type value with the result of the square root of the argument.

- **LOG**: This function is given a numeric argument and returns a `double`-type value with the result of the base 10 logarithm of the argument.

- **RAND**: This function does not receive any arguments and returns a random `double`-type value between zero and one.

### 3.7.2 Text Processing Functions

Text processing functions have the objective of executing a transformation or calculation on a text-type attribute or literal.

These functions can also be used to generate transformations of values belonging to other data types, but considering them as `text`. In general, if a function takes an argument of some other type when the expected one is `text`, the parameter will be converted to `text` before calling the function.

Appendix 20.1.2 contains examples of how to use these functions:

- **CONCAT**: receives one or more arguments and returns a text-type element containing the result of concatenating its parameters. The infix version of this function receives 2 arguments and is represented by the symbol `||`. 

Language for Defining and Processing Data: VQL 17
• **INSTR**: returns the index of a string within another string.

• **LEN**: receives a text-type argument and returns the number of characters that form it.

• **LOWER**: receives a text-type argument and returns the same string in lower case.

• **LTRIM**: returns a copy of the string without its leading whitespaces.

• **REGEXP**: This function allows executing transformations on character strings based on regular expressions. It receives three arguments: one text-type element, one input regular expression and one output regular expression. The regular expressions must be expressed using the regular expression syntax in JAVA language [REGEXP]. The function behaves in the following manner: The input regular expression is evaluated against the text from the first argument and the output regular expression may include the "groups" defined in the input regular expression. The portions of text matching them will be replaced in the output expression. For example, the result of evaluating:

  \[
  \text{REGEXP('Shakespeare,William', '(\w+),(\w+)', '$2 $1')}
  \]

  is 'William Shakespeare'.

• **REMOVEACCENTS**: receives a text-type argument and returns that same argument value but with no accents. For instance, the string 'Aébá' would be transformed into "Aeba".

• **REPLACE**: receives three text-type arguments and returns the result of replacing the occurrences of the second argument in the first one by those of the third.

• **REPLACEMAP**: receives a text and a map of transformations as inputs, specifying a series of texts (known as keys) that must be replaced by others (known as replacement values) in the original text. This includes two possible signatures:

  o **REPLACEMAP** (originalText: text, mapName: text). The keys and the replacement values are specified by a key/value map defined by the administrator (see section 10.2 to learn how to create maps). The function is given two arguments: The first indicates the text on which to make the transformations and the second the name of the map.

  o **REPLACEMAP** (key: text, viewName: text, keyField: text, valueField: text). The keys and replacement values are specified through a DataPort view. It is given four parameters: the text on which to make the transformations, the name of the view containing the transformation map, the name of the view attribute containing the keys and the name of the view attribute containing the replacement values.

  Both signatures return a text-type element containing the original text, once all the specified transformations have been made (where the key does not exist, it is returned as null). The key is upper/lower case-insensitive.

**Example**: Suppose that the test map contains the following correspondences:
ADSL -> DSL
Error -> Warning

The following query returns tuples with an attribute known as `new_summary`, the values of which are obtained by taking the value of the `summary` attribute from the `internet_inc` view and replacing the occurrences of the word "ADSL" with "DSL" and "Error" with "Warning".

```
SELECT REPLACEMAP (summary,'test') AS new_summary FROM internet_inc
```

Appendix 20.1.2.6 contains more examples of REPLACEMAP.

- **RTRIM**: returns a copy of the string without its trailing whitespaces.

- **SIMILARITY(value1: text, value2: text, algorithm: text)**: receives two strings and returns a number between 0 and 1, which is an estimated measurement of similarity between the strings. The third parameter (optional) specifies the algorithm to use to calculate the similarity measurement. DataPort includes the following algorithms (if no algorithm is specified, DataPort chooses the one to apply):
  
  o Based on the editing distance between the text strings: `ScaledLevenshtein`, `JaroWinkler`, `Jaro`, `Level2Jaro`, `MongeElkan`, `Level2MongeElkan`.

  o Based on the appearance of common terms in the texts: `TFIDF`, `Jaccard`, `UnsmoothedJS`.

  Combinations of both: `JaroWinklerTFIDF`.

**Example**: The following query returns tuples for which their `customername` field has a similarity of over 0.7 with the 'General Motors Inc' string, using the Jaro Winkler editing distance algorithm between strings:

```
SELECT * FROM internet_inc_cname WHERE similarity(customer_name,'General Motors Inc','JaroWinkler') > 0.7
```

- **SPLIT**: splits strings around matches of a given regular expression and returns the array of those substrings.

- **SUBSTRING**: has three parameters: a text-type argument and two integers. It returns a substring of the first argument that corresponds to the positions indicated by the second (beginning) and third (end) arguments.

- **TEXTCONSTANT**: creates a text-type element from a literal passed as a parameter. It is only needed in the `SELECT` clause to specify constant string as value for a new field.
• TRIM: receives a text-type argument and returns the same argument, but removing all the spaces and carriage returns that are either at the beginning or the end of the argument.

• UPPER: This function receives a text-type argument and returns the same string in upper case.

3.7.3 Date Processing Functions

Date functions to manipulate date values.

Appendix 20.1.3 contains examples of how to use these functions:

• ADDHOUR: Receives two arguments: a date and an integer. It returns this date with its field hour rolled up (or down, if the integer is negative) by the amount specified.

• ADDMINUTE: Receives two arguments: a date and an integer. It returns this date with its field minute rolled up (or down, if the integer is negative) by the amount specified.

• ADDSECOND: Receives two arguments: a date and an integer. It returns this date, with its field second rolled up (or down, if the integer is negative) by the amount specified.

• ADDDAY: Receives two arguments: a date and an integer. It returns this date with its field day rolled up (or down, if the integer is negative) by the amount specified.

• ADDWEEK: Receives two arguments: a date and an integer. It returns this date with its field week rolled up (or down, if the integer is negative) by the amount specified. That is, rolled up or down by multiples of 7 days.

• ADDMONTH: Receives two arguments: a date and an integer. It returns this date with its field month rolled up (or down, if the integer is negative) by the amount specified.

• ADDYEAR: Receives two arguments: a date and an integer. It returns this date with its field year rolled up (or down, if the integer is negative) by the amount specified.

• FIRSTDAYOFMONTH and FIRSTDAYOFWEEK: Receive a date-type argument and return that date with the field day rolled down to the first day of the month or week, respectively.

• FORMATDATE: Receives a date-type argument and a date pattern (following the JAVA syntax [JAVADATEFORMAT]) and returns a formatted date string. There is an optional third parameter, localeName, which indicates the locale (see section 3.1.1) of the output string.

• GETDAY: Receives a date-type argument and returns a long-type object that represents the day of the date received.
- **GETHOUR**: Receives a `date`-type argument and returns a `long`-type object that represents the time of the date received.

- **GETMINUTE**: Receives a `date`-type argument and returns a `long`-type object that represents the minutes of the date received.

- **GETSECOND**: Receives a `date`-type argument and returns a `long`-type object that represents the seconds of the date received.

- **GETTIMEINMILLIS**: This receives a `date`-type argument and returns a `long`-type object representing the number of milliseconds since 1 January 1970 at 00:00:00 GMT until the date received as parameter, the second of the date received.

- **GETMONTH**: Receives a `date`-type argument and returns a `long`-type object that represents the month of the date received.

- **GETYEAR**: Receives a `date`-type argument and returns a `long`-type object that represents the year of the date received.

- **LASTDAYOFMONTH** and **LASTDAYOFWEEK**: Receive a `date`-type argument and return this `date` with the field day rolled up to the last day of the month or week, respectively.

- **MAX, MIN**: `MIN` and `MAX` return the higher or lower date of the list of `date`-type values passed as parameter.

- **NEXTWEEKDAY**: Receives two arguments: a `date` and an `integer`. Returns this `date` with its field day rolled up to the day of the next week specified by the second parameter. The numbers of each day of the week are: Sunday = 0, Monday = 1, Tuesday = 2 …

- **NOW**: This function creates a new data value containing the current date.

- **PREVIOUSWEEKDAY**: Receives two arguments: a `date` and an `integer`. Returns this `date` with its field day rolled down to the day of the last week specified by the second parameter. The numbers of each day of the week are: Sunday = 0, Monday = 1, Tuesday = 2 …

- **TO_DATE**: Converts a string containing a date to a `date`-type element. It has three text-type arguments. The first represents a pattern to express dates (following the standard syntax in JAVA language specified in [JAVADATEFORMAT]). The second is a date expressed according to that pattern. The third one is a text-type parameter which indicates the internationalization configuration that represents the “locale” of the date to process. As a result, a `date`-type element equivalent to the specified date is returned.

- **TRUNC**: Receives a `date`-type argument and can receive a `text`-type argument with a date pattern. It returns this `date` truncated to a specific unit of measure indicated by the pattern. This function has the same syntax as the `TRUNC(date)` function of the Oracle database.
3.7.4 Type Conversion Functions

These functions allow for different transformations among different types of data.

Appendix 20.1.5 contains examples of how to use these functions:

- **ARRAY_TO_STRING**. Converts an array field to a string that contains the elements of the array separated by a character.

- **CAST**. This function has two arguments. The first specifies the name of a data type and the second specifies a value to which said data type is to be converted. The following table shows the possible type conversions:

<table>
<thead>
<tr>
<th>Output Type</th>
<th>Input Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>Blob</td>
<td>text, blob</td>
</tr>
<tr>
<td>boolean</td>
<td>text, int, long, float, double, boolean</td>
</tr>
<tr>
<td>Date</td>
<td>text, date, long</td>
</tr>
<tr>
<td>double</td>
<td>text, int, long, float, double, money</td>
</tr>
<tr>
<td>enumerated</td>
<td>text, enumerated</td>
</tr>
<tr>
<td>float</td>
<td>text, int, long, float, double, money</td>
</tr>
<tr>
<td>Int</td>
<td>text, int, long, float, double, money</td>
</tr>
<tr>
<td>Long</td>
<td>text, int, long, float, double, money</td>
</tr>
<tr>
<td>money</td>
<td>text, int, long, float, double, money, date</td>
</tr>
<tr>
<td>register</td>
<td>xml, register</td>
</tr>
<tr>
<td>Text</td>
<td>text, int, long, float, double, boolean, date, xml, money, link, blob, enumerated, register, array</td>
</tr>
<tr>
<td>Xml</td>
<td>text, blob, xml, register, array</td>
</tr>
</tbody>
</table>

Type conversions permitted with the CAST function

- **CREATETYPEFROMXML**. Creates a register compound type (see section 19.1) based on an XML-type element. It receives two arguments: the first belongs to the text type and must contain the name of the new type, whereas the second contains the XML element. The XML parameter provided as second argument can be of type xml or text. See section 3.7.5 for more details.

- **REGISTER**. Creates a register compound type (see section 19.1) with the values of the fields of a view.

- **TO_DATE**. Converts text strings representing dates into date-type elements. See section 3.7.3.

3.7.5 XML Processing Functions

These functions allow for XML-type elements to be created and processed.

Appendix 20.1.4 contains examples of how to use these functions:

- **CREATETYPEFROMXML**. Creates a register or array compound type (see appendix 19.1) based on an XML-type element. It receives two arguments: the name of the new type and a string containing an
example of the XML element (of text type). This function infers the structure of the new type by analyzing the XML. It returns the name of the new type created. See next sub-section for an example.

- **XMLQUERY**. Extracts information from an XML document using the XQuery language [XQUERY].

- **XPATH**. Applies an Xpath expression [XPATH] to an XML-type element. It receives two mandatory arguments: one XML-type element and one text containing the Xpath expression. It returns an XML element with the result of applying the expression. It can optionally receive a third Boolean-type parameter. When this third parameter takes the value true the XML header ('"<?xml version="1.0" encoding="UTF-8"?>') will be added to the result. Note that the result of applying an Xpath expression may be an individual value (integer, text, etc.). In this case, it is possible to use the CAST function to convert it into the corresponding Virtual DataPort type.

- **XSLT**. Returns the result of applying an XSLT stylesheet to a XML value. Both the XML and the XSLT stylesheet can be obtained from a DataPort view or from a local file.

### 3.7.5.1 Converting XML Data into Virtual DataPort Compound Types

By combining the CAST and CREATETYPEFROMXML functions we can create new register-type or array-type compound attributes in a view (see section 19.1) from XML data. For example: suppose we have a view V with an XML-type attribute called PERSONAL_DATA_XML. The data contained in this attribute has the following structure:

```
<person>
  <name> </name>
  <age> </age>
</person>
```

Now consider the following expression:

```
CREATE VIEW PERSON AS
SELECT CAST(
    CREATETYPEFROMXML(
        'personaldata_type',
        '<person><name> John Smith </name><age>25</age></person>'
    ), PERSONAL_DATA_XML) PERSONALDATA
FROM V
```

The type of the derived attribute PERSONALDATA of the new view PERSON is personaldata_type. This type is a register type made up of the fields name (text type) and age (long type).

The second parameter of the CREATETYPEFROMXML function must be an example of the values contained in the PERSONAL_DATA_XML field of the view V.

CREATETYPEFROMXML can also create array types. This will happen when the XML data passed to the first parameter and second parameters has repeated elements. E.g.:

```
<titles>
  <title lang="en"> </title>
  <title lang="en"> </title>
</titles>
```

In this case, the type created by CREATETYPEFROMXML is a register of arrays. Each component of the array is a register with two components: title and lang.
Converting XML-type data into DataPort compound-type data allows the data in XML code to be combined with data from other relations. For example, suppose you have a view RISK_LEVEL with two attributes called age (long type) and risk (double type), which includes some type of risk index calculated according to the age of an individual. It would be possible to run a join operation between the PERSON view and the RISK_LEVEL view using the age attribute of RISK_LEVEL and the age field of the PERSONALDATA attribute in the PERSON view.

### 3.7.6 Other Functions

This section describes miscellaneous functions.

Appendix 20.1.7 contains examples of how to use these functions:

- **COALESCE:** This function receives a variable number of arguments (two or more); all of the same data type, and returns the first non-null argument. COALESCE is equivalent to the expression:

  ```sql
  CASE WHEN arg1 IS NOT NULL THEN arg1
       WHEN arg2 IS NOT NULL THEN arg2
  ...
  END
  ```

- **CONTEXTUALSUMMARY:** This function obtains a contextual summary of a text based on a keyword search. A series of text fragments containing the word or sentence specified is obtained. This has the following signature:

  ```sql
  CONTEXTUALSUMMARY(content:text, keyword:text, 
                      [beginDelim:text, endDelim:text, fragmentSeparator:text, 
                      fragmentLength:int [,maxFragmentsNumber:int]]
  ```

  where:

  - **content:** text to analyze and the one from which the most relevant fragments are extracted (mandatory)
  - **keyword:** the keyword used to extract the text fragments (mandatory). The content of this argument can be a single word, or a sentence.
  - **beginDelim:** text to add as prefix of the keyword whenever it appears in the text (optional, default value is "").
  - **endDelim:** text to add as suffix of the keyword whenever it appears in the text (optional, default value is ").
  - **fragmentSeparator:** text to use as separator of the different text fragments obtained as a result (optional, default value is ".")
  - **fragmentLength:** approximate number of characters that will appear before and after the keyword occurrences inside of the text (optional, default value is 5).
  - **maxFragmentNumber:** maximum number of fragments to retrieve.
  - **analyzer:** analyzer to use when performing the keywords search. By default, the Standard Analyzer (std) is used: this analyzer does not consider lemmatization or stopwords. Analyzers for English (en) and Spanish (es) that include those features are also included.
• **GETSESSION**: Provides information about the session established with a Virtual DataPort server.

• **HASH**: This function receives a single text-type argument and returns an MD5 HASH of it.

• **IS_PROJECTED_FIELD**: returns true if a certain field is projected in the view.

• **MAP**: This function returns the value associated with a key. The pair key-value can be obtained from a view or from a Map (see section 10.2) When the key doesn’t exist, the function returns NULL. There are two possible signatures:

  MAP (key:text, view_name:text, key_field:text, value_field:text)
  
  It obtains the value associated with a key. MAP searches the value of a key in the columns of a view.
  
  o **key**: The value to search in the view.
  o **view_name**: The view where the key and the value are stored.
  o **key_field**: The column of the view that contains the keys.
  o **value_field**: The column of the view that contains the values.

  MAP (key:text, map_name:text[, i18n:text ])
  
  It obtains the value associated with a key from a Map.

  **Note**: key is case-insensitive parameter.

• **NULLIF**: This function compares two values or expressions and returns NULL if they are equal. Otherwise it returns the first value:

  NULLIF(<expression1>, <expression2>)

  This function is equivalent to the statement:

  ```
  CASE WHEN <expression1> = <expression2> THEN NULL ELSE <expression1> END
  ```

  **Note 1**: NULLIF removes the leading and trailing whitespaces of the parameters of type String before comparing them.

  **Note 2**: NULLIF performs implicit type conversion: if the two parameters have different type, it will try to cast one of them in order to make the comparison. I.e.: if the first parameter is ‘1’ (String) and the second is 1 (Integer), it will convert the String parameter to an Integer and they will be considered equal even if their type is different.
3.7.7 Aggregation Functions

Aggregation functions are used in SELECT statements to return one single value for every group of tuples obtained as result of a grouping operation.

The aggregation functions currently supported by Virtual Dataport receive as a parameter an expression indicating the name of the attribute to which it is applied. This parameter can optionally be preceded by one of two modifiers: ALL or DISTINCT. These modifiers affect the semantics of certain aggregation functions, applying them to all tuples in a group or only to those with a different value for the attribute in question.

Appendix 20.1.6 contains examples of how to use these functions:

- **AVG**: Calculates the average of the values of a specific attribute. Applicable to attributes of the type int, long, float, double and money. It always returns a double value.

- **COUNT**: Returns the number of tuples resulting from a selection operation (if the special wildcard '*' is specified as an attribute) or the number of tuples that have a non-null value for a specific attribute. Applicable to any type of attribute. This function can be used in queries not including a GROUP BY clause, but in that case it may only be used with the special attribute '*'.

- **FIRST**: Returns the value of an attribute in the first tuple of each group of values. Applicable to any type of attribute. This function ignores the ALL/DISTINCT modifier.

- **GROUP_CONCAT**: Concatenates the non-NULL values of each group into a single string. Applicable to any type of attribute.

- **LAST**: Returns the value of an attribute in the last tuple of each group of values. Applicable to any type of attribute. This function ignores the ALL/DISTINCT modifier.

- **LIST**: Returns a list with all the values of a specified attribute. Applicable to any type of attribute.

- **MAX**: Returns the highest value of a specified attribute. Applicable to attributes of the type int, long, float, double, date and money. This function ignores the ALL/DISTINCT modifier.

- **MIN**: Returns the lowest value of a specified attribute. Applicable to attributes of the type int, long, float, double, date and money. This function ignores the ALL/DISTINCT modifier.

- **NEST**: Returns an array with the values of the selected fields. Its result is inverse to the result of the FLATTEN views (see section 5.1.2 for more information about FLATTEN views)

- **SUM**: Returns the sum of all the non-null values of a specific attribute. Applicable to attributes of the type int, long, float, double and money.

Additional functions can be added to the server by uploading Jars containing custom functions (19.3.1) to the server.
3.8 **SYNTAX CONVENTIONS**

The following sections of this document describe the different operations that can be executed using VQL. The notation and syntax conventions used for this description are provided below.

- The language is not case-sensitive.

- The text in lower case and specified between the symbols ‘<’ and ‘>’ (e.g. `<name>`) indicates an element whose specific syntax will be specified later. If the separator ‘:’ appears (e.g. `<name:element-definition>`), this indicates a name of a representative element followed by the name of the element that defines it.

- The symbols ‘::=’ declare the definition of an element.

- The square brackets ([ ]) indicate optional elements. When they must appear in a statement, they are specified in inverted commas to explicitly indicate that they should appear and that they do not indicate optional elements.

- The asterisk (*) indicates that an element can be specified zero or more times. Example: `[[<search_method_clause>]]*` indicates that the element `[[<search_method_clause>]]` can be repeated as many times as necessary.

- The plus sign (+) indicates that an element can be specified one or more times. Example: `[[<field>]]+` indicates that the element `[[<field>]]` should appear at least once and can be repeated as many times as required.

- Elements separated by the character “|” and possibly grouped between braces ({})) indicate alternative elements. For example: `{element1 | element2}` indicates that `element1` or `element2` have to be written in this position.

- The commas (,) are used in syntax constructions to separate the elements of a list.

- The brackets (()) normally serve to group expressions and increase priority. In some cases they are required as part of the specific syntax of a statement.

- The full stop (.) is used in numeric constants and to separate names of tables, columns and fields.

- The blank space character can be a space, a tab, a carriage return or a line jump.

- Identifiers (<identifier>). Identifiers allow names to be linked to the different elements of the catalog and, in general, they are alphanumeric and may not commence with a number. A series of reserved words exists that cannot be used as identifiers (see Figure 3).

- Numbers (<number>). A number is a combination of digits that can be preceded by a ‘-’ symbol and can include a full stop as a decimal separator point and optionally an exponent (if they are real numbers).

- Logical values (<boolean>). Representation of the "true" and "false" logical values.
• Literals (<literal>). They represent any string that is not an identifier nor a number nor a logical value. This may be any string that is found in inverted commas (single or double commas). If a literal contains single or double comma characters (depending on the case), they should be escaped (' and " respectively).

• Operators (<operator>). Represent operators in the system.

```
<identifier> ::= [A-Za-z\200-\377_\[A-Za-z\200-\377_0-9$]*
<integer> ::= [-][0-9]+
<boolean> ::= true | false
<literal> ::= '\[^\]' | "[^\"]"
<operator> ::= <unary operator> | <binary operator>
<opsymbol> ::= [\~\!\@#$\%^&\|\`\?\<>\=]+
<unary operator> ::= is null | is not null | is true | is false
<binary operator> ::= = |
| <identifier>
| <opsymbol>
<reserved VQL word> ::= ADD, ALL, ALTER, AND, ANY, ARN, AS, ASC, BASE, CALL, CASE, CLEAR, CONNECT, CONTEXT, CREATE, CROSS, CUSTOM, DATABASE, DEFAULT, DESC, DF, DISTINCT, DROP, EXISTS, FALSE, FILTER, FLATTEN, FROM, FULL, GRANT, GROUP BY, GS, HASH, HAVING, HTML, IF, INNER, IS, JDBC, JOIN, LDAP, LEFT, MERGE, MY, NATURAL, NESTED, NOS, NOT, NULL, OBL, ODBC, OF, OFF, ON, ONE, OPT, OR, ORDER BY, ORDERED, PRIVILEGES, READ, REVERSEORDER, REVOKE, RIGHT, ROW, SELECT, SWAP, TABLE, TO, TRACE, TRUE, UNION, USER, USING, VIEW, WHEN, WHERE, WITH, WRITE, WS, ZERO
```

Figure 3  Basic primitives for specifying VQL statements

3.8.1 Syntax of Functions and Condition Values

As mentioned throughout this manual, different types of functions exist in Virtual DataPort: aggregation functions and functions used in conditions and to create derived attributes.

Virtual DataPort functions syntax is shown in Figure 4.

```
<field name> ::= <identifier> [. <identifier>] |
| <identifier> [. <identifier>] ! ['<integer>'] |
| [compound field name]* |
| <identifier> [. <identifier>] ! [compound field name]* |
| <compound field name> ::= . <identifier> | ['. <integer>']
<funcsymbol> ::= [\+\-\*\/*\%]+
<value> ::= NULL |
| <field name> |
| <number> |
| <boolean>
```
To define the syntax of a function we use the following elements:

- The element `<field name>` defines the syntax for specifying an attribute of a view or base view. Notice that attributes may be of compound types (see section 19.1 for a detailed description of support for compound types).

- The `<value>` element defines the syntax for any parameter of a function. They can be the name of an attribute, a numeric, Boolean or literal constant. It is also possible to create a compound value using the ROW constructor (see section 5.3.1). As can be observed, the parameter of a function can also be a new function. In addition, a `<value>` allows infix notations to be specified for a function (see the `<value> <funcsymbol> <value>` rule).

A function element is defined as an identifier followed by a list of parameters in brackets and separated by commas. The parameters of a function can be "*", single valued `{<value> elements}` or multivalued `{<value> elements in square brackets and separated by commas}`.

The syntax explained earlier is common for all types of functions existing in Virtual DataPort. However, some peculiarities may exist for a particular function type. These peculiarities, when they exist, are mentioned in the section of the manual corresponding to each function type.
Finally, it is important to remember that the format to be used to represent date-type constants and other fields whose data type shows internationalization characteristics when querying a view or base relation is set by the internationalization configuration being used for same. See section 19.2 for more information on the different internationalization configuration parameters and section 12 to find out how to consult the parameters assigned to a specific internationalization configuration.
4 CREATING A BASE RELATION (OR BASE VIEW)

The statement CREATE TABLE allows creating a base relation (also called base views) in Virtual DataPort. A base relation represents an external source (Web, relational, etc.) that supplies data for the mediator system.

**NOTE:** It is strongly recommended to graphically perform the base view creation process using the DataPort administration tool (see [ADMIN_GUIDE]) instead of manually writing VQL statements.

The syntax of the statement CREATE TABLE is shown in Figure 5. Each base relation or view is composed of a group of attributes. Each attribute of a relation belongs to a data type.

When creating the base relation its name, internationalization configuration and schema are specified.

```
CREATE [ OR REPLACE ] TABLE <name:identifier> I18N <name:identifier>
   ( <field> [, <field> ]* )

<field> ::=<name:identifier>:<type:identifier> [ ( <property list> ) ]

<property list> ::=<name:identifier> [= <value:identifier>]
   [, <name_i:identifier> [= <value_i:identifier>] ]*
```

**Figure 5** Syntax of the statement CREATE TABLE

The use of the OR REPLACE modifier specifies that, if there is a base view with the name indicated, this must be replaced by the new view. Where, due to the change in view definition, the query capabilities (see section 4.2) of some derived views have been altered (e.g. due to the addition of another field or a query restriction that did not previously exist), DataPort will update the schema and query capabilities of the derived views wherever possible.

Figure 6 shows an example of the creation of a base view using the statement CREATE TABLE. A base view with name 'book' is created, with Spanish internationalization configuration (es_euro) and with two text-type attributes TITLE and AUTHOR.

```
CREATE TABLE book I18N es_euro (
   title:TEXT,
   author:TEXT
);
```

**Figure 6** Example of creating a base view

4.1 MODIFYING A BASE VIEW

By using the sentence ALTER TABLE it is possible to configure the following properties of a base view:

- Its internationalization configuration
- Its cache configuration (CACHE). That is, if the tuples extracted from the source as a result of executing the queries should be stored in the local cache. Section 19.2.2 provides more details on this matter.
- Its swapping configuration (\texttt{SWAP} and \texttt{SWAPSIZE}). That is, the swapping to disk policy for queries that use the base view and involve a large number of tuples. See section 19.2.3 for a detailed description.

- Add, delete or modify a search method. Search methods are composed of rules that represent the restrictions with which a specific query should comply in order to be executed using this search method. Furthermore, each search method has an associated \texttt{wrapper} which contains the data necessary to translate the user query for the source and interpret its response. Section 4.2 provides more details on this matter.

- Renaming the view: \texttt{ALTER TABLE \texttt{<name> RENAME}}...

```sql
ALTER TABLE \texttt{<name:identifier>}
  [ I18N \texttt{<name:identifier>} ]
  [ CACHE { ON | POST | OFF | INVALIDATE } ]
  [ TIMETOLIVEINCACHE \texttt{<seconds:integer>} ]
  [ SWAP { ON | OFF } ]
  [ SWAPSIZE \texttt{<megabytes:integer>} ]
  [ { \texttt{<table search method clause>} }* ]
    | QUERYPLAN = \texttt{<query plan>}
    ]
  [ DESCRIPTION = \texttt{<literal>} ]

| ALTER TABLE \texttt{<name:identifier>}
  ( \texttt{<alter column clause>}+ )

| ALTER TABLE \texttt{<name:identifier> RENAME \texttt{<new_name:identifier>}}

\texttt{<table search method clause>} ::= ...

\texttt{<alter column clause>} ::=...

\texttt{<nullable clause>} ::= \{ \texttt{TRUE | FALSE} \}
```
4.2 QUERY CAPABILITIES: SEARCH METHODS AND WRAPPERS

In the context in which Virtual DataPort works, information sources might offer limited query capabilities. For instance, most web sources can only be queried with constraints imposed by a specific HTML query form.

The description of the query capabilities in Virtual DataPort is done through the so-called search methods. For each view, the administrator can define one or more search methods.

When creating a search method the following elements should be specified: the list of query constraints, the list of output attributes and the wrapper, created beforehand using the statement CREATE WRAPPER, which is responsible for extracting the data from the source.

4.2.1 Query Constraints

To specify search methods, a series of 5-uples, which we will call ‘query constraints’, must be specified. The following elements should be indicated for each query constraint:

- **Attribute** – is an attribute of the relation.
• **Operators** – is the group of operators that can be used in the queries to this source and with this search method. ‘ANY’ represents any operator admitted by the data type of the attribute. If the obligatoriness field (explained later) is ‘NOS’, the value is not specified.

• **Obligatoriness** – four values can be specified: ‘OBL’ indicates that the attribute should obligatorily appear in any query on the source. ‘OPT’ indicates that the attribute can appear or not in the query (it is optional). ‘NOS’ indicates that the queries for this attribute are not permitted in the source.

• **Multiplicity** – indicates how many values the source can be queried simultaneously for the attribute and the given operator. The values ‘ZERO’ (which is equivalent to ‘0’), ‘ONE’ (which is equivalent to ‘1’), ‘ANY’ and any integer number can be specified. If it is not possible to make queries for this attribute (value ‘NOS’ in the obligatoriness field), the value is necessarily ‘0’ or ‘ZERO’. ‘ANY’ indicates a number of values greater than ‘0’ but without an upper limit.

• **Possible Values** – is the list of values with which the attribute can be queried. If it contains the value ‘ANY’ (or it is not specified), this means that it can be queried using any value (within the range associated with the data type of the attribute). If the obligatoriness field is set in the 5-uple to the value ‘NOS’, then it necessarily takes the value of an empty set.

After specifying the query constraints, the attributes that appear in the output of the queries made through the search method are indicated. The output attributes of a search method are specified by enumerating the attributes and separating them with commas.

### 4.2.2 Assigning Wrappers to Search Methods

As can be seen in the syntax of Figure 7, to assign a wrapper to a search method two elements must be indicated: the wrapper type and the name of same.

The type of wrapper indicates the nature of the external source from which the data are extracted. Details on how to create a wrapper are provided in section 18.

### 4.2.3 Example of How a Search Method is Created

An example is shown in Figure 8 of how a search method is added to a relation.

```sql
ALTER TABLE bookview
ADD SEARCHMETHOD bookview_sm1 (CONSTRAINTS (ADD TITLE (any) OBL ANY
ADD AUTHOR (like) OPT ANY
ADD FORMAT NOS ZERO ()
ADD PRICE NOS ZERO ()
)
OUTPUTLIST (TITLE, AUTHOR, FORMAT, PRICE)
WRAPPER (itp booktest)
);
```

Figure 8 Example of how a search method is created with ALTER TABLE

In the example of Figure 8 a search method named bookview_sm1 is added to the base relation called bookview with four query constraints. The search method constraints indicate that to make a query to the source
the attribute TITLE (specifying any number of values) must be searched for. Optionally, a search can be made for the attribute AUTHOR (specifying any number of values) and the operator like. Direct queries for the rest of the attributes (FORMAT, PRICE, etc.) are not admitted. Furthermore, the search method definition indicates that all the attributes appear in the output. Finally, the WWW-type wrapper (wrapper created with ITPilot) called booktest is associated with the search method. It will be responsible for extracting the results, when a query is executed using this search method.

It is important to highlight that although the source does not natively support queries for specific attributes (in the previous example this occurs with FORMAT, PRICE, etc.), Virtual DataPort is capable of executing some of the queries on those attributes through post-processing of the results obtained from the sources. For example, if the server receives the query SELECT * FROM BOOKVIEW WHERE TITLE like 'java' AND FORMAT = 'eBook', Virtual DataPort is capable of responding by extracting from the source the books that contain the word ‘java’ in the title (as the source does allow this query) and later by applying a post-processing to filter the results and remain with just those that also take the value ‘eBook’ in the attribute FORMAT.
5 QUERIES: SELECT STATEMENT

Virtual DataPort allows executing queries on previously created views using the SELECT statement. The syntax is shown in Figure 9. The syntax of this and of all VQL statements can also be queried by using the HELP command (see section 17).

The following subsections describe the use of each of the clauses of the SELECT statement.

```
<query> ::= 
  { <select> | <union select> } 
  [ 
    FILTER <function> [ ; <function> ]* 
    | ORDER BY <order by field> [ ASC | DESC ] [, <order by field> [ ASC | DESC ] ]* 
  ] 
  [ OFFSET <number> { ROW | ROWS } ] 
  [ FETCH { FIRST | NEXT } [ <count> ] { ROW | ROWS } ONLY ] 
  [ CONTEXT ( <context information> [, <context information>] )* ] 
  [ TRACE ]

<select> ::= 
  SELECT [DISTINCT] <select fields>
  FROM <view> [ , <view> ]*
  WHERE <condition>
  GROUP BY <group by field> [ , <group by field> ]*
  HAVING <condition>

<union select> ::= <select> [ UNION <select> ]+
<projected union select> ::= SELECT <select fields> FROM ( <union select> )

<select fields> ::= 
  <select field> [ [ AS ] <alias:identifier> ]
  [, <select field> [ [ AS ] <alias:identifier> ] ]*

<select field> ::= * | <value>

&view ::= 
  <simple view>
  | <join view>
  | ( <select> )

<simple view> ::= 
  <view:identifier> [ [ AS ] <alias:identifier> ]
  | <procedure:identifier>
  { [ <procedureParameter> [, <procedureParameter> ]* ] } 
  [ [ AS ] <alias:identifier> ]
  | <flatten view>

<join view> ::= 
  <inner view1:view> [ [method type] ] [ [order type] ] [ [join type] ]
  JOIN <inner view2:view> ON <condition>
```
| <inner view1:view> NESTED PARALLEL [ <order type> ] [ <join type> ]
| JOIN [ <parallel number:integer> ] <inner view2:view> ON <condition>
| <inner view1:view> [ <method type> ] [ <order type> ]
| NATURAL [ <join type> ] JOIN <inner view2:view>
| <inner view1:view> NESTED PARALLEL [ <order type> ]
| JOIN <inner view2:view> USING ( <field> [, <field> ]* )
| <inner view1:view> [ <method type> ] [ <order type> ]
| NATURAL [ <join type> ] JOIN [ <parallel number:integer> ]
| <inner view2:view>
| <inner view1:view> [ <method type> ] [ <order type> ] [ <join type> ]
| JOIN <inner view2:view> USING ( <field> [, <field> ]* )
| <inner view1:view> NESTED PARALLEL [ <order type> ] [ <join type> ]
| JOIN [ <parallel number:integer> ] <inner view2:view>
| USING ( <field> [, <field>] )
| <inner view1:view> CROSS JOIN <inner view2:view>

<inner view> ::= ( <view> )

<join type> ::= LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ] | INNER

<method type> ::= HASH | NESTED | MERGE

<order type> ::= ORDERED | REVERSEORDER

<flatten view> ::= FLATTEN ( <view identifier> [. <register field>] *. <array field> )
| FLATTEN ( <view identifier> AS <alias>
       [ , <alias> [. <register field>] *. <array field> AS <alias> ]*,
       <alias> [. <register field>] *. <array field> )

[value] ::= (see Figure 4)

<condition> ::= <condition> AND <condition>
| <condition> OR <condition>
| NOT <condition>
| ( <condition> )
| <value> <binary operator> <value> [ , <value> ]*
| <value> <binary operator> ( <value> [ , <value> ]* )
| <value> BETWEEN <value> AND <value>
| <value> <unary operator>

&view identifier> ::= <view name:identifier>
| <view name:literal>

[value] ::= (see Figure 4)

<join condition> ::= (simple join condition) [ AND <simple join condition> ]*
| ( <join condition> )
<simple join condition> ::= <field1:field name> <binary operator> <field2:field name>
| <field2:field name> <binary operator> <field1:field name>
<group by field> ::= { <field name> | <field position:integer> }
<order by field> ::= { <field name> | <field position:integer> }
<unary operator> ::= (see Figure 3)
<binary operator> ::= (see Figure 3)
<field name> ::= (see Figure 4)
<context information> ::= (see Figure 13)
<query plan> ::= { } | [view name:identifier] : <view plans>+ 
<view plans> ::= <view plan> | [ ( [view plan] ) ]+
<view plan> ::= <any method type> <any order type>
  | NESTED PARALLEL [nestedParallelNumber:integer] <any order type>
<any method type> ::= <method type> | ANY
<any order type> ::= <order type> | ANY
&view properties ::= [view name:identifier] : ( <view property> [, <view property> ]* ) ]+
&view property ::= 'begindelimiter' = <literal> [ 'ISDATA' ]

Figure 9  Syntax of the SELECT statement

5.1 FROM CLAUSE

Specification of the origin view is carried out using the FROM clause. In said clause the name of the relation - or relations - from which data are to be extracted is indicated. It is possible to specify aliases for the relations in the FROM clause. Aliases can be used in the other clauses in the SELECT statement and will facilitate the creation of Join conditions. If an alias is indicated for a relation in the FROM clause, the name of the relation should not be used in the rest of the SELECT statement to prefix fields of same; the alias should always be used.

It is possible to use subqueries in the FROM clause. The subquery must be included between brackets.

Example: The following statement uses a subquery that carries out a UNION operation between the internet_inc and phone_inc views:

```
SELECT * FROM (SELECT * FROM internet_inc UNION SELECT * FROM phone_inc)
WHERE taxid='B78596011'
```

If several relations are listed in the FROM clause without separating them from the JOIN clause, then their cross product will be performed. The following subsection deals with the different join operations available.

The FROM clause may also contain calls to stored procedures. The results returned by the calling up of a procedure will be dealt with in this case like the tuples of a view. See section 9 for more details.
5.1.1 Join Operations

The Join operation combines records from two or more views. The following construction must be used to do so:

FROM view1 JOIN view2 ON (joinCondition)

where joinCondition specifies the required join condition. Usually, this condition only includes comparisons between the fields of the views involved in the JOIN. But it can also include expressions with functions, comparisons with literals, etc.

The following modifiers can be used on the JOIN clause:

- **INNER**: The join operation made will be of the inner type. The ‘inner joins’ only include in the result the tuples built from the tuples of both relations associated according to the join conditions. This is the most common join type and is used by default. **Examples**:
  
  FROM view1 JOIN view2 ON (joinCondition)
  FROM view1 INNER JOIN view2 ON (joinCondition)

- **OUTER**: The join operation made will be of the outer type. There are three options for ‘outer’ joins (one of them must always be used): FULL, LEFT and RIGHT. If the FULL option is used, the tuples of both relations will be included in the result, although they do not have an associated tuple in the other relation according to the join condition; the attributes of the other relation will be completed with NULL in the resulting tuple.
  
  If the LEFT option is used, only the tuples of the first relation that do not have associated tuples in the second are included. If the RIGHT option is used, only the tuples of the second relation that do not have associated tuples in the first are included. **Examples**:
  
  FROM view1 FULL OUTER JOIN view2 ON (joinCondition)
  FROM view1 LEFT OUTER JOIN view2 ON (joinCondition)
  FROM view1 RIGHT OUTER JOIN view2 ON (joinCondition)

- **NATURAL**: The natural join operation will be executed. Conditions will not be indicated in this type of join, as this will be done by associating the attributes with the same name in both input relations using the operator ‘=’. This can be used with both ‘inner’ and ‘outer’ joins. **Examples**:
  
  FROM view1 NATURAL JOIN view2
  FROM view1 NATURAL LEFT OUTER JOIN view2

- **CROSS**: The cross product of the specified views will be made. This is equivalent to listing the relations in the FROM clause without using JOIN. **Example**:
  
  FROM view1 CROSS JOIN view2

Instead of specifying a join condition, it is also possible to use the USING clause to specify a list of attributes with the same name and type in both relations. If any of the attributes specified does not exist in some branch of the join tree, or types are not coincident in both branches, an error will be raised. **Example**:

FROM view1 JOIN view2 USING {attribute1,…,attributeN}

Lastly, it is also possible to establish an execution strategy for a specific join operation. See section 19.2.1 for more details on this matter.
5.1.2 Flatten View (Flattening Data Structures)

Denodo Virtual DataPort supports the modeling of hierarchical data types through the use of the types register and array (see section 19.1).

In Virtual DataPort, an array-type element must be thought of as a subrelation. A DataPort array will always have a register type internally associated. Each subelement contained in the array will belong to this register data type. Hence, the fields of this register may be seen as the schema of the subrelation being modeled.

You may wish to “flatten” a compound field that contains an array of registers. This is particularly frequent when processing XML-type sources and Web services. This section describes how this is done.

Imagine that we have a Web Service with a getAverageMonthlySales operation. This operation receives no input parameters and returns data on the monthly sales of all the clients of a company through an array of objects, where each object has two properties: taxId and revenue.

The base relation created on this new operation has one single attribute of the type array containing register-type elements and one sole tuple, where all the data returned by the Web service is found. For combination of data with other sources a view with two attributes (taxId and revenue) and one tuple for each client may be much more useful. This can be achieved through a “flattening” operation on the original view. The process is described below.

In the FROM clause a special constructor (FLATTEN) can be used to define queries on “flattened” views of views with compound data types (see section 19.1). The constructor FLATTEN allows tuples to be generated from the compound subfields of array type of a specific view. Its syntax (see Figure 10) allows the following alternatives:

- Specifying the name of an attribute of array type, a view is generated that has as its schema that of the register contained in the indicated array. The specified array subelement can be inside one register (or even several nested registers), but it cannot be nested inside other array.
- Specifying the name of a view and an alias it is possible to obtain the flattened representation of an array (even when it is nested inside other arrays). Furthermore, in this case the remaining fields of the view are preserved.

The syntax is specified by initially indicating an alias for the original view and then the array element on which the FLATTEN operation is to be applied. To apply to an array that is nested inside another, an alias must be added to the parent array; the array we wish to flatten will be specified by indicating a path from the alias (that is, the container array is specified using analogous system as if it were a view containing the inner array). To traverse more levels of nested array elements continue in a similar manner.

The resulting schema will contain the fields of the original view (except that on which the FLATTEN operation is carried out) and all the elements of all the registers involved in the flattening operation.

```
<flatten view> ::= 
   FLATTEN ( <view name:identifier>[.<register field>]*.<array field> ) 
   | FLATTEN ( <view name:identifier> AS <alias>, 
               [ <alias>[.<register field>]*.<array field> AS <alias> ]*, 
               <alias>[.<register field>]*.<array field> )
```

**Figure 10** Syntax of a FLATTEN view

**Example:** Imagine that we have the base relation AVERAGE_REVENUE_ARRAY the schema of which is comprised of a field of the type array of registers called RETURN. Each register contains two fields: TAXID and REVENUE. The following statement returns the “flattened” contents of AVERAGE_REVENUE_ARRAY:

```
SELECT TAXID, REVENUE FROM FLATTEN (AVERAGE_REVENUE_ARRAY AS V, V.RETURN)
```
### 5.2 SELECT CLAUSE

The `SELECT` clause indicates the attributes to be obtained from the relations specified in the `FROM` clause.

If the character "*" is specified in the `SELECT` clause, this means that all the attributes of the views to which the query is made are selected.

Aliases may also be defined for the columns obtained, thus allowing the name of any attribute to be modified. In the case of derived attributes (see section 5.2.1), if an alias is not specified, Virtual Dataport will automatically generate a name for the new attribute.

In the queries and views, no two fields with the same name are allowed, so it would be necessary to rename any of them (by using aliases).

Finally, the `DISTINCT` modifier may be included. In this case, all duplicated tuples will be deleted from the result.

#### 5.2.1 Derived Attributes

The `SELECT` clause may include derived attributes. These attributes are created by evaluating an expression that may use functions, constants and the values of other attributes.

A description of the functions supported by Virtual Dataport can be found in section 3.7. Detailed examples of the use of each function can be found in section 20.1.

Some examples of how to use derived attribute functions are shown below. The following query obtains a column named `newSalary` containing the result of adding 1000 to the values contained in the `salary` column of the `emp` view.

```
SELECT SUM(1000, salary) newSalary
FROM emp;
```

And the following example shows how to use a nested function as parameter:

```
SELECT NAME, SUM(SALARY, DIV(SALARY,1000)) salaries
FROM emp;
```

### 5.3 WHERE CLAUSE

The `WHERE` clause specifies the conditions the results of the query should comply with. The syntax for specifying conditions is shown in Figure 11.
A condition is a sequence of condition elements separated by the logical operators AND, OR, or NOT. At evaluation time, it obtains a boolean result. The conditions can be grouped between the symbols ‘(‘ and ‘)’ to vary their priority.

A condition is comprised of three elements: a left-side operator which will be the one to which the condition is applied, an operator and zero, one or several right-side operands, depending on the operator used. The comparison operators supported by Virtual DataPort are specified in section 3.6; they include operators of equality, greater/lesser comparison, string contention, etc.

A condition operand can be the name of an attribute, a constant, an expression to be evaluated or a compound value (see section 5.3.1).

5.3.1 Conditions with Compound Values

The ROW constructor allows creating register-type compound values (see section 19.1 for more detail about DataPort compound types). For example:

```
ROW (value1,...,valueN)
```

would create a register-type value with N fields. Each specified value may be an attribute, a literal, a number, a logical value, an expression to evaluate or a new ROW element. Each register field created will be of the corresponding value data type.

It is also possible to create DataPort array types by using ROW combined with the constructors ‘{’ and ‘}’. For example:

```
{ROW (value1,...,valueN), ROW (valueN+1,...,value2N)}
```

would create an array-type value containing two register values.

**NOTE**: See Figure 4 for a formal description of the compound value creation syntax.

Conditions with compound values can only be used with equality ‘=’ and inequality ‘<>’ operators. Both operands must have compatible types for the comparison to be possible.

5.4 GROUP BY CLAUSE

The GROUP BY clause allows grouping the results of a query by the values of a series of attributes, obtaining for each one of these groups one sole tuple in the results. The attributes with which the group-by operation is to be carried out are specified in the GROUP BY clause. If group-by attributes are not specified (without GROUP-BY clause), but aggregation functions are indicated in the SELECT clause, then all the results obtained by the SELECT statement would form one single group.
When the `GROUP BY` clause is specified in a query, the content of the `SELECT` clause is restricted. Only the attributes specified in the `GROUP BY` clause can be specified in it. The remainder of the attributes can only appear as parameters of aggregation functions. When an aggregation function is specified in the `SELECT` clause, an alias must be indicated for the new attribute. Where this is not done, an alias is generated automatically which will be the name of the applied function.

In a group-by view, derived attribute functions can also appear in the `SELECT` clause, although only applied to aggregation fields or functions.

### 5.4.1 Use of Aggregation Functions

An aggregation function is applied to the tuples belonging to a group resulting from a `GROUP BY` operation and calculates an aggregated value from same. The aggregation functions that exist in Virtual DataPort are enumerated in section 3.7.4.

The aggregation functions follow the general syntax of the predefined functions (see section 3.8), but only the name of the attribute subject to alteration is admitted as a parameter (nested functions are not admitted either). The `ALL/DISTINCT` modifiers can also be specified.

One exception is the `COUNT()` aggregation function that can receive as a parameter the special character “*” to indicate that it should return the number of tuples that belong to each group.

For example, given a relation `emp` representing the employees of a company that contains an attribute `department` which indicates to which department each employee belongs, to obtain the different departments together with the number of employees that belong to each one of them, the following query would be executed:

```sql
SELECT count(*) AS numOfWorkers, department
FROM emp
GROUP BY department;
```

Or, using the alias of the field:

```sql
SELECT count(*) AS numOfWorkers, department AS dept_name
FROM emp
GROUP BY dept_name;
```

### 5.5 HAVING CLAUSE

The `HAVING` clause specifies filtering conditions on the results returned by a query using the `GROUP BY` clause.

For example, continuing with the example from the previous section, to obtain only the data corresponding to departments with more than 10 employees, the following query could be made:

```sql
SELECT COUNT(*) AS numOfWorkers, department
FROM emp
GROUP BY department
HAVING COUNT(*)>10
```

### 5.6 UNION CLAUSE

The `UNION` clause allows obtaining a new view containing the tuples from another two views or queries. This corresponds with the relational algebra union operation but with some differences. In principle, to execute a
relational algebra union all the relations must have the same schema, i.e. the same attributes. However, in Virtual DataPort if some of the views have an attribute that the others do not have, this is added to the resulting view (this corresponds to the relational operation called extended union).

Furthermore, in this case the union includes repeated rows, that is, if a row is in two tables, the tuple appears twice in the resulting view. The modifier DISTINCT can be used in the SELECT clause to avoid this.

### 5.6.1 Specifying Projections in UNION Queries

The fields to be projected from a union view can be indicated in the SELECT statements of Virtual DataPort; the syntax is shown in Figure 12.

```
<union select> ::= <select> [ UNION <select> ]+
<projected union select> ::= SELECT <select fields> FROM ( <union select> )
```

*Figure 12* Syntax for a projection of the result of an union

### 5.7 ORDER BY CLAUSE

In the SELECT command the ORDER BY clause can be used to indicate that the result should be obtained ordered according to a list of attributes.

The ORDER BY clause is followed by the attribute or attributes of the final view for which the tuples are to be sorted and the ascending or descending order to be used in each attribute. By default, the results are shown in an ascending order. For example, the following query obtains the employees ordered according to the attribute `pay` in a descending order.

```
SELECT * FROM emp ORDER BY pay DESC;
```

It is also possible to specify the sort attributes by their order number in the SELECT clause. For example:

```
SELECT name,pay FROM emp ORDER BY 2 DESC;
```

In general, the results of a query using Virtual DataPort are processed in an asynchronous manner, i.e. the results are obtained as they are extracted from the sources, without it being necessary to wait for all the results to be available to process those that have already arrived. However, if an ORDER BY clause is specified in a query, the result of the query is obtained in a synchronous manner (i.e. no result can be accessed until all have been obtained).

### 5.8 OFFSET AND FETCH

The OFFSET and FETCH clauses limit the number of rows obtained when executing a query. Use OFFSET `<number>` { ROW | ROWS } to skip the first `n` rows of the result set. Use FETCH { FIRST | NEXT } [ `<count>` ] { ROW | ROWS } ONLY to obtain only `<count>` rows of the result set.

You can also use both, combined. For example,

```
SELECT ... FROM ...  
OFFSET 10 ROWS  
FETCH NEXT 10 ROWS ONLY;
```

executes the query and returns the rows number 10 to number 20 (both included). The first row is row number 0.
If you use `FETCH` without `<count>`, the Server only returns one row. For example:

```
SELECT ... FROM ...
FETCH NEXT ROW ONLY;
```

returns the first row of the result set.

The parameters `ROW` and `ROWS` have the same meaning and can be used indistinctly. `FIRST` and `NEXT` can also be used indistinctly.

## 5.9 CONTEXT CLAUSE

The `CONTEXT` clause is used to modify certain configuration preferences to execute a specific query, without overriding the values configured by default.

In general, the `CONTEXT` clause receives key-value pairs (separated by commas), where the `key` is the name of the execution characteristic to be modified and the `value` indicates the new value for said characteristic. Both key and value are literals, so they must be set with quotation marks or double quotation marks. The name of the key is not case-sensitive, while in the case of the value it depends on the characteristic it represents (see Figure 13 for a formal description of the syntax). The execution characteristics that can be configured through `CONTEXT` are:

- **Cache**: The use of result caches, i.e. if the results of previous queries should be used to resolve the query. This property can take the values “ON” (to use the cache according to the current configuration of the views involved on the query) and “OFF” (to deactivate the cache for the query). By default, this is “ON”. Please see section 19.2.2 for more information.

- **i18n**: Internationalization configuration for the results of the query. This property takes the name of a valid internationalization configuration as a value (e.g. `es_euro`).

**Example**: the following statement obtains all rows from view `V` setting the `us_pst` internationalization configuration only for this query:

```
SELECT * FROM V CONTEXT ('i18n' = 'us_pst')
```

- **noDelegateViews**: List of views that will not be delegated to the data source, in the execution of the query. There are scenarios where a data combination can be delegated to a source but we do not want to do so (e.g. bad performance/limited resources of the source). In these scenarios, it is useful to specify if we do not want to delegate a certain view.

For example, we have a view `incidences` that is the join of the JDBC base views `internet_inc` and `phone_inc` that were created over the same data source.

The query `SELECT * FROM incidences` will result in sending the JOIN query to the database:

```
SELECT * FROM phone_inc INNER JOIN internet_inc...
```

If use execute

```
SELECT * FROM incidences CONTEXT('nodelegateviews', 'incidences')
```

Virtual DataPort will send two queries to the database: `SELECT * FROM phone_inc` and `SELECT * FROM internet_inc`.

- **QueryPlan**: This allows different characteristics of the query running plan to be specified in run time. See section 19.2.1 for more details.
• **Swap.** This indicates whether swapping is enabled for the query. This property must take the "ON" value to indicate that the swapping of intermediate results is permitted, while the query is being run. The "OFF" value indicates the opposite. See section 19.2.3 for more details.

• **SwapSize.** This property indicates the maximum size an intermediate result obtained by running this query can reach without swapping to disk. It is given the maximum size (in megabytes) as a parameter. It is only effective where the SWAP ON option has been specified. See section 19.2.3 for more details.

• **ViewProperties.** This enables you to indicate a series of properties for the views forming part of the query tree. At present, only the begindelimiter property is supported. This property can be applied to base views generated based on data sources from delimited files (see section 18.3.6 for a description of these data sources and of the begindelimiter parameter) to dynamically choose the point from which to begin access to the delimited source file through a regular expression. If isdata is also specified, the delimiter will be considered to form part of the data.

**Example:** Supposing that V2 is a base view created based on a data source of the delimited file type forming part of the V definition tree, the following statement obtains the tuples from the delimited file from the first tuple matching the regular expression specified (in this case, any starting with the string 05/24/2008):

```
SELECT * FROM _V CONTEXT (VIEWPROPERTIES=V2:('begindelimiter'='05/24/2008(.*)' 'ISDATA'))
```

```
<context information> ::= 'i18n' = <literal>                         // e.g. 'es_euro', ...
  | 'cache' = { 'on' | 'off' }                 // 'on' by default
  | 'swap' = { 'on' | 'off' }
  | 'swapsize' = <number>
  | 'var <var name>' = <literal>
  | VIEWPROPERTIES = <view properties>
  | QUERYPLAN = <query plan>

<view properties> ::= [<view name:identifier> : ( <view property> [, <view property>] )* ]+  

<view property> ::= 'begindelimiter' = <literal>

<query plan> ::= (see section 19.2.1.1)

<context value> ::= <number> | <boolean> | <literal>
```

**Figure 13** Syntax of the CONTEXT clause

**NOTE:** The 'View Properties' option is deprecated and should not be used in new applications. If you need to specify at runtime the value for the begindelimiter parameter of a delimited files data source, you can use interpolation variables in the value of such parameter (see section Paths with Interpolation Variables in [ADMIN_GUIDE]).

**NOTE:** Apart from these properties, we can also set the values of the selection conditions' variables of the views involved in the query. The appendix 19.6 explains what selection conditions with variables are.
5.10 TRACE CLAUSE

Using the TRACE clause of the SELECT command, the server will return detailed information about the execution process of a query.

The trace of a statement provides a detailed examination of its execution plan. This plan can be modeled as a tree, where each node represents an intermediate view involved in the execution of a query or an access to a data source via a wrapper.

The TRACE clause shows the most relevant parameters for each node on the query execution tree. The DataPort administration tool (see Administration Guide [ADMIN_GUIDE]) allows examining the trace information using much more user-friendly graphical interface.

Among the parameters displayed by the TRACE clause are:

- **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 until the first tuple of results was obtained.** 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, the exact sub-queries executed on each data source and the connection data used to access each one are indicated.
  
  - For each view-type node, it is indicated whether the cache has been used, whether swapping has been necessary, etc. are indicated.
  
  - A parameter of particular interest for optimization reasons is “No Delegation Cause”. In the views defined based on tables from the same JDBC or ODBC data source, DataPort will try to delegate the entire process to the source database, obtaining all the tuples from the view through a single query. This strategy may save a significant amount of execution time in complex views. When DataPort is unable to delegate the entire process of a certain query to a source database, it will indicate a reason in this parameter. For example, the query may use an expression that includes a function that is not supported by the source database, which will force
DataPort to post-process the results obtained. In light of the reason where the processing could not be delegated, it may be possible to rewrite the view so that it can be delegated.

- **Error conditions.** The trace also indicates any errors produced during node execution.

As an example, Figure 18 shows the trace of the following query execution:

```
SELECT * FROM INTERNET_INC TRACE
```

INTERNET_INC is a base view created on a table of the same name accessible via a JDBC data source.
BASE PLAN
(name = INTERNET_INC
startTime = Wed Jan 10 17:50:01 850 GMT+01:00 2007
endTime = Wed Jan 10 17:50:04 063 GMT+01:00 2007
responseTime = Wed Jan 10 17:50:04 053 GMT+01:00 2007
numRows = 4
state = OK
completed = true
fields = [IINC_ID, SUMMARY, TTIME, TAXID, SPECIFIC_FIELD1,
SPECIFIC_FIELD2]
search conditions = []
filter conditions = []
numOfFilteredTuples = 0
numOfDuplicatedTuples = 0
numOfSwappedTuples = 0
swapping = false

JDBC WRAPPER
(name = internet_inc
startTime = Wed Jan 10 17:50:02 070 GMT+01:00 2007
endTime = Wed Jan 10 17:50:04 063 GMT+01:00 2007
responseTime = Wed Jan 10 17:50:04 063 GMT+01:00 2007
numRows = 4
state = OK
completed = true
searchConditions = []
orderByFields = []
projectedFields = [IINC_ID, SUMMARY, TTIME, TAXID,
SPECIFIC_FIELD1, SPECIFIC_FIELD2]

JDBC ROUTE
(name = internet_inc
startTime = Wed Jan 10 17:50:03 782 GMT+01:00 2007
endTime = Wed Jan 10 17:50:04 063 GMT+01:00 2007
responseTime = Wed Jan 10 17:50:04 063 GMT+01:00 2007
numRows = 4
state = OK
completed = true
sqlSentence = SELECT t0.iinc_id, t0.summary,
t0.ttime, t0.taxId, t0.specific_field1, t0.specific_field2 FROM
test_vdb.internet_inc t0
parameters = []
DBUri = jdbc:mysql://localhost/test_vdb
userName = vdb
connectionTime = 0
cachedStatus = false
)}

Figure 14 Execution trace
To analyze the query execution trace, the use of the DataPort administration tool is strongly recommended (see Administration Guide [ADMIN_GUIDE]). This tool displays the execution trace in graphic form.

5.11 CASE CLAUSE

The CASE clause provides an if-then-else type of logic. The syntax for the case clause is shown in Figure 15.

```
CASE <value:expression> WHEN <compare_value:expression>
    THEN result [ WHEN <compare_value:expression> THEN result ...]
    [ELSE result]
END

CASE WHEN <condition>
    THEN result [ WHEN <condition> THEN result ...]
    [ELSE result]
END

<condition> ::=<condition> AND <condition>
| <condition> OR <condition>
| NOT <condition>
| ( <condition> )
| <value> <binary operator> <value> [ , <value> ]*
| <value> <binary operator> ( <value> [ , <value> ]* )
| <value> BETWEEN <value> AND <value>
| <value> <unary operator>
```

Figure 15 CASE Syntax

The CASE clause can be used in two different ways:

1. CASE evaluates an expression and obtains a value. Then, it compares that value with the expression of every WHEN clause. When it finds a match, returns the result value.

2. CASE evaluates the condition of every WHEN clause until it finds a match. When it does, returns the result value.

In both versions, if there is no ELSE clause and there isn’t any matching condition, CASE returns NULL.

All the result expressions must have a compatible type. So, for instance, it is not possible that one result has type boolean and other, integer. But it is possible that one result has type integer and the other float.

See the appendix section 20.4 for more examples on how to use CASE.
6 DEFINING A DERIVED VIEW

The administrator can use the base views of the system to define new relations. These new relations are called derived views.

Derived views are created through the statement CREATE VIEW. The syntax is shown in Figure 16.

```
CREATE [ OR REPLACE ] VIEW <name:identifier> AS <select>
  [ORDER BY <field name> [ ASC | DESC ] [, <field name> [ ASC | DESC ] ]* ]
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
  [ CONTEXT ( <context information> [, <context information>]* ) ]

<select> ::= (see Figure 9)
<context information> ::= (see Figure 9)
```

**Figure 16** Syntax of the statement CREATE VIEW

As can be seen, a name and the query that defines it are specified, when creating a view. The query is specified using the syntax of the SELECT statement, which has been explained in detail in section 5.

Therefore, the administrator can create new derived views by combining other existing views using operators such as unions, joins, Cartesian products, selections, projections or group-by operations.

Furthermore, existing derived views can be also used to create new derived views, therefore allowing view trees with as many levels as required.

For example, considering the views A, B and R as base relations (those that directly access the sources to obtain their data) the administrator can define a view G as the join of the result of applying the union (A, B) with R, as can be seen in Figure 17.

```
CREATE [ OR REPLACE ] VIEW <name:identifier> AS <select>
  [ORDER BY <field name> [ ASC | DESC ] [, <field name> [ ASC | DESC ] ]* ]
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
  [ CONTEXT ( <context information> [, <context information>]* ) ]

<select> ::= (see Figure 9)
<context information> ::= (see Figure 9)
```

**Figure 17** Example of how a view is defined in accordance with others

The optional ORDER BY clause indicates that when querying the view, the results will be ordered by those field(s). ASC sorts in ascending order and DESC, in descending order. If ASC or DESC are omitted, DataPort will sort in ascending order.

The creation of a view also accepts the SQL standard clause WITH CHECK OPTION, which is related to the updating of view contents using INSERT / UPDATE / DELETE statements. The function of this modifier is described in detail in section 7.4.
The use of the `OR REPLACE` modifier specifies that, if there is a view with the name indicated, this must be replaced by the new view. Where, due to the change in view definition, the query capabilities (see section 5.2) of some derived views have been altered (e.g. due to the addition of another field or a query restriction that did not previously exist), DataPort will update the schema and query capabilities of the upper level derived views wherever possible.

### 6.1 MODIFYING A DERIVED VIEW

Once a derived view has been created, it is possible to modify some of its properties:

- Its internationalization options through option `i18n` (see section 3.1.1),
- Cache configuration through the `CACHE` and `TIMETOLIVEINCACHE` options (see section 19.2.2),
- DataPort swapping policy configuration through the `SWAP` and `SWAPSIZE` options (see section 19.2.3),
- Execution strategy configuration for the joins involved in defining the view through the `QUERYPLAN` option (see section 19.2.1),
- Rename the view: `ALTER VIEW ... RENAME ...`

The statement `ALTER VIEW` allows the Virtual DataPort administrator to execute all these operations. The syntax is shown in Figure 18.

```sql
ALTER VIEW <name:identifier>
    [ CACHE { ON | POST | OFF | INVALIDATE [WHERE <condition>] [CASCADE]} ]
    [ TIMETOLIVEINCACHE <seconds:integer> ]
    [ SWAP { ON | OFF } ]
    [ SWAPSIZE <megabytes:integer> ]
    [ QUERYPLAN = <query plan> ]

| ALTER VIEW <name:identifier> RENAME <new_name:identifier>

<operator> includes “any” to represent any operator.
<query plan> ::= (see section 19.2.1.1)
```

Figure 18 Syntax of the statement `ALTER VIEW`
7 INSERTIONS, UPDATES AND DELETION OF VIEWS

INSERT / UPDATE / DELETE statements allow respectively inserting, updating and deleting tuples in a view. They will directly update the underlying data source.

These statements can only be executed on views created using database-type sources (JDBC/ODBC sources. See sections 18.3.1 and 18.3.2) or CUSTOM-type sources (see section 18.4.10). Furthermore, these views must be updateable according to the definition of standard SQL-92.

In short, an updatable view must verify the following restrictions:

- The SELECT statement used in the view definition cannot include DISTINCT, GROUP BY or HAVING.
- The FROM clause of the statement refers to exactly one view. This view must be either a base view or an updatable view. In the case of a base view, it must either belong to a database (JDBC/ODBC Data Sources. See sections 18.3.1 and 18.3.2) or use a CUSTOM-type wrapper providing support for updates (see section 18.4.10).
- The derived attributes cannot be updated.
- A view using an aggregation function (even when there is not GROUP BY clause) cannot be updated.

7.1 INSERT STATEMENT

The INSERT statement allows inserting a new tuple in a view, updating the underlying data source directly. Figure 19 shows its syntax.

```
INSERT INTO <name:identifier> (<field name>[, <field name>]*)
VALUES (<value>[, <value>]*)
[ CONTEXT ( <context information> [, <context information>]* ) ]
[ TRACE ]
```

```
INSERT INTO <name:identifier>
SET <field name> = <value> [, <field name> = <value> ]*
[ CONTEXT ( <context information> [, <context information>]* ) ]
[ TRACE ]
```

```
<field name> ::= <identifier>[.<identifier>]
<value> ::=<identifier>[
| <number>
| <boolean>
| <literal>
| <function>
```

Figure 19 Syntax of the INSERT statement
For example, the following statement adds a new tuple to the internet_inc view:

```
INSERT INTO internet_inc (iinc_id, summary, ttime, taxid, specific_field1, specific_field2)
VALUES (6, "Error in ADSL Router", "31-mar-2005 22h 35m 24s", "B78596015", "5", "6")
```

As a result of executing this statement, a new tuple will be added in the source database to the table associated with the internet_inc view.

It is also possible to use the alternative syntax:

```
INSERT INTO internet_inc
SET iinc_id=6, summary="Error in ADSL router", ttime="31-mar-2005 22h 35m 24s", taxid="B78596015", specific_field1="5", specific_field2="6"
```

### 7.2 UPDATE STATEMENT

The UPDATE statement allows modifying the value of certain attributes in all tuples of a view that verify a certain condition, directly updating the underlying data source. Figure 20 shows its syntax:

```
UPDATE <name:identifier>
SET (<field name>[, <field name>]) = (<value>[, <value>])
[ WHERE <condition> ]
[ CONTEXT ( <context information>[, <context information>]) ]
[ TRACE ]
```

For example, the following statement alters the tuples of the internet_inc view where the value for the iinc_id attribute is 6, setting to 10 its value for specific_field1 and specific_field2 attributes:
UPDATE internet_inc
SET (specific_field1, specific_field2) = ("10","10")
WHERE iinc_id=6

As a result of executing this statement, the corresponding tuples in the source database will be altered in the table associated with the internet_inc view.

It is also possible to use the alternative syntax:

UPDATE internet_inc
SET specific_field1="10", specific_field2="10"
WHERE iinc_id=6

7.3 DELETE STATEMENT

The DELETE statement deletes the tuples of a view that verify a certain condition by updating the underlying data source. Figure 21 shows its syntax:

DELETE FROM <name:identifier> [ WHERE <condition> ]
[ CONTEXT ( <context information> [, <context information>]* ) ]
[ TRACE ]

<condition> ::= 
  <condition> AND <condition>
| <condition> OR <condition>
| NOT <condition>
| ( <condition> )
| <value> <binary operator> <value> [ , <value> ]*
| <value> <unary operator>

Figure 21 Syntax of the DELETE statement

For example, the following statement deletes the tuples of the internet_inc view where the value for the iinc_id attribute is greater than 4:

DELETE FROM internet_inc WHERE iinc_id>4

As a result of executing this statement, the corresponding tuples in the source database will be deleted in the table associated with the internet_inc view.

Note: this statement does not work with Microsoft Excel sources because of limitations in the Excel ODBC Driver provided by Microsoft Windows.

7.4 USE OF THE WITH CHECK OPTION

On creating a view, DataPort also supports the use of the SQL standard clause WITH CHECK OPTION [CASCADE]. If a view has been created using this option, the data updates that are inconsistent with the definition of the view will be rejected and DataPort will return an error message. For example, if the incidences_acme view is defined using the following statement:

CREATE VIEW incidences_acme AS
SELECT * FROM Internet_inc WHERE taxid="B78596011"
WITH CHECK OPTION
Then, on executing the following insert statement, an error message will be obtained, as the value indicated for the \texttt{taxid} attribute is inconsistent with the selection condition used to define \texttt{incidences_acme}.

\begin{verbatim}
INSERT INTO incidences_acme (iinc_id, summary, ttime, taxid, specific_field1, specific_field2)
VALUES (6,"Error in ADSL Router", "31-mar-2005 22h 35m 24s", "B78596015", "5", "6")
\end{verbatim}

The \texttt{CASCADE} modifier is used so that this check is also applied to the conditions of lower level views (see Figure 16). Where not indicated, the check will only be made using the conditions defined in this view.
8 **TRANSACTIONS IN VIRTUAL DATAPORT**

DataPort allows defining transactions, using the following clauses:

- **BEGIN.** Begins a transaction.
- **COMMIT.** Confirms the active transaction.
- **ROLLBACK.** Undoes the changes made to the active transaction.

Transactions in DataPort are distributed by nature. Therefore, only data sources implementing the Two-Phase-Commit protocol can take part in them. Most commercial database managers use this protocol. So, usually the views participating in transactions are the ones which data source type is JDBC or ODBC (see sections 18.3.1 and 18.3.2).

In addition, **CUSTOM**-type wrappers and stored procedures can also take part in transactions, provided that the necessary operations to do so are implemented (see sections 18.4.13 and 9.3).

It is possible to specify whether a data source supports distributed transactions by using the `supportsDistributedTransactions` property of the source configuration (see sections 18.3.13 and 18.4.16).
9 STORED PROCEDURES

DataPort supports the creation of stored procedures written in JAVA language. This section describes how to import them in DataPort using VQL language. Section 19.3.2 describes how to create a new stored procedure. The DataPort distribution contains different examples of stored procedures (including their source code) in the DENODO_HOME/samples/vdp/storedProcedures path. The README file in this path contains instructions to compile and install these procedures.

9.1 IMPORTING A STORED PROCEDURE

The statement CREATE PROCEDURE allows adding a new stored procedure to the DataPort server. Figure 22 shows its syntax.

```
CREATE [OR REPLACE] PROCEDURE <name:identifier>
    [CLASSNAME <className:literal>]
    [CLASSPATH <classPath:literal>]
    [JARS <jar name:literal> [, <jar name:literal>]* ]
```

**Figure 22** CREATE PROCEDURE syntax

The CLASSNAME clause indicates the name of the JAVA class implementing the stored procedure. This class must be present in a library loaded into the DataPort server (see the Importing Extensions section of the Administration Guide [ADMIN_GUIDE]).

The CLASSPATH clause can optionally be used to indicate additional libraries used by the stored procedure and that are not in the server’s CLASSPATH.

The use of the OR REPLACE modifier specifies that, if there is a procedure with the name indicated, this must be replaced by the new procedure. This will lead to the recalculation of the schemas and query capabilities of the derived views using the procedure.

Once created, a stored procedure can be modified using the ALTER PROCEDURE statement, the syntax of which is shown in Figure 23.

```
ALTER PROCEDURE <name:identifier>
    [CLASSNAME <className:literal>]
    [CLASSPATH <classPath:literal>]
    [JARS <jar name:literal> [, <jar name:literal>]* ]
```

**Figure 23** ALTER PROCEDURE syntax

The meaning of the CLASSNAME and CLASSPATH clauses is the same for the CREATE PROCEDURE statement.

9.2 USE OF STORED PROCEDURES

A stored procedure is called up using the CALL statement. The syntax is shown in Figure 24.
CALL <procedureName:identifier> (  
  [,<paramValue:literal>]* 
) 
[ CONTEXT ( "il8n" = <literal> ) ] [ TRACE ]

Figure 24 Syntax of the CALL statement

For example, the following statement calls up the stored procedure DropIncidence, passing it a single
numeric-type parameter:

```sql
CALL DropIncidence(5)
```

The result of the execution of a stored procedure is a row or a list of rows with one attribute for each stored
procedure output parameter. In our last example, the stored procedure returns a row with a single attribute
specifying the number of deleted incidences.

Stored procedures can be used in the FROM clause of a SELECT statement. The values returned by the procedure
are, in this case, processed like the tuples of a view. For example:

```sql
SELECT avgrevenue FROM CalculateAvgRevenue({ROW("B78596011"),ROW("B78596012")})
```

In this case, we have used the stored procedure CalculateAvgRevenue as an example. This procedure
receives a parameter of array of registers type as input. Each register contains a single field, which corresponds to
a client’s Tax ID. This procedure returns a single tuple of results with an attribute called `avgrevenue` that contains
the average revenue of the indicated clients.

In addition, when a stored procedure is called using the FROM clause of a query, its output schema also includes the
stored procedure input parameters (in this case, `taxid_list` attribute). So, a stored procedure can be used in the same
way than a view, by specifying the values for its input parameters as conditions in the WHERE clause. The last
sentence would be equivalent to the following:

```sql
SELECT avgrevenue FROM CalculateAvgRevenue()
WHERE taxid_list = { ROW("B78596011"),ROW("B78596012") }
```

A stored procedure can have optional parameters. In that case, it is possible to pass `NULL` to these parameters and
the stored procedure will ignore them.

9.3 PRE-DEFINED PROCEDURES

DataPort includes the following pre-defined stored procedures:

- **WriteLogInfo** (String text). This writes a message in the DataPort server log at info
  level.

- **WriteLogError** (String text). This writes a message in the DataPort server log at error
  level.

- **Wait** (long timeInMillis). This waits the specified time (in milliseconds).
• LogController (String logCategory, String logLevel). This changes the log level for a certain log category. The change is non-persistent between different server executions.

• Dual (). This procedure does not do anything. It has only one field, named DUMMY, of type text, it has not input parameters and it returns only one empty row. It allows low cost ping queries against VDP, like SELECT * FROM DUAL(). Also, it can be used to evaluate VQL functions on the server. For instance, it can be used to obtain the current time in the server: SELECT NOW() FROM DUAL().

• CATALOG_VIEWS(...): This procedure searches for views or base views in the catalog of the current Virtual DataPort database. The syntax for invoking this procedure is:

```sql
CALL CATALOG_VIEWS (<view name>,
<creator username>,
<last modifier username>,
<begin creation date:date>,
<end creation date:date>,
<init last modification date:date>,
<end last modification date:date>,
$view type:int>,
<swap active:int>,
<cache status:int>,
<description>);
```

It is possible to pass the value NULL as parameters. Following, there is an explanation of some of the parameters of this procedure:

- For the parameters of type text, the comparison is of type "contains". I.e. if the creator username parameter is "adm", the procedure will search for all the view which creator’s username name of the creator contains the text "adm".
- The search by parameters of type date is done in intervals. That is, the procedure will search the views created between the begin creation date and the end creation date.
- view type. Allowed values are: 0 = base view; 1 = view.
- swap active. Allowed values are: 0 = swap disabled for this view; 1 = swap enabled for this view.
- cache status. Allowed values are: 0 = cache disabled for this view; 1 = cache enabled for this view.

• CATALOG_ELEMENTS(...): This procedure searches elements in the catalog of the current Virtual DataPort database. The syntax for invoking this procedure is:

```sql
CALL CATALOG_ELEMENTS (<element name>,
['{Datasources, WebServices, Widgets, Wrappers, storedProcedures}'],
<creator username>,
<last modifier username>,
<begin creation date:date>,
<end creation date:date>,
<init last modification date:date>,
<end last modification date:date>,
<description>);
```

It is possible to pass the value NULL as parameters. Following, there is an explanation for some of the parameters of this procedure:

- For the parameters description and element name, the comparison is of type "contains". I.e. if the description parameter is "desc", the procedure will search for all the elements which description contains the text "desc".
The search by parameters of type date is done in intervals. That is, the procedure will search the views created between the **begin creation date** and the **end creation date**.
10  DEFINING OTHER ELEMENTS OF THE CATALOG

10.1  DEFINING A DATA TYPE

The Virtual DataPort catalog incorporates a series of predefined data types (see section 3.1). As already mentioned, the data types included can be divided into two groups: the basic types and the compound types.

Virtual DataPort allows new compound data types to be defined through the statement CREATE TYPE, i.e. it allows data types of the types array, enumerated and register to be created. See section 19.1 for a more detailed explanation of how to handle the compound types array and register.

Figure 25 shows the syntax of the statement CREATE TYPE.

```
CREATE [ OR REPLACE ] TYPE <name:identifier> AS { <array> | <enumerated> | <register> }

<array> ::= ARRAY OF <register>
<enumerated> ::= ENUMERATED OF ( <literal> [, <literal>] )* 
<register> ::= REGISTER OF ( <name:identifier>:<type:identifier> [, <name:identifier>:<type:identifier>] )* 
```

**Figure 25** Syntax of the statement CREATE TYPE

When creating a data type it must be assigned a unique name that identifies it and differentiates it from the other types that exist.

The data type enumerated are created by enumerating the list of values admitted, separated by commas. In Figure 26 an enumerated data type is created to represent the days of the week.

```
CREATE TYPE daysOfWeek AS ENUMERATED OF ('MONDAY', 'TUESDAY', 'WEDNESDAY', 'THURSDAY',
                                          'FRIDAY', 'SATURDAY', 'SUNDAY');
```

**Figure 26** Creating an enumerated data type

To create a new register type, it is needed to specify the name and data type of the elements it contains. In Figure 27 a register data type is created that contains personal data: name (attribute NAME of text-type), surname (attribute SURNAME of text-type), telephone (attribute PHONE of array_phone-type), salary (attribute PAY of money-type) and birthday (attribute BIRTH of date-type).
CREATE TYPE registerPersonalData AS REGISTER OF {
    NAME: text,
    SURNAME: text,
    PHONE: array_phone,
    PAY: money,
    BIRTH: date
};

Figure 27 Creating a register data type

When defining an array data type the name of the register type of the elements it contains must be indicated. In Figure 28 the array data type called array_phone is created, which encapsulates a list of telephones, where each telephone is represented by an integer. Each element of the array array_phone is of the register-type register_phone. As can be seen, the type register_phone encapsulates an element of the type int called number.

CREATE TYPE registerPhone AS REGISTER OF {
    NUMBER: int
};
CREATE TYPE arrayPhone AS ARRAY OF registerPhone;

Figure 28 Creating a data type array and the register type it contains

The use of the OR REPLACE modifier specifies that, if there is a type with the name indicated, this must be replaced by the new type. If the new type is not compatible with the previous one, the views depending on this type or on any other type using them in its definition will be marked as erroneous. The types are considered compatible if the fields of the new type are a “superset” of the fields of the previous type, i.e. the new type may add new fields but must maintain the previous ones of the same name and type.

10.2 DEFINING A MAP

A map represents a list of key-value pairs. The following types of maps exist:

- simple. This is used with the MAP function (see section 3.7).

- i18n. These represent internationalization configurations referring to specific locations. Some examples of parameters configured through these files are: currency, symbols used as decimal and thousands separators for currency, date format, etc. See section 3.1.1 for more details.

Maps can be created with Virtual DataPort by using the statement CREATE MAP. The syntax is shown in Figure 29. This statement allows creating the previously mentioned different types of maps. It is needed to specify the map type (i18n or simple), the map name and the list of key-value pairs that form the map.

CREATE MAP { I18N|SIMPLE } <name:identifier> (
    [<name:literal> = <value:literal>]* )

Figure 29 Syntax of the statement CREATE MAP

Figure 30 shows an example of how a map of type simple is created.
CREATE MAP SIMPLE daysOfWeek (  'lunes' = 'Monday'
'Martes' = 'Tuesday'
'miercoles' = 'Wednesday'
'jueves' = 'Thursday'
'viernes' = 'Friday'
'sabado' = 'Saturday'
'domingo' = 'Sunday'
);

Figure 30 Creation of an map of type simple

10.3 DEFINING .JAR EXTENSIONS

Stored procedures (see section 9), Custom functions (see section 19.3.1) and Custom wrappers (see section 18.4.13) are implemented using JAVA. The CREATE JAR statement allows for new JAVA libraries (.jar files) implementing any of these elements to be added to the server. Figure 31 shows its syntax.

NOTE: It is strongly recommended that the extensions be loaded using the Virtual DataPort graphic administration tool (see [ADMIN_GUIDE]). The syntax of the CREATE JAR statement is provided as a reference.

CREATE [ OR REPLACE ] JAR <name:identifier> <jar encoded as base64:literal>

Figure 31 Syntax of the CREATE JAR statement

An identifier must be specified for the .jar file along with its contents coded as a string of bytes. The OR REPLACE modifier replaces the file, if it already exists.

10.4 DEFINING JMS LISTENERS

Virtual DataPort can subscribe to a JMS server [JMS] to listen to VQL requests. Therefore, clients, instead of connecting to Virtual DataPort via JDBC, ODBC or a Web service, can send a request to a JMS server, which forwards it to Virtual DataPort. Then, the response is sent back to a queue or a topic of the JMS server, which forwards it to the client/s.

E.g. a client sends a message such as 'SELECT * FROM internet_inc WHERE iinc_id=1' to the JMS server. The server will forward this to Virtual DataPort, which will send a response like:

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <item>
    <iinc_id>1.00</iinc_id>
    <summary>Error in ADSL router</summary>
    <ttime>29-jun-2005 19h 19m 41s</ttime>
    <taxid>B78596011</taxid>
    <specific_field1>1</specific_field1>
    <specific_field2>1</specific_field2>
  </item>
</response>

Figure 32 Response message sent by a JMS listener
If the request is a DML sentence such as `ALTER VIEW incidences CACHE INVALIDATE`, the response will be empty:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<response />
```

**Figure 33** Response message sent to a DML query

The following figures contain the syntax of the various commands to deal with JMS listeners.

```sql
CREATE ( OR REPLACE ) LISTENER JMS <name:identifier>
  VENDOR { ACTIVEMQ | WEBSPHEREMQ | JNDI }
  DESTINATION = <name:literal>
  [ REPLYTO = <name:literal> ]
  { QUEUE | TOPIC }
  [ USER = <name:literal> PASSWORD = <name:literal> ]
  VDPDATABASE = <name:literal>
  VDPUSER = <name:literal>
  ENABLED = { TRUE | FALSE }
  PROPERTIES ( <property> [, <property> ]* );

<property> ::=<key:literal> = <value:literal>
```

**Figure 34** Command to create a JMS listener: CREATE LISTENER JMS

- **VENDOR.** Use the value **JNDI** if the JMS server that this listener will connect to, is neither Apache ActiveMQ nor IBM WebSphere MQ.

- **DESTINATION** is the name of the queue or topic that Virtual DataPort will subscribe to, waiting for requests. Depending on the vendor of the JMS server, we might have to create the JMS destination, or it will be created automatically when the listener tries to subscribe to it.

- **QUEUE** or **TOPIC.** Use one parameter or the other depending on the type of the destination that this listener will connect to.

- **USER** and **PASSWORD.** They are the credentials to access the JMS server.

- **VDPDATABASE** is the Virtual DataPort database that the listener will connect to. JMS listeners, as JDBC clients, have to connect to a specific Virtual DataPort database where they will execute queries.

- **VDPUSER** is the username that Virtual DataPort will use to check if the listener has enough privileges to execute a query. E.g. if our Virtual DataPort server has two users:
  - admin: is an ‘administrator’ so it can access any view of any database.
  - user1: is a ‘normal user’ that only has READ privileges over the database ‘samples’.

If the parameter **VDPUSER** is ‘admin’, the listener will be able to execute any query in the database **VDPDATABASE.** However, if **VDPUSER** is ‘user1’, the parameter **VDPDATABASE** has to be ‘samples’.
because this user can only access that database. Besides, the CREATE / UPDATE / DELETE queries will fail because ‘user1’ only has READ privileges.

- **ENABLED**. **TRUE** to enable the listener. That is, after creating it, the listener will try to connect to the JMS server to listen to requests. **FALSE**, to disable it.

- **PROPERTIES**: List of properties that will be used to obtain a connection to the JMS server. The appendix ‘JMS Connection Details: JNDI Properties and Client Jars’ of the Administration Guide [ADMIN_GUIDE] contains a list of the properties required to connect to the most popular vendors. Usually, the JMS listeners need these properties, at least:
  a. `java.naming.factory.initial` (javax.naming.Context.INITIAL_CONTEXT_FACTORY)
  b. `java.naming.provider.url` (javax.naming.Context.PROVIDER_URL)
  c. `transport.jms.ConnectionFactoryJNDIName`. Name of the connection factory in the JNDI context of the JMS server.

```
ALTER LISTENER JMS <name:identifier>
ENABLED EQ ( true | false )
```

**Figure 35** Command to enable/disable a JMS listener: ALTER LISTENER JMS

If **ENABLED** = **TRUE** and it was **FALSE**, it tries to connect to the JMS server to listen to requests.
If **ENABLED** = **FALSE** and it was **TRUE**, it closes the connection with the JMS server.
11 CREATING DATABASES, USERS AND ACCESS LEVELS

Various key concepts of the Virtual DataPort architecture are described in this section.

Section 11.1 describes the concept of databases as understood in the context of a Virtual DataPort server. Section 11.2 describes the general concepts of user and access right management in DataPort. Finally, section 11.3 describes the VQL commands for managing this structure.

11.1 DATABASES IN VIRTUAL DATAPORT

A Virtual DataPort server can contain various different databases (do not confuse with the possible external databases that can act as data sources). A Virtual DataPort database represents a virtual schema comprised of a series of data sources, wrappers, views and base views.

Each database is independent of the rest of the server databases and, as described in detail in the following section, the different users can have different privileges for each database.

When a Virtual DataPort server is installed, an example database is created called admin, which cannot be deleted.

11.2 USER AND ACCESS STRUCTURES IN VIRTUAL DATAPORT

11.2.1 Types of Users

Denodo Virtual DataPort distinguishes two types of users:

- **Administrators**: They can create, modify and delete databases in a DataPort server without any limitation. Likewise, they can create, modify and delete users. When the server is installed, a default administrator user is created whose user name is admin and whose password is also admin. This user can never be deleted.

- **Normal users**: They cannot create, modify or delete users. They cannot create or delete databases, although they can have connection, reading, creating or writing privileges for one or various databases or specific views contained in same.

11.2.2 Types of Access Rights

Virtual DataPort access rights are applied to a specific user to delimit the actions she/he is permitted to use on databases, stored procedures and views of a specific server. User access rights can be applied globally to a database or specifically to a view or stored procedure in a specific database. Access rights to particular views or stored procedures are applied only if the user does not have the corresponding access right on a global level.

Denodo Virtual DataPort supports the following types of global access rights to databases:

- **Read access**: If a user has this privilege on the whole database, he/she can carry out the following tasks on same:
  - View the list of base relations, stored procedures and/or views of the database (corresponds to the VQL LIST command). If a user does not have read access to a database, but does have it for
some of its views and/or stored procedures, the list command may be executed, but it will only display the group of views and procedures to which the user has read access.

- View information about a base relation, view or stored procedure of the database. For example, access the schema, search methods, cache configuration, swapping configuration, etc., of a base view (corresponds to the VQL DESC command).
- Execute queries to any view and/or stored procedure of the database (corresponds to the VQL SELECT command).

- **Create access**: If a user has this privilege on the whole database, he/she can carry out the following tasks on same:
  - Creating data sources, views, stored procedures and base relations on the database (corresponds to the VQL CREATE command).

- **Write access**: Having write privileges implies that you automatically also have read privileges. If a user has this privilege on a database, he/she can execute the following additional actions on it:
  - Delete any view, stored procedure and/or base relation of the database. He/she can also delete any data source of the database he/she has created, but cannot delete data sources created by other users (corresponds to the VQL DROP command).
  - Modify any view, stored procedure and/or base relation of the database. Can also modify any data source of the database he/she has created, but cannot modify data sources created by other users (corresponds to the VQL ALTER command).

- **Connection access**: If a user has this privilege on a database, then he/she can connect to same, otherwise he/she cannot. This type of access is useful if, for example, you wish to temporarily revoke user access to a database without having to modify her/his other normal privileges manually.

Denodo Virtual DataPort also supports individual privileges for specific views and stored procedures. The types of access that can be applied to a specific database view and/or stored procedure are:

- **Read access**: If a user has this privilege on a view or stored procedure, he/she can execute the following tasks on it:
  - View information about a base relation, view or stored procedure of the database. For example, access the schema, search methods, cache configuration, swapping configuration, etc., of a base view (corresponds to the VQL DESC command).
  - Execute queries against the view or stored procedure (corresponds to the VQL SELECT and CALL commands).
  - Create new views that use it, wherever creation access is available in the database to which the view belongs. Corresponds to the VQL CREATE VIEW command.
  - If a user does not have read privileges on a database, but does have them for some of its views and/or stored procedures, the list command may be executed, but only said components will be displayed.

- **Write access**: Having write privileges implies that you automatically also have read privileges. If a user has this privilege on a view and/or stored procedure, he/she can execute the following additional tasks on it:
  - Delete the component (corresponds to the VQL DROP command).
  - Modify the component (corresponds to the VQL ALTER command).

- **Insertion access**: This allows inserting tuples in the view through INSERT statements. Not applicable to stored procedures.

- **Update access**: This allows updating tuples in the view through UPDATE statements. Not applicable to stored procedures.
• **Deletion access.** This allows deleting tuples in the view through `DELETE` statements. Not applicable to stored procedures.

### 11.3 VQL STATEMENTS OF DATABASES, USERS AND PRIVILEGES

To manage the databases, users and privileges of a Virtual DataPort server, it is needed to connect to the server using an administrator-type user. It is not needed to specify a database in the connection `Uri`.

When the server is installed, a default administration user is created whose user name is `admin` and whose password is also `admin`.

The following sections respectively describe how to create new databases, how to modify or delete them, how to create new users and, finally, how to modify or delete existing users.

#### 11.3.1 Creating Databases

The VQL `CREATE DATABASE` statement allows an administrator-type user to create a new database in the server, indicating a name for the new database and, optionally, a description of same. Figure 36 shows the syntax of the `CREATE DATABASE` statement. Use of the user privilege assignment options is described in section 11.3.6.

```
CREATE DATABASE <name:identifier> 
[<description:literal>]
[<grant>]*
```

<grant> ::= (see section 11.3.6.1)

*Figure 36* Syntax of the CREATE DATABASE statement

#### 11.3.2 Modifying and Deleting Databases

To view the list of current databases in the server the `LIST` command should be used (see section 13). Each user will see the databases for which they have connection access rights. An administrator user will have access to all databases of the management system.

Once a database is created, an administrator user can modify its description using the `ALTER DATABASE` statement (see Figure 37).
Figure 37 Simplified syntax of the ALTER DATABASE statement

Through this statement, it is possible to modify user access privileges for the database (see section 11.3.6.1) and the default preferences in the database for cache configuration (see section 19.2.2) and the swapping to disk of large queries (see section 19.2.3).

The DESC command (see section 12) allows obtaining data about a database, showing the user access rights for this database. If the user is an administrator, then it will show the access rights of all the users of the indicated database.

An administrator user can also delete a database from the management system using the DROP command (see section 14). Note than when a database is deleted all its components are deleted: data sources, views, base relations, etc.

11.3.3 Creating Users

The CREATE USER statement (see Figure 38) allows creating a new user in the server. As mentioned earlier, two types of users exist. An administrator user can create users of any of the two types.

To create a new user the name and password must be indicated, and an optional description may also be included. The create statement also specifies whether it is a new administrator user (ADMIN modifier) or a normal user. The ENCRYPTED modifier specifies that the provided password is already encrypted and, as such, does not need to be encrypted again (this option is typically used only by the Dataport import / export processes, so users typically do not have to care about it).

Users can be authenticated against DataPort or against an LDAP-type data source registered in DataPort (see section 18.3.10). The second case is specified using the LDAP modifier. In this case, two additional pieces of data must be provided:

- LDAP server [DATASOURCE]. The format is <databaseName>..<dataSourceName>, where <databaseName> specifies the database where the LDAP data source has been registered and <dataSourceName> is the name of the data source.
• LDAP user (USERNAME). This specifies the name of the user in the LDAP server. For example, the value 'cn=test,ou=People,dc=denodo,dc=com' identifies the test user in an organizational unit People for the domain denodo.com.

How to assign privileges to users is described in section 11.3.6.

```
CREATE [ OR REPLACE ] USER [ ADMIN ] <name:identifier>
  <authentication>
  [ <description:literal> ]
  [ <grant> ]*

<authentication> ::= <password:literal> [ ENCRYPTED ]
| LDAP {
  DATASOURCE <databaseName:identifier>.<dataSourceName:identifier>
  USERNAME <name:literal>
}

<grant> ::= (see section 11.3.6.2)
```

Figure 38 Syntax of the CREATE USER statement

**NOTE:** If a LDAP data source is deleted on cascade (see section 14), then the users depending on it will be also deleted. This operation can only be executed by an administrator user.

### 11.3.4 Modifying and Deleting Users

The LIST statement (see section 13) allows obtaining the list of users in the server. Using the DESC command (see section 12), you can obtain data about one user, including her/his access rights to the different databases and views. Administrator users can access all the data of any user. The remaining users can only obtain their own data.

Administrator users can remove users from the server using the DROP statement (see section 14). The predefined "admin" administrator cannot be deleted.

#### 11.3.4.1 Modifying User Data

Any user can change their access code and description using the ALTER USER statement (see Figure 39). In the case of a user being authenticated against an LDAP server, the server data can also be modified (see section 11.3.3). It is also possible to modify the privileges of a user (see section 11.3.6).

```
ALTER USER <name:identifier>
  [ <authentication> ]
  [ <description:literal> ]
  [ <grant> ]*

<authentication> ::= PASSWORD <password:literal>
| LDAP {
  DATASOURCE <databaseName:identifier>.<dataSourceName:identifier>
  [ USERNAME <name:literal> ]
}

<grant> ::= (see section 11.3.6.2)
```

Figure 39 Syntax of the ALTER USER statement
11.3.5  Changing the Active Database

During a session with the Virtual DataPort server a user may wish to connect to a certain database or use a different user to execute certain tasks that require other access rights. To allow this functionality the commands CONNECT and CLOSE can be used (Figure 40).

CONNECT [USER <name:identifier> PASSWORD <password:literal>]
    [DATABASE <name:identifier>]

CLOSE

Figure 40  Syntax of the CONNECT and CLOSE statements

The CONNECT command allows indicating a user name and password to initiate a new session in the server with a new profile. A session may also be initiated with a new database (with the current user or another user).

The CLOSE command allows the previous session to be reestablished after having established a new session with the CONNECT command.

11.3.6  Modifying the Privileges of a User

For users that are not administrators, privileges to different system databases, stored procedures and views can be modified. This task can only be executed by administrator users.

Modifying the privileges of users can be executed on a database level for a series of users or individually by user.

11.3.6.1  Specifying Privileges by Databases

The CREATE DATABASE (Figure 36) and ALTER DATABASE (Figure 37) statements can include the GRANT and REVOKE clauses to grant or revoke access rights to each user in a database (see Figure 41)

The access rights are:

- CONNECT. The user can connect to the database. If a user does not have this access right to a database, the other privileges are ignored.
- CREATE. The user can create new elements in the database.
- READ. The user can access all the views and stored procedures of the database.
- WRITE. The user can modify / delete any view / stored procedures of the database. Write access implies read access.
- ALL PRIVILEGES. The user is granted all the previous privileges: CONNECT, CREATE, READ and WRITE.

There is a special type of access right, ADMIN, which gives the user the following privileges:

- The same privileges as with ALL PRIVILEGES.
- Set the configuration parameters of the database: I18N, cache, swap, etc.
- Edit the description of the database.
- Grant / revoke privileges to normal users (not admin or database-admin users). It cannot grant the ADMIN privilege to other users.

A user with this privilege cannot:

- Create / delete users.
- Change password of users.
- Create / drop databases.
- Grant ADMIN privileges to other users.
<grant> ::= 
   GRANT <database privileges> TO <user:identifier> 
   | REVOKE <database privileges> TO <user:identifier>

<database privileges> ::= 
   ALL PRIVILEGES 
   | ADMIN 
   | <database privilege list>

<database privilege list> ::= <database privilege> [, <database privilege>]*

<database privilege> ::= 
   CONNECT 
   | CREATE 
   | READ 
   | WRITE

Figure 41 Syntax of the GRANT/REVOKE clauses for Databases

11.3.6.2 Specifying privileges by User

User privileges can be assigned, with the statements CREATE USER (Figure 38) (when the user is created) or ALTER USER (Figure 39) (when the user has already been created).

User privileges are managed through the GRANT (assign privileges) and REVOKE (revoke privileges) clauses. Two cases can be distinguished:
- assign user access to databases
- assign user access to database views and stored procedures.

Figure 42 shows the syntax of these clauses for assigning user access to databases on a global level. On the database level it is possible to grant or revoke all access rights (ALL PRIVILEGES) or a list of the following access rights:
  • CONNECT. The user can connect to the database. If the user does not have this privilege on a database, the other privileges are ignored.
  • CREATE. The user can create new elements in the database.
  • READ. The user can access all the views and stored procedures of the database.
  • WRITE. The user can modify / delete any view / stored procedures of the database. Write access implies read access.

There is a special type of access right, ADMIN, which gives the user the following privileges:
  • The same privileges as with ALL PRIVILEGES.
  • Set the configuration parameters of the database: I18N, cache, swap, etc.
  • Edit the description of the database.
  • Grant / revoke privileges to normal users (not admin or database-admin users). It cannot grant the ADMIN privilege to other users.

A user with this privilege cannot:
  • Create / delete users.
  • Change password of users.
  • Create / drop databases.
  • Grant ADMIN privileges to other users.
Figure 42 Syntax of the clauses GRANT/REVOKE for Databases

Figure 43 shows the syntax of these clauses for assigning user access rights to individual views and/or stored procedures. These assignments are considered when a user does not have read access or global write access to all the elements of the database.

In the case of associating user privileges to views of a database, only READ, WRITE, INSERT, UPDATE and DELETE access rights are applicable.
<grant>::=
   GRANT <view privileges> ON <database:identifier>.<view:identifier>
   | GRANT <procedure privileges> ON PROCEDURE <database:identifier>.<procedure:identifier>
   | REVOKE <view privileges> ON <database:identifier>.<view:identifier>
   | REVOKE <procedure privileges> ON PROCEDURE <database:identifier>.<procedure:identifier>

<view privileges>::=
   ALL PRIVILEGES
   | <view privilege list>

<view privilege list>::=<view privilege>[, <view privilege>]*

<view privilege>::=
   READ
   | WRITE
   | INSERT
   | UPDATE
   | DELETE

<procedure privileges>::=
   ALL PRIVILEGES
   | <procedure privilege list>

<procedure privilege list>::=<procedure privilege>[, <procedure privilege>]*

<procedure privilege>::=
   READ
   | WRITE
   | INSERT
   | UPDATE
   | DELETE

---

**Figure 43** Syntax of the clauses GRANT/REVOKE for views

Figure 44 shows an example in which two databases are created, “database1” and “database2”. One user called “user1” is also created. The new user is assigned the following privileges on “database1” and “database2”:
- has full privileges to “database1”
- has connection and creation access to “database2”. It only has read/write access to “view1”.

CREATE DATABASE database1 'Database1 Description';
CREATE DATABASE database2 'Database2 Description';
CREATE USER user1 'user1password' 'User1 Description'
   GRANT ALL PRIVILEGES ON database1
   GRANT CONNECT, CREATE ON database2
   GRANT READ,WRITE ON database2.view1;

**Figure 44** Example of assigning privileges to users
12 DESCRIBING CATALOG ELEMENTS

The DESC statement allows obtaining a description of the elements in the Virtual DataPort server. Its syntax is given in Figure 45.

```
DESC { DATABASE | USER | TYPE | PROCEDURE | VIEW [ TREE ] } <name:identifier>  
[ ( <conversionProperties> [, <conversionProperties>]* ) ]

DESC DATASOURCE <datasource type> <name:identifier>

DESC MAP { I18N | SIMPLE } <name:identifier>

DESC OPERATOR <name:operator> <type:identifier>

DESC PROCEDURE AS VIEW <name:identifier>  
( [ <procedureParameter> [, <procedureParameter>]* ] )

DESC WRAPPER <wrapper type> <name:identifier>

DESC WRAPPER ITP <name:identifier>  
[ <itpConversionProperties> ]

DESC WEBSERVICE <name:identifier>

DESC WIDGET <name:identifier>

DESC QUERYPLAN <query>

DESC SESSION

DESC VQL { PROCEDURE | TYPE } <name:identifier>  
( [ <descProperties> [, <descProperties>]* ] )

DESC VQL VIEW <name:identifier>  
( [ <conversionProperties> [, <conversionProperties>]* ] )

DESC VQL DATASOURCE <datasource type> <name:identifier>  
( [ <conversionProperties> [, <conversionProperties>]* ] )

DESC VQL WRAPPER <wrapper type> <name:identifier>  
( [ <descProperties> [, <descProperties>]* ] )

DESC VQL WRAPPER ITP <name:identifier>  
[ <itpConversionProperties> ]

DESC VQL MAP { I18N | SIMPLE } <name:identifier>  
( [ <descProperties> [, <descProperties>]* ] )

DESC VQL WEBSERVICE <name:identifier>  
( [ <descProperties> [, <descProperties>]* ] )

DESC VQL WIDGET <name:identifier>  
( [ <descProperties> [, <descProperties>]* ] )

DESC VQL DATABASE [ <name:identifier> ]  
( [ <conversionProperties> [, <conversionProperties>]* ] )

<datasource type> ::=  
{ ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | OLAP | SAPBW | SAPERP | WS | XML }

<wrapper type> ::= <datasource type> | ITP
```
Describing Catalog Elements

The first group of statements allows describing different catalog elements:

- The first statement obtains a description of a database, a user, a data type, a stored procedure or a view. If a view is described, the modifier TREE can be indicated. This will return a representation of the view tree: that is, the group of views on which the view is defined together with the relational algebra operators that combine them.
- DESC DATASOURCE returns the information about a data source defined in the catalog.
- DESC MAP returns the content of a map. We have to indicate the type of the map we want to describe: simple or i18n.
- DESC OPERATOR returns the description of an operator for a specific data type.
- The DESC PROCEDURE AS VIEW statement describes a stored procedure, treating it like a view. This is useful because DataPort stored procedures can appear in the FROM clause of a query or view (see section 9).
- DESC WRAPPER returns information about a wrapper defined in the catalog.
- DESC WEBSERVICE returns information about a Denodo Web service defined in the catalog (see section 15).
- DESC WIDGET returns information about a widget defined in the catalog.

The DESC QUERYPLAN statement provides a look ahead at the execution plan that DataPort is going to use to execute a query. Although it will be possible to access detailed trace information, including the execution plan used, after executing the query (see section 5.10), the QUERYPLAN provides this information without having to run the query.

DESC SESSION returns the name of the database that the user is connected to, along with her login name.

The DESC VQL statements return the statement required to create the element. If the described element is a view, the statement also returns the statements needed to create the elements it depends on. E.g. DESC VQL VIEW V will return the statement required to create the view V and the statements needed to create the data types, wrappers, data sources and other views required to define the view V completely. If we are executing DESC VQL VIEW but we do not need the VQL statements of the elements that this view depends on, use the option includeDependencies.

E.g. DESC VQL VIEW V ('includeDependencies'='no') returns the sentence CREATE VIEW V ..., but not the sentences to create the views that V depends on and its data sources.

Section 12.1 explains the meaning of the other <conversionProperties>.
12.1 EXPORTING METADATA

The DESC VQL DATABASE sentence allows exporting all the metadata from a certain Virtual DataPort database or from all the databases of the server. This is very useful for backup and migration purposes.

The syntax of the command is:

```
DESC VQL DATABASE [ <database_name> ]
    ('<property name>'='{yes|no}'
     [, '<property name>'='{yes|no}' ]*)
```

If the DESC VQL DATABASE sentence includes the parameter <database_name>, all the metadata from that database will be exported. It will not include:
- User definitions and privileges.
- The CREATE DATABASE sentence required for creating the database

If the DESC VQL DATABASE sentence does not include the parameter <database_name>, all the Server metadata will be exported. That is:
- The metadata from all the database, along with their CREATE DATABASE statement.
- User definitions and privileges.
- Server settings
- …

The user that executes this statement needs administrator privileges.

The configuration parameters (<property name>) of this command are:

- `includejars`. If yes, the output will include the jars that contain the JAVA classes associated with extensions (see section 19.3).

- `includeEnvSpecificElements` and `includeNonEnvSpecificElements`. In many organizations, installing a new application requires its validation in different environments (e.g. development, pre-production and production environments). Certain elements such as the paths or the authentication information used to access the data sources will normally be different in each environment. Usually, data sources are configured in each environment with the same name, but access the data source version available in each environment. Under these circumstances, it may be useful to export separately the elements that typically change from one environment to another and those that do not.

For example, if users have created a new group of views in an environment and wish to pass them to another environment, they can export only the independent elements of the environment with these options:

```
('includeNonEnvSpecificElements'='yes',
 'includeEnvSpecificElements'='no')
```

The catalog elements considered dependent on the environment are data sources (see section 18), users and databases (see section 11), and server configuration properties (ports, etc.).

- `replaceExistingElements`. If 'no', the result will return a CREATE OR REPLACE… for each element, instead of just CREATE…

- `dropElements`. If no, the result will not include a DROP… sentence before each CREATE… one.

- `includescanners`. If yes, creation statements of the WWW wrappers will contain the binary files of the ITPilot scanners used by these wrappers (see the ITPilot documentation [ITPILOT] for further information on the scanners and ITPilot wrappers in general).
- **includeCustomComponents.** If yes, the output file will include the ITPilot custom components used by the existing WWW data sources.

- **dropElements.** If yes, the output file will include a command **DROP … CASCADE** before each command **CREATE**...

  E.g.  
  DROP DATASOURCE JDBC IF EXISTS internet_ds CASCADE;  
  CREATE DATASOURCE JDBC internet_ds...

  The **DROP … CASCADE** sentence deletes the data source **internet_ds** and all the views that depend on it. This option is useful when we want to make sure that the imported data sources do not have more views that the ones contained in the VQL file.

- **replaceExistingElements.** If yes, the output file will not include **DROP** statements. Just **CREATE OR REPLACE**.

  E.g.  
  CREATE OR REPLACE DATASOURCE JDBC internet_ds...

  In this case, the Server that imports this file, will replace the JDBC data source **internet_ds**. But, as it does not include the sentence **DROP … CASCADE**, it will not delete the views that depend on this data source.

For metadata backup and migration purposes, there also exists the **DESC VQL WRAPPER ITP** statement. This statement exports only the metadata related to WWW-type wrappers of the active database. It is useful for ITPilot server backups or for migrations of ITPilot wrappers from a Virtual DataPort installation to an ITPilot installation. Please see the ITPilot documentation [ITPILOT] for more information about this type of wrappers. To use this statement is necessary to be connected to the database which WWW wrappers are to be exported before executing the statements.

The **DESC VQL WRAPPER ITP** statement also allows to specify a value for the ‘**INCLUDESCANNERS**’ property.

**Example:** the following statement exports a database named DB. The WWW wrappers container in DB will be exported, including the binary files required to regenerate the scanners that make use of them.

```
DESC VQL DATABASE DB ('INCLUDESCANNERS'='YES')
```

### 12.2 IMPORTING METADATA

To import the exported metadata, you need to execute the VQL file obtained during the export process. This can be done by using the Virtual DataPort administration tool or by using the import script included in the utilities for importing and exporting metadata (see the sections 'Exporting / Importing the Server Metadata’ and ‘Use of the Import/Export Scripts for Backup and/or Replication’ of the Administration Guide [ADMIN_GUIDE]).

It is strongly recommended to switch to **single user** mode before importing metadata. When this mode is established from a given connection, only the sentences executed from that connection will be executed by Virtual DataPort. The sentences emitted from other active connections will be queued until the server exits the **single user** mode. Only users of administrator type can switch to **single user** mode.

To activate the single user mode enter the following command:

```
ENTER SINGLE USER MODE
```
When this sentence is executed, DataPort will wait for the active sequences to finish before returning control. Once the server has entered the \emph{single user mode}, DataPort will only execute the statements emitted from the connection that switched to \emph{single user}. The sentences emitted from other active connections will be queued until the server exits the \emph{single user} mode. Notice that the time spent by a statement in the queue counts with respect to execution timeouts.

To exit the single user mode, enter the following command:

\begin{verbatim}
EXIT SINGLE USER MODE
\end{verbatim}

If the connection that established the \emph{single user} mode is closed, DataPort automatically exits this mode.
13 LISTING ELEMENTS IN THE CATALOG

The **LIST** statement allows listing the different types of elements in the server’s catalog:

- The first allows the list of all the databases, users or internationalization configurations to be requested.
- **LIST DATASOURCES** lists the data sources of the type specified (see section 18.2).
- **LIST FUNCTIONS CUSTOM** lists the functions uploaded by the user (see section 19.3.1)
- **LIST JARS** lists the loaded extensions (see section 19.3).
- **LIST LISTENERS JMS** lists the listeners that receive queries from JMS servers (see section 10.4)
- **LIST MAPS** lists maps of the type simple or i18n.
- **LIST OPERATORS** allows listing the operators that are applicable on a given data type.
- **LIST PROCEDURES** allows listing the stored procedures.
- **LIST TYPES** shows all data types from the catalog or only those of a certain type (enumerated, array or register).
- **LIST VIEWS** allows listing all the base relations or all the views.
- **LIST WEBSERVICES** lists the published Web services (see section 15).
- **LIST WIDGETS** lists the published widgets (see section 16).
- **LIST WRAPPERS** lists wrappers of the specified type (see section 18).

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

LIST WIDGETS

LIST WRAPPERS <wrapper type>

<datasource type> ::= 
   { ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | SAPERP | 
     SAPBW | WS | XML }

<wrapper type> ::= <datasource type> | ITP

Figure 46 Syntax of the statement LIST

For example, to list the existing databases the following statement is executed:
   LIST DATABASES;

To list the maps of the type i18n the following statement is used:
   LIST MAPS I18N;

And to list the operators that operate on the data type int, use the statement:
   LIST OPERATORS int;
14 REMOVING ELEMENTS FROM THE CATALOG

The **DROP** statement allows removing specific elements from the server. Its syntax is shown in Figure 47.

```sql
DROP { DATABASE | USER } [ IF EXISTS ] <name:identifier>
DROP TYPE [ IF EXISTS ] <name:identifier> [ CASCADE ]
DROP { VIEW | TABLE } [ IF EXISTS ] { <name:identifier> | <name:literal> } [ CASCADE ]
DROP WRAPPER <wrapper type> [ IF EXISTS ] <name:identifier> [ CASCADE ]
DROP DATASOURCE <datasource type> [ IF EXISTS ] <name:identifier> [ CASCADE ]
DROP MAP { I18N | SIMPLE } [ IF EXISTS ] <name:identifier>
DROP PROCEDURE [ IF EXISTS ] <name:identifier>
DROP JAR [ IF EXISTS ] <name:literal>
DROP LISTENER JMS [ IF EXISTS ] <name:literal>
DROP SCANNER <name:literal>
DROP WEBSERVICE [ IF EXISTS ] <name:identifier>
DROP WIDGET [ IF EXISTS ] <name:identifier>

<datasource type> ::= 
    { ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | SAPBW | SAPERP | WS | XML }

<wrapper type> ::= <datasource type> | ITP
```

**Figure 47** Syntax of the statement **DROP**

The available options for the **DROP** statement are:
- Remove database or a user from the server.
- Remove a data type.
- Remove a view (base or derived).
- Remove a specific wrapper (see section 18) or data source (see section 18.3), indicating its type and name.
- Remove a specific data dictionary map indicating its type (**i18n** or **simple**) and its name.
- Remove a stored procedure.
- Remove a **.jar** (see section 19.3).
- Remove an ITPilot scanner used by a WWW wrapper (see ITPilot User Guide [ITPILOT]).
- Remove a published Web service (see section 15).
- Remove a published Widget (see section 16)
- Remove a JMS listener (see section 10.4)
The `IFEXISTS` modifier can be included in all of the above cases. In this case, the DROP sentence will only be run in the event of the specified element existing.

The statements for deleting views, types, wrappers and data sources allow the optional modifier `CASCADE`. If this modifier is not indicated, when an attempt is made to delete one of these elements, an error will occur if another catalog element depends on it (for example, if a data source is deleted and a wrapper that uses it exists). In this case, the element will not be deleted. If the modifier `CASCADE` is specified, then the indicated element will be deleted and all the elements that depended on it will be also deleted. If the user executing the delete operation has not enough privileges over all the involved elements, the operation will fail.

Some examples of use of the `DROP` statements are shown below. To eliminate the view `shopview` the following statement is executed:

```
DROP VIEW shopview;
```

To remove the WWW wrapper `shopview` simply execute:

```
DROP WRAPPER ITP shopview;
```

And to remove the map type `i18n es_euro` the following statement is used:

```
DROP MAP I18N es_euro;
```
15 PUBLICATION OF WEB SERVICES

Virtual DataPort allows publishing one or several views (and/or stored procedures) as a Web Service to enable use by any external application. The Web Services published can be deployed to the Web Service container embedded in the Denodo Platform. It is also possible to automatically generate a .war file to deploy it in an external JEE applications server.

The Web Services can be published in the following versions:

- SOAP-based Web Services.
- REST-style Web Services that use HTTP directly as the transport protocol and return data coded in XML.
- JSON Web Services. Similar to the REST-style Web services, although the output will be JSON (see [JSON]).
- RSS Web Services. Similar to the REST-style Web services, although the output will be produced in the RSS format.
- HTML Web Services. Similar to the REST-style Web services, but the output consists of an HTML table containing the response data for the query run. The table includes JavaScript code to sort the results by any field and/or paginate the returned results. It is also possible to adjust the size of the table and the cells and to modify its graphic layout using a CSS file.

It is possible to use different authentication methods to validate users’ identity (see section 15.2)

This section describes how to create and deploy these Web services using VQL.

NOTE: It is strongly recommended that the Web service publication process be undertaken graphically using the DataPort administration tool (see [ADMIN_GUIDE]). That way, all the necessary VQL statements will be generated automatically.

15.1 CREATION OF NEW WEB SERVICES

Figure 48 shows the syntax for creating a new Web service that publishes a view or a stored procedure.

```
CREATE [ OR REPLACE ] WEBSERVICE <name:identifier>
    CHUNKSIZE = <integer>
    CHUNKTIMEOUT = <integer>
    QUERYTIMEOUT = <integer>
    POOLENABLED = <boolean>
    POOLINITSIZE = <integer>
    POOLMAXACTIVE = <integer>
    I18N <name:identifier>
    [ DATETYPEMAPPING { DATE | DATETIME } ]
    [ AUTHENTICATION ( HTTP <http_authentication> SOAP <soap_authentication> ) ]
```
OUTPUT TYPE ( [ 
  REST ( <rest_xslt_config> [, <rest_xslt_config> ]* ) ] 
  | JSON 
  | SOAP ( STYLE { DOCUMENT | RPC } 
    [ XSLT ( <soap_xslt_config> [, <soap_xslt_config> ]* ) ] 
    [ JMS ( 
      VENDOR { ACTIVEMQ | WEBSHOPERMQ | JNDI } 
      DESTINATION = <name:literal> 
      { QUEUE | TOPIC } 
      [ USER = <name:literal> 
        PASSWORD = <name:literal> [ ENCRYPTED ] 
      ] 
      PROPERTIES ( <properties> [, <properties> ]* ) 
    ) 
    ) 
  | HTML [ ( CSS = <css:literal> ) ] 
  | RSS ( [ <rssmapping>* ] ) 
]+ ) 
[ <operation> ]* 

<http_authentication> ::= { 
  NONE 
  | BASIC <credentials> 
  | BASIC LDAP <ldap_credentials> 
  | BASIC VDP [ VDPACCEPTEADUSERS <users_list> ] 
  | DIGEST <credentials> 
} 

<soap_authentication> ::= { 
  NONE 
  | BASIC <credentials> 
  | BASIC LDAP <ldap_credentials> 
  | BASIC VDP [ VDPACCEPTEADUSERS <users_list:users_list> ] 
  | DIGEST <credentials> 
  | WSS BASIC <credentials> 
  | WSS BASIC LDAP <ldap_credentials> 
  | WSS BASIC VDP [ VDPACCEPTEADUSERS <users_list:users_list> ] 
  | WSS DIGEST <credentials> 
} 

<credentials> ::= 
  USER <user_name:literal> PASSWORD <password:literal> [ ENCRYPTED ] 

<ldap_credentials> ::= 
  LDAPDATASOURCIE <server_uri:literal> 
  LDAPUSERPATTERN <user_pattern:literal> 
  [ LDAPACCEPTEADUSERS <users_list> ] 

<users_list> ::= <user_name:literal> [, <user_name:literal> ]* 

<operation> ::= OPERATION <name:literal> ( 
  TYPE { SELECT | INSERT | UPDATE | DELETE } 
  SCHEMA { 
  ( VIEW | WRAPPER <wrapper_type> | STOREDPROCEDURE ) 
}
Below is a brief description of the use of the statement. A Web service published by Virtual DataPort is formed by a list of operations defined using the \textit{OPERATION} clause. Each operation will run a VQL statement that is indicated in the \textit{VQL} property of the operation. The operation may act on a view, a wrapper, a stored procedure or execute a
specific VQL statement (SCHEMA property). The type of the statement (TYPE property) can be: select (most common), insert, update or delete. Each operation contains a list of input parameters and one output parameter. In the case of query operations, the output parameter will be an array of registers containing the results of the query run. Insert / update and delete operations return the number of tuples affected by the operation.

The versions in which the service is to be published are indicated in the OUTPUT_TYPE clause. The I18N parameter allows specifying the internationalization configuration used by the service.

The parameters URI, LOGIN, PASSWORD, CHUNKSIZE, CHUNKTIMEOUT, QUERYTIMEOUT, POOLENABLED, POOLINITSIZE and POOLMAXACTIVE allow configuring different aspects of the connections that the Web service will use to run the statements in the DataPort server:

- **URI, LOGIN and PASSWORD.** These parameters are only used when the Web service is deployed in an external web service container. They indicate the URI, user ID and password to be used by the Web service to access the DataPort server. The ENCRYPTED modifier indicates that the password provided is encrypted.

- **CHUNKSIZE, CHUNKTIMEOUT, QUERYTIMEOUT.** Their interpretation is the same as in any other VDP client (see VDP Developer’s Guide [DEVELOPER_GUIDE]).

- **POOLENABLED.** This must be set to the true value to enable the connection pool (highly recommended).

- **POOLINITSIZE.** Initial number of connections to be opened in the pool.

- **POOLMAXACTIVE.** Maximum number of connections in the pool. If this is a negative value, then the number is not limited.

The RSS format specifies a series of specific fields for each item. Therefore, on exporting a view in RSS format, the correspondence between the fields of the view and the fields in RSS format must be specified. This is possible through the MAPPING clause. An RSS feed contains a channel element that specifies general information on the feed. The CHANNEL parameter of the MAPPING clause allows specifying constant values for each of the channel subelements permitted by the RSS format. An RSS feed contains a list of item elements. DataPort will generate an item element for each tuple returned by the query made on the view or stored procedure used in the service. The ITEM parameter of the MAPPING clause allows selecting the attribute of the view corresponding to each item subelement defined in RSS format. The RSS format specifies that at least one value must be assigned either to the title subelement or to the description subelement.

The Web services published by Virtual DataPort can subscribe to a JMS server to listen to SOAP messages (SOAP over JMS [SOAP_JMS]). To do that, add the parameter JMS and its appropriate parameters. Section 10.4 explains the meaning of the parameters related to establishing connections with JMS servers.

In an environment with existing SOAP/REST clients and services, we do not need to modify those clients to work with Virtual DataPort Web services. We can define XSLT stylesheets [XSLT] to:

- Transform the incoming SOAP messages to adapt them to the format that the Denodo Web service expects.
- Transform the SOAP responses before sending them to the existing clients.
- Transform the REST (XML) responses before sending them to the existing clients.

To do this, use the parameter XSLT inside the parameters REST or SOAP.

For more information about this, read sections ‘SOAP XSLT Transformations’ and ‘REST XSLT Transformations’ of the Administration Guide [ADMIN_GUIDE].
### 15.2 WEB SERVICES AUTHENTICATION

It’s possible to protect the access to a Web service by using different authentication methods. The available authentication methods depend on the Web service type:

<table>
<thead>
<tr>
<th>Authentication method</th>
<th>SOAP</th>
<th>HTML / REST / JSON / RSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Basic</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HTTP Basic / LDAP / VDP</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HTTP Digest</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WSS Basic</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WSS Basic / LDAP / VDP</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WSS Digest</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Available authentication methods for the Denodo Web services

With the Virtual DataPort authentication (options **BASIC VDP** and **WWW VDP**) the clients of the Web service have to use the credentials of the Virtual DataPort users. Besides, the Web service will connect to Virtual DataPort with the credentials provided by the client.

The parameter **VDPACCEPTEDUSERS** is a comma-separated list of user names. Only users, whose user name is in that list, will have access to the Service. If this parameter is missing, the Service will accept all Virtual DataPort’s users.

Unlike with the other authentication methods, with this one, we have to grant the user privileges to access the published views.

#### 15.2.1 Basic and Digest

Basic and Digest use the Basic and Digest HTTP Access Authentication methods [HTTP_AUTH].

In HTTP Basic the credentials are passed as plaintext and in HTTP Digest they are sent encrypted.

All the users will use the same credentials indicated in the parameters **USER** and **PASSWORD**.

The **ENCRYPTED** modifier indicates that the password provided is encrypted (this option is typically only used by the server export/import metadata processes. Users do not need to use this option).

#### 15.2.2 Basic LDAP

In the Basic LDAP authentication the credentials are passed as plain text and validated using a LDAP server.

Unlike Basic authentication, which forces every user to use the same user name and password, by using a LDAP server, every user has its own user name and password.

The following parameters are required to configure this authentication method:

- **LDAPDATASOURCE**: URI of the LDAP server used to validate the users’ credentials.

- **LDAPUSERPATTERN**: Pattern used to build the user’s Distinguished Name replacing the @login token with the received user name. 
  
  I.e. if User pattern is `cn=@login,ou=People,dc=YourOrganization,dc=com`, the @login token is replaced by the user name provided by the invoker of the Service.
15.2.3 WSS

WSS [WSS] enforces integrity and confidentiality on Web Services messaging. It works on top of the Basic or Digest authentication methods. Currently, Virtual DataPort supports the authentication profile called "Username Token".

15.2.4 VDP

When using the authentication methods HTTP BASIC VDP, SOAP BASIC VDP or SOAP WSS BASIC VDP the Web Service will connect to Virtual DataPort with the credentials used by the client of the Web Service. Only users whose user name is in the VDPACCEPTEDUSERS list will have access to the Service. If the list is empty, all Virtual DataPort's users will be accepted. With this authentication method, the users also need to have permission to access the published views.

15.3 EMBEDDED WEB CONTAINER MANAGEMENT

The VQL WEBCONTAINER statement allows managing for the Web container integrated into the Denodo Platform. Figure 49 shows its syntax.

```
WEBCONTAINER {
    START | STOP | STATUS | SET <propertylist>  }
```

Figure 49 Syntax of the WEBCONTAINER statement

The statement allows the following options:

- **START / STOP / STATUS**. This allows starting, stopping and checking the current status of the Web container.
- **SET**. This allows specifying the port numbers used by the embedded Web container.

The command `DESC WEBSERVICE <web service name:literal>` provides a more detailed description of a web service: operations, fields of every operation, etc.

15.4 DEPLOYMENT AND EXPORTING OF WEB SERVICES

The statements `DEPLOY WEBSERVICE`, `REDEPLOY WEBSERVICE` and `UNDEPLOY WEBSERVICE` deploy, redeploy and undeploy the specified Web service. The web service must have been previously created using the `CREATE WEBSERVICE` statement.

If the Web service is to be deployed in an external J2EE Web container instead of using the embedded container, the `EXPORT WAR FROM WEBSERVICE` statement generates a `.war` file containing the Web service specified in the `WEBSERVICE` parameter. The file name will be as specified in the `NAME` parameter.

The `EXPORT WSDL FROM WEBSERVICE` statement generates a `.wsdl` file, specifying the SOAP version interface of the Web service specified in the `WEBSERVICE` parameter. The file name will be as specified in the
NAME parameter. It can be used with a utility for SOAP Web Service programming to generate the necessary stubs to implement a client program accessing the SOAP Web service.

Both exported files will be accessible for downloading in the /export path of the Web container embedded in the Platform (e.g. if the default path is used and accessed from the machine where DataPort is installed, they can be accessed through http://localhost:9090/export).

Figure 50 shows the syntax of these statements:

```
DEPLOY WEBSERVICE <name:identifier>
  LOGIN = <literal>
  PASSWORD = <literal> [ ENCRYPTED ]

REDEPLOY WEBSERVICE <name:identifier>
  LOGIN = <literal>
  PASSWORD = <literal> [ ENCRYPTED ]

UNDEPLOY [IF EXISTS] WEBSERVICE <name:identifier>

EXPORT WAR FROM WEBSERVICE <name:identifier>
  NAME = <name:literal>
  URI = <literal>
  LOGIN = <literal>
  PASSWORD = <literal> [ ENCRYPTED ]

EXPORT WSDL FROM WEBSERVICE <name:identifier>
  NAME = <name:literal>
```

Figure 50 Syntax of the DEPLOY, EXPORT WAR and EXPORT WSDL statement
16 PUBLICATION OF WIDGETS

Virtual DataPort can publish a view or a stored procedure as a widget. Once a widget is deployed, it allows querying the content of the desired element. It contains both a query form and a table to visualize the obtained results. The widgets are compliant with the most common widget inter-operability standards; therefore, they can collaborate with other independently developed widgets (see section Publication of Views as Widgets in [ADMIN_GUIDE] for detail).

The widgets can be exported to different widget technologies:

- Java Portal Servers (Portlets JSR-168 and JSR-286) [PORTLET_STANDARDS]
- Microsoft SharePoint [MOSS]
- OpenAjax Mashup Editor [OPENAJAX]

**NOTE:** It is strongly recommended that the Widget publication process be undertaken graphically using the DataPort administration tool (see [ADMIN_GUIDE]). That way, all the necessary VQL statements will be generated automatically.

16.1 CREATE NEW WIDGETS

Figure 51 shows the syntax for creating a new widget.

```
CREATE [ OR REPLACE ] WIDGET <name:identifier>
  [ DISPLAYNAME = <literal> ]
  [ DESCRIPTION = <literal> ]
  ELEMENTTOPUBLISH = <identifier>
  ELEMENTTOPUBLISHTYPE = { VIEW | STOREDPROCEDURE }
  CHUNKSIZE = <integer>
  CHUNKTIMEOUT = <integer>
  QUERYTIMEOUT = <integer>
  POOLENABLED = <boolean>
  POOLINITSIZE = <integer>
  POOLMAXACTIVE = <integer>
  I18N = <identifier>
  HELPMODEENABLED = <boolean>
  [ CUSTOMIZEDHELPMODECONTENTS = <html_fragment:literal> ]
  [ OPTIONS
    ( PORTLETJSR286 ( PUBLISHCUSTOMTABLEEVENT = <boolean> )
    )
  ]
```

**Figure 51** Syntax of the CREATE WIDGET statement

Below is a brief description of some of the parameters of this statement:

- ELEMENTTOPUBLISH: Name of the view, base view or stored procedure to publish.
- **HELPMODEENABLED**: Enables the Help Mode for the widget. In the three widget platforms, a widget might have a help mode used to display information about the widget. If this parameter is true, but the `CUSTOMIZEDHELPMODECONTENTS` parameter is not present, the widget’s help mode will display a text with instructions on how to use the widget.

- **CUSTOMIZEDHELPMODECONTENTS**: HTML fragment that will be shown when the user opens the widget’s Help Mode. This parameter is only useful if `HELPMODEENABLED` is true.

- **CHUNKSIZE, CHUNKTIMEOUT, QUERYTIMEOUT**: Their interpretation is the same as in any other VDP client (see VDP Developer’s Guide [DEVELOPER_GUIDE]).

- **POOLENABLED**: If true, the connection pool will be enabled (highly recommended).

- **POOLINITSIZE**: Initial number of connections to be opened in the pool.

- **POOLMAXACTIVE**: Maximum number of connections in the pool. If this is a negative value, then the number is not limited.

- **PUBLISHCUSTOMTABLEEVENT**: Enables the option that the widget exported to a JSR-286 Portlet [PORTLET_STANDARDS] can send complex objects to other portlets. These complex objects contain the whole result of the executed query view/stored procedure obtained from Virtual DataPort (see section ‘Export to JSR-168 or JSR-286 Portlet’ of the [ADMIN_GUIDE]).

  **Note**: Unless needed, this property should be set to false for performance reasons.

### 16.2 EXPORT A WIDGET

This command exports an existing widget to a particular widget technology. Depending on the target, the output will be different:

- **Portlets JSR 168 and 286**: Output is a .war file that can be deployed in any standard Portlet server.

- **OpenAjax Widget**: Output is a .zip file containing the OpenAjax metadata file and a few resource files.

- **MS Web Part**: Output is a .zip containing a .webpart file for the deployment in Microsoft Office SharePoint and an XML file containing information about the exported element.

**Important**: In order to use the OpenAjax Widget and the MS Web Part, their auxiliary Web Services must be deployed. See section 16.3.
Figure 52  EXPORT WIDGET syntax

- **URI**: Its meaning depends on the target widget:
  - Portlet: the URI of the Virtual DataPort server that the portlet will connect to in order to retrieve the data.
  - OpenAjax and MS Web Part: the URI of the auxiliary Web Service that they will connect to in order to obtain the data.
- **LOGIN** and **PASSWORD**: Credentials to connect to Virtual DataPort.
- **DEPLOYMENTURI**: The URI of the server where the contents of the generated file will be located. When exporting an OpenAjax widget, the target .zip file must be decompressed in a location in the same server where the OpenAjax Mashup Editor is deployed.

### 16.3 DEPLOYMENT AND EXPORT OF AUXILIARY WEB SERVICES

The auxiliary web service is a special web service used by the OpenAjax widgets and MS Web Parts to obtain the data from the views or stored procedures of the Virtual DataPort server. This Web Service can be deployed into the web container embedded in the Denodo Platform; or it can also be exported to a .war file and deployed into another J2EE Application Server. Remember that these web services have to be deployed before using the widgets that need them.

The following figures show the syntax of the commands available to manage these web services.

Figure 53  Syntax of the DEPLOY, UNDEPLOY and EXPORT statements
In this command the parameter URI is URL of the Virtual DataPort server that the web service will retrieve the data from.

The following command displays a list of the auxiliary web services deployed in the embedded container:

WEBCONTAINER WIDGETS STATUS

See section 15.3 for more information about the embedded Web container.
17 HELP COMMAND

Virtual DataPort includes a help statement, HELP, which gives the user a detailed description of the syntax of all the existing commands. The syntax of the HELP command is shown in Figure 56.

If it is not assigned any parameters, the HELP statement presents its own syntax. Optionally, it receives as a parameter the command name for which help is required. For example, the statement of Figure 57 allows the user to know in detail the syntax of the command ALTER TABLE. Detailed information on the general VQL syntax can be obtained using the HELP HELP statement.

```
HELP <topic>

<topic> ::= 
    ALTER DATABASE 
    | ALTER DATASOURCE <datasource type> 
    | ALTER PROCEDURE 
    | ALTER TABLE 
    | ALTER USER 
    | ALTER WRAPPER <wrapper type> 
    | BEGIN 
    | CALL 
    | CHOWN 
    | CLEAR CACHE 
    | CLOSE 
    | COMMIT 
    | CONNECT 
    | CREATE DATABASE 
    | CREATE DATASOURCE <datasource type> 
    | CREATE JAR 
    | CREATE LISTENER JMS 
    | CREATE MAP 
    | CREATE PROCEDURE 
    | CREATE SCANNER 
    | CREATE TABLE 
    | CREATE TYPE 
    | CREATE USER 
    | CREATE VIEW 
    | CREATE WRAPPER <wrapper type> 
    | CREATE WEBSERVICE 
    | CREATE WIDGET 
    | DELETE 
    | DEPLOY WEBSERVICE 
    | DSSC 
    | DROP 
    | EXPORT WAR 
    | EXPORT WSDL 
    | HELP 
    | INSERT 
    | LIST 
    | QUERY WRAPPER <wrapper type> 
    | REDEPLOY WEBSERVICE 
```
| ROLLBACK |
| SELECT |
| SET |
| SET TRANSACTIONAL MODE |
| SHOW TRANSACTIONAL MODE |
| SINGLE USER MODE |
| UNDEPLOY WEBSERVICE |
| UPDATE |
| WEBCONTAINER |

<datasource type> ::= 
( ARN | CUSTOM | DF | GS | JDBC | JSON | LDAP | ODBC | SAPBW | SAPERP | WS | XML )

<wrapper type> ::= <datasource type> | ITP

**Figure 56** Syntax of the statement HELP

HELP ALTER TABLE

**Figure 57** Syntax to request help on the command ALTER TABLE
Generators Wrappers and Data Sources

Wrappers are components responsible for offering the server overall common interface for accessing the data sources. Each search method in a base relation has an associated wrapper, which is in charge of receiving the queries issued to the base relation, transforming them into queries to the data source and obtaining the results, returning them to the logical layer of Virtual DataPort in accordance with the format specified by the base relation. Wrappers make the peculiarities of obtaining data from the sources transparent for the server.

Virtual DataPort includes the following predefined types of wrappers:

- **WWW**: This is used to incorporate wrappers for semi-structured sources created using Denodo ITPilot into the system [ITPILOT]. These sources can be accessed from the local file system, via Web, or via FTP. HTML websites are the most important type of sources this wrapper is used for, although it can also be used for other semi-structured sources such as PDF, Word and Excel files (see Denodo ITPilot documentation [ITPILOT]).

- **JDBC**: Extract data from a remote database via JDBC.

- **ODBC**: Extract data from a remote database via ODBC.

- **Multidimensional Databases**: Extract data from multidimensional databases such as SAP BW and SAP BI.

- **Web Services**: Extract data invoking operations defined by Web services.

- **XML**: Extract data from XML files, optionally following a specific DTD or schema. These sources can be accessed via web, local file system or FTP.

- **JSON**: Extract data from JSON files. These sources can be accessed via web, local file system or FTP.

- **DF**: Extract data from flat text files that represent tuples using a regular format, such as using specific characters as tuple and field delimiters or disposing the tuple fields according to a regular expression. Amongst the files supported are files in CSV format. These sources can be accessed via web, local file system or FTP.

- **ARACNE**: They provide access to indexes on non-structured data created using Denodo Aracne [ARCN].

- **GOOGLE MINI**: They provide access to indexes on non-structured data created using the search tool Google Mini [GMINI].

- **LDAP**: They extract data from LDAP servers such as Microsoft Windows Server Active Directory [MS_AD].

- **BAPI**: They invoke SAP BAPIs (Business Application Programming Interfaces) to extract data stored in SAP ERP and other SAP applications.
CUSTOM: Extract data from a source through a specific Java implementation. This type of wrapper allows ad hoc construction of a wrapper program for a specific type of source.

There are corresponding data sources elements for all wrappers, except WWW-type ones, to encapsulate certain data on data source access and configuration.

This section describes how to create and modify wrappers (and their data sources) of any type in Virtual DataPort by using VQL.

**NOTE:** It is strongly recommended that the wrapper and data sources creation process be undertaken graphically using the DataPort administration tool (see [ADMIN_GUIDE]).

The remainder of this section is structured as follows. Sections 18.1 and 18.2 define aspects of general interest for the rest of the section: valid conversions of types between wrappers and base relations in Virtual DataPort and ways of specifying paths to resources. Section 18.3 specifies how to add to the system data sources of the various available types. Finally, section 18.4 shows how to create wrappers for each of these source types.

### 18.1 VALID CONVERSIONS BETWEEN TYPES IN WRAPPERS AND VDP TYPES

This section describes compatibility mappings between the Java types exported by the wrappers and the data types used by Virtual DataPort in the base relations and views (see section 3.1). When assigning wrappers to base relations it is important to bear these compatibility rules in mind to ensure that the defined schemas for the wrappers and base relations are compatible.

The following table shows mappings of the more common types. These are also the mappings applied automatically by the Virtual DataPort graphic administration tool (see Administration Guide [ADMIN_GUIDE]).

<table>
<thead>
<tr>
<th>Java Types</th>
<th>Virtual DataPort Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, java.lang.Short, java.lang.Integer</td>
<td>int</td>
</tr>
<tr>
<td>long, java.lang.Long</td>
<td>long</td>
</tr>
<tr>
<td>float, java.lang.Float</td>
<td>float</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>double</td>
</tr>
<tr>
<td>boolean, java.lang.Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>text</td>
</tr>
<tr>
<td>java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Timestamp, java.sql.Time</td>
<td>date</td>
</tr>
<tr>
<td>byte[], java.sql.Blob</td>
<td>blob</td>
</tr>
</tbody>
</table>

Automatic conversions between JAVA types and Virtual DataPort types

Any other java data type not specified in this table will be associated by default to the VDP data type **text**. Other possible mappings exist between Java types and Virtual DataPort types that can be specified but that are not applied automatically. These can be seen in the following table.

<table>
<thead>
<tr>
<th>Java Types</th>
<th>Virtual DataPort Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>enumerated</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>link</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>xml</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>money</td>
</tr>
</tbody>
</table>
Likewise, wrappers can provide compound elements such as arrays and registers that are directly associated with VDP arrays and registers.

18.1.1 Native-type Conversions of a Wrapper to Java Types

Each wrapper type has its own associations between native types of the sources modeled and java types. The following sections show the conversions applied to the different wrapper types supported by Virtual DataPort.

In general, for those wrappers that access sources that may return objects or arrays of objects the wrapper is responsible for representing these structures using Virtual DataPort registers and arrays respectively.

18.1.1.1 Type Conversion Tables for JDBC Wrappers

<table>
<thead>
<tr>
<th>JDBC types</th>
<th>Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>JDBCArrayVO proprietary class</td>
</tr>
<tr>
<td>BIGINT</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>BINARY</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>BIT</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>BLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>CLOB</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>DATALINK</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>FLOAT</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>INTEGER</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NULL</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>OTHER</td>
<td>JDBCRegisterVO</td>
</tr>
<tr>
<td>REAL</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>REF</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>STRUCT</td>
<td>JDBCRegisterVO</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TINYINT</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Other types are converted to java.lang.String.

**NOTICE**: the table shows the generic conversions associated to JDBC sources. Depending on the vendor and the version of the database which is being accessed, these conversions may vary slightly.
18.1.1.2 Type Conversion Table for ODBC Wrappers
For ODBC wrappers the same conversions are applied as for the JDBC wrappers.

18.1.1.3 Type Conversion Table for Web Source Wrappers
Wrappers for Web sources generated using ITPilot 4.0 or later use the following type conversion table:

<table>
<thead>
<tr>
<th>ITPilot Types</th>
<th>Java Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>date</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>url</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

ITPilot-type conversions

The Web source wrappers generated using versions of ITPilot prior to 4.0 do not provide data about the type of elements obtained. Therefore, they are encapsulated using the java class java.lang.String.

18.1.1.4 Type Conversion Table for Web Services Wrappers

<table>
<thead>
<tr>
<th>SOAP Types</th>
<th>Java Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>byte</td>
</tr>
<tr>
<td>xsd:dateTime</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>xsd:double</td>
<td>double</td>
</tr>
<tr>
<td>xsd:float</td>
<td>float</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:int</td>
<td>int</td>
</tr>
<tr>
<td>xsd:integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:long</td>
<td>long</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>java.lang.String with format &quot;{namespace}localPart&quot;</td>
</tr>
<tr>
<td>xsd:short</td>
<td>short</td>
</tr>
<tr>
<td>xsd:string</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Type Conversion Table for Web Services Wrappers

Compound elements are converted to Java objects by following the standard mapping defined by the JAX-RPC standard [JAXRPC].
### 18.1.1.5 Type Conversion Table for XML Wrappers

<table>
<thead>
<tr>
<th>XML/Schema Types</th>
<th>Java Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>positiveinteger</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>negativeinteger</td>
<td></td>
</tr>
<tr>
<td>nonpositiveinteger</td>
<td></td>
</tr>
<tr>
<td>nonnegativeinteger</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>unsignedint</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>gYear</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>gMonth</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>gDay</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>unsignedlong</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>unsignedbyte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>double</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>unsignedshort</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>normalizedString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>token</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>base64Binary</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>hexBinary</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>duration</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dateTime</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>date</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>time</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

### 18.1.1.6 Type Conversion Table for Delimited File Wrappers

DF wrappers always consider the extracted data as `java.lang.String`.

### 18.1.1.7 Type Conversion Table for CUSTOM Wrappers

A CUSTOM wrapper indicates the types of its fields with Java classes and, therefore, requires no conversion.

### 18.1.1.8 Type Conversion Table for Aracne Wrappers

All the fields in Aracne indexes are translated to attributes of type `text` in DataPort.

Wrappers created from Aracne indexes include some additional attributes besides the ones contained in the original index. These fields may be of other types. See section 18.4.6.2.
18.1.1.9 Type Conversion Table for Google Mini Wrappers

All the fields in Google Mini indexes are translated to attributes of type `text` in DataPort, except for the field `RATING` which is of `int` type.

Wrappers created from Google Mini indexes include some additional attributes besides the ones contained in the original index. These fields may be of other types. See section 18.3.9.

18.2 SPECIFYING PATHS IN VIRTUAL DATAPORT

Virtual DataPort needs paths to be specified in various points of the data source and wrapper creation processes.

There are three types of paths in Virtual DataPort. These are described below together with the parameters that generally need to be specified for each of these in a VQL statement:

- **LOCAL**: Path that accesses a resource in the local system. Requires the following parameters:
  - The class name used to implement the connection used by the path. For this type of path one sole connection class is provided: `LocalConnection`.
  - The local path to the resource (e.g. file).

- **HTTP**: Path that represents access to a resource through a Web server. The following parameters must be specified:
  - The class name used to implement the connection used by the path. For this type of path the server provides two different classes:
    - `http.CommonsHttpClientConnection`: Makes a connection to a Web server using the http protocol to access a remote resource. Optionally, it receives as a parameter the maximum time to wait for the response. For example, the following connection declaration indicates that this type of connection is used with a maximum response time of 2 minutes: `http.HTTPClientConnection, 120000`.
    - `http.DenodoBrowserPoolConnection`: Makes a connection to a Web server using the Denodo Browser [ITPILOT] which is capable of executing complex navigation sequences written in ITPilot’s NSEQL (Navigation SEQuence Language) [NSEQL]. The browser can be obtained from the internal browser pool or from a remote one. I.e. `HTTP 'http.DenodoBrowserPoolConnection, 3, 1'` will create a HTTP route using a browser obtained from the internal port. Change the second parameter to 2, to obtain the browser from a remote pool.
  - Access pattern `[uri]`. Represents a navigation sequence to a Web source the format of which should be understood by the connection class used. The class `http.HTTPClientConnection` allows specifying an http request (expressed in the normal format used for GET requests). ITPilot [ITPILOT] provides a navigation sequence language called NSEQL for the connection class `http.IEBrowserConnection`. In both cases the path can include interpolation variables the value of which will be obtained in execution time (see section 19.5).
- Access method (method). Indicates the http access method to be used with the path. Can take the values GET or POST. Currently, this parameter is only considered, if the class connection http.HTTPClientConnection is used.

- (Optional) Check SSL certificates (CHECKCERTIFICATES). If the http connection is secure (HTTPS) and this parameter is present, the connection will only be established if the Web server provides a certificate that the Java Virtual Machine considers valid. If this parameter is not present, any certificate will be accepted. See the keytool reference manual [KEYTOOL] for details on how to import an SSL certificate into the Sun Microsystems JVM.

- (Optional) Authentication information (authentication). If the http Server accessing the path requires authentication, this parameter sets the user identifier and password.

- (Optional) Proxy information (proxy). If the http access is performed through a proxy, the host name and port where the Proxy is running must be provided. If the Proxy is authenticated, valid user identification and password must be also provided. It is also possible to use the default http Proxy configuration (see the Administration Guide [ADMIN_GUIDE] to learn how to configure these default values) by using the “DEFAULT” option.

- **Denodo Browser**: The file is obtained using the Denodo Browser, which is capable of executing complex navigation sequences written in NSEQL (Navigation SEQuence Language). See the Denodo ITPilot User Guide [ITPILOT] and the NSEQL Manual [NSEQL] for more information about the Denodo Browser and the NSEQL sequences.

- **FTP / FTPS / SFTP**: Path that accesses a file via FTP. Receives as parameters:

  - The class name used to implement the connection used by the path (optional). For this type of path the server provides a sole connection class: ftp.FTPBeanConnection.
  
  - Server URL pointing to the resource (host:port/path/file).
  
  - User identifier that should be used for access and
  
  - Password for this user.

### 18.2.1 Filters

After defining the path to a resource, it is possible to establish filters that will be executed before processing the file. The available filters are:

- **UNZIP**: decompress a ZIP compressed file.
- **DECRYPT**: decrypts a file that was encrypted with the 'Password-Based-Encryption with MD5 and DES' algorithm. This encryption method is described in [JCA].

I.e. The command `CREATE DATASOURCE … ROUTE … FILTER UNZIP` will create a data source that will retrieve the data file, decompress it and finally process it.
18.3 CREATING DATA SOURCES

In Virtual DataPort, wrappers are associated to data sources. Data sources encapsulate the configuration needed to access a certain data repository used by Dataport (e.g. a database, a web service, etc.)

The following sections describe the manual creation process for each data source type.

**NOTE:** It is strongly recommended that the wrapper and data source creation process be undertaken graphically using the DataPort administration tool (see [ADMIN_GUIDE]).

18.3.1 JDBC Data Sources

To define a JDBC data source it is necessary to specify:
- **DRIVERCLASSNAME:** The driver class to be used for connection to the data source.
- **DATABASEURI:** The connection URL to the database.
- **USERNAME:** The user name to be used for access.
- **USERPASSWORD:** The password for the user. The ENCRYPTED modifier indicates that the provided password is encrypted (this option is typically used by the Denodo export/import process only).
- **CLASSPATH:** Path to the JAR file containing the JDBC driver for the specified source (optional).
- **Identification parameters for the database accessed (important for considering special characteristics of the different databases used as data source). These fields are optional. If not specified, then the general database access configuration is used.
  - **DATABASENAME:** Name of the database to be accessed.
  - **DATABASEVERSION:** Version number of the data source.
- **Parameters for initializing the connection pool associated with this data source (optional).**
  - **VALIDATIONQUERY:** SQL query used by the pool to verify the status of the cached connections. It is important for the query to be simple and that the table in question exists. If not specified, “SELECT COUNT (*) FROM SYS.DUAL” is used by default.
  - **INITIALSIZE:** Number of connections with which the pool is to be initialized. A number of connections are established and created in "idle" state, ready for use. 4 by default, if not specified.
  - **MAXACTIVE:** Maximum number of active connections the pool can manage at the same time. 8 by default, if not specified (a negative value implies no limit).
  - **TESTONBORROW:** if this property is set, and there is an active ping query, each connection retrieved from the connection pool will be validated by executing the ping query.
- **Data source configuration parameters (SOURCECONFIGURATION).** Virtual DataPort allows indicating specific characteristics of the underlying data sources, so that they are taken into account when executing statements on them. See section 18.3.13 for further details.

The **OR_REPLACE** modifier can also be specified in the data source creation statement. In this case, if a data source with the same name already exists, its definition will be substituted with the new one.

The create syntax of JDBC DataSources is shown in the following figure.

```
CREATE [ OR REPLACE ] DATASOURCE JDBC <name:identifier> 
  DRIVERCLASSNAME = <literal> 
  DATABASEURI = <literal> 
  USERNAME = <literal> 
  USERPASSWORD = <literal> [ ENCRYPTED ] 
  [ CLASSPATH = <literal> ] 
  [ DATABASENAME = <literal> DATABASEVERSION = <literal> ] 
  [ VALIDATIONQUERY = <literal> ]
```
INITIALSIZE = <integer>
MAXACTIVE = <integer>
[ TESTONBORROW = <boolean> ]

[ DESCRIPTION = <literal> ]
[ SOURCECONFIGURATION ( [ <source configuration property>
[, <source configuration property> ]* ] ) ]

<boolean> ::= true | false
<source configuration property> ::= [ DELEGATEALLOPERATORS = { true | false | DEFAULT }
| DELEGATEARRAYLITERAL = { true | false | DEFAULT }
| DELEGATECOMPOUNDFIELDPROJECTION = { true | false | DEFAULT }
| DELEGATEGROUPBY = { true | false | DEFAULT }
| DELEGATEHAVING = { true | false | DEFAULT }
| DELEGATEINNERJOIN = { true | false | DEFAULT }
| DELEGATEJOIN = { true | false | DEFAULT }
| DELEGATELEFTFUNCTION = { true | false | DEFAULT }
| DELEGATELEFTLITERAL = { true | false | DEFAULT }
| DELEGATENATURALOUTERJOIN = { true | false | DEFAULT }
| DELEGATENOTCONDITION = { true | false | DEFAULT }
| DELEGATEORCONDITION = { true | false | DEFAULT }
| DELEGATEORDERBY = { true | false | DEFAULT }
| DELEGATEPROJECTION = { true | false | DEFAULT }
| DELEGATEREGISTERLITERAL = { true | false | DEFAULT }
| DELEGATERIGHTFIELD = { true | false | DEFAULT }
| DELEGATERIGHTFUNCTION = { true | false | DEFAULT }
| DELEGATERIGHTLITERAL = { true | false | DEFAULT }
| DELEGATESELECTION = { true | false | DEFAULT }
| DELEGATEUNION = { true | false | DEFAULT }
| SUPPORTSAGGREGATEFUNCTIONSOPTIONS = { true | false | DEFAULT }
| SUPPORTSBRANCHOUTERJOIN = { true | false | DEFAULT }
| SUPPORTSEQOUTERJOINOPERATOR = { true | false | DEFAULT }
| SUPPORTSEXPLICITCROSSJOIN = { true | false | DEFAULT }
| SUPPORTSFULLEQOUTERJOIN = { true | false | DEFAULT }
| SUPPORTSEQUALNOTEQOUTERJOIN = { true | false | DEFAULT }
| SUPPORTSFUSINGINUSINGANDNATURALJOIN = { true | false | DEFAULT }
| SUPPORTSJOINONCONDITION = { true | false | DEFAULT }
| SUPPORTSUSINGJOIN = { true | false | DEFAULT }
| DELEGATEAGGREGATEFUNCTIONS = { DEFAULT | ( <function:identifier>
[, <function:identifier> ]* ) }
| DELEGATESCALARFUNCTIONS = { DEFAULT | ( <function:identifier>
[, <function:identifier> ]* ) }
| DELEGATEOPERATORSLIST = { DEFAULT | ( <operator:identifier>
[, <operator:identifier> ]* ) }

Figure 58 Syntax of the CREATE DATASOURCE JDBC statement

A JDBC data source modification statement exists (ALTER DATASOURCE JDBC). The syntax allows indicating the same parameters as the creation statement.
ALTER DATASOURCE JDBC <name:identifier>
[ DRIVERCLASSNAME = <literal> ]
[ DATABASEURI = <literal> ]
[ USERNAME = <literal> ]
[ USERPASSWORD = <literal> [ ENCRYPTED ] ]
[ CLASSPATH = <literal> ]
[ DATABASENAME = <literal> DATABASEVERSION = <literal> ]
[ VALIDATIONQUERY = <literal>
  INITIALIZE = <integer>
  MAXACTIVE = <integer>
  TESTONBORROW = <boolean> ]
[ DESCRIPTION = <literal> ]
[ SOURCECONFIGURATION ( [ <source configuration property>
  [, <source configuration property> ]* ] ) ]

<boolean> ::= true | false
<source configuration property> ::= (see CREATE DATASOURCE JDBC for details)

Figure 59 Syntax of the ALTER DATASOURCE JDBC statement

18.3.2 ODBC Data Sources

Figure 60 shows the syntax of the VQL statement for creating an ODBC data source. For more information on the different parameters that must be established to define the connection and to define the pool of connections for the data source, see section 18.3.1.

The data source creation statement also allows for the OR REPLACE modifier to be specified. In this case, if there is already a data source with the same name, its definition will be replaced with the new one.

If the DATABASEURI parameter contains a path to a file, this path can contain interpolation variables. See section 19.5 to learn how to create base views that use a data sources with interpolation variables.

Configuration of different parameters belonging to the data source can also be specified (SOURCECONFIGURATION). See section 18.3.13 for further details.

In the case of ODBC data sources, the driver class to be used for the connection to the may not be specified. In that case, the DSN attribute should be specified. When the DSN attribute is specified, the driver used will be the JDBC/ODBC bridge driver.

NOTE: In the case of ODBC source types, the accessed data source should be located in the local machine of the Virtual DataPort server or, where not possible, an ODBC management system must be installed in which the ODBC driver of the remote database server should be registered.

CREATE [OR REPLACE] DATASOURCE ODBC <name:identifier>
{   DSN = <literal>
    | DATABASEURI = <literal>
    | DRIVERCLASSNAME = <literal>
  }
USERNAME = <literal>
USERPASSWORD = <literal> [ ENCRYPTED ]
| PROPERTIES = <literal> ]

Generating Wrappers and Data Sources 107
[ CLASSPATH = <literal> ]
[ DATABASENAME = <literal>
  DATABASEVERSION = <literal>
]
[ VALIDATIONQUERY = <literal>
  INITIALSIZE = <integer>
  MAXACTIVE = <integer>
  TESTONBorrow = <boolean> ]
[
  DESCRIPTION = <literal> ]
[ SOURCECONFIGURATION ( [ <source configuration property>
  [, <source configuration property> ]* ] ) ]

<source configuration property> ::= 
  | DELEGATEALLOPERATORS = { true | false | DEFAULT }
  | DELEGATEARRAYLITERAL = { true | false | DEFAULT }
  | DELEGATECOMPOUNDFIELDPROJECTION = { true | false | DEFAULT }
  | DELEGATEGROUPBY = { true | false | DEFAULT }
  | DELEGATEHAVING = { true | false | DEFAULT }
  | DELEGATEINNERJOIN = { true | false | DEFAULT }
  | DELEGATEJOIN = { true | false | DEFAULT }
  | DELEGATELEFTFUNCTION = { true | false | DEFAULT }
  | DELEGATELEFTLITERAL = { true | false | DEFAULT }
  | DELEGATENATURALOUTERJOIN = { true | false | DEFAULT }
  | DELEGATENOTCONDITION = { true | false | DEFAULT }
  | DELEGATENORCONDITION = { true | false | DEFAULT }
  | DELEGATERIGHTFUNCTION = { true | false | DEFAULT }
  | DELEGATERIGHTFIELD = { true | false | DEFAULT }
  | DELEGATERIGHTLITERAL = { true | false | DEFAULT }
  | DELEGATESELECTION = { true | false | DEFAULT }
  | DELEGATEUNION = { true | false | DEFAULT }
  | SUPPORTSAGGREGATEFUNCTIONSOPTIONS = { true | false | DEFAULT }
  | SUPPORTSBRANCHOUTERJOIN = { true | false | DEFAULT }
  | SUPPORTSEQOUTERJOINOPERATOR = { true | false | DEFAULT }
  | SUPPORTSEXPLICITCROSSJOIN = { true | false | DEFAULT }
  | SUPPORTSFULLNOTEQOUTERJOIN = { true | false | DEFAULT }
  | SUPPORTSFUSINGINUSINGANDNATURALJOIN = { true | false | DEFAULT }
  | SUPPORTSNATURALJOIN = { true | false | DEFAULT }
  | SUPPORTSUSINGJOIN = { true | false | DEFAULT }
  | DELEGATEAGGREGATEFUNCTIONS = { DEFAULT | ( <function:identifier>
    [, <function:identifier> ]* ] ) }
  | DELEGATESCALARFUNCTIONS = { DEFAULT | ( <function:identifier>
    [, <function:identifier> ]* ] ) }
  | DELEGATEOPERATORSLIST = { DEFAULT | ( <operator:identifier>
    [, <operator:identifier> ]* ] ) }

Figure 60 Syntax of the CREATE DATASOURCE ODBC statement

The ALTER DATASOURCE ODBC has the same syntax as the creation statement.
Virtual DataPort 4.6 Advanced VQL Guide

18.3.3 Multidimensional Data Sources

Virtual DataPort can obtain data from multidimensional databases such as SAP BW3, SAP BI 7 or Mondrian.

**Important**: before creating a multidimensional database source that connects to SAP BW or SAP BI, we need to install the SAP Business Intelligence JDK in the system where Virtual DataPort is running. The appendix ‘Installing the Connector for SAP BW and SAP BI (Multidimensional Sources)’ of the Administration Guide [ADMIN_GUIDE] explains how to do this.

Although the Administration Tool only lists ‘Multidimensional DB’ sources, the Server distinguishes between two types of multidimensional data sources: SAP multidimensional sources and generic multidimensional sources (OLAP4J) such as Mondrian.

Figure 62 and Figure 63 contain the syntax of the commands to create and modify data sources that connect to SAP BW and SAP BI multidimensional databases.

```
CREATE [ OR REPLACE ] DATASOURCE SAPBW <name:identifier>
[ DATABASENAME = <literal> DATABASEVERSION = <literal> ]
XMLAURI = <literal>
SystemName = <literal>
LANGUAGE = <literal>
USERNAME = <literal>
USERPASSWORD = <literal> [ENCRYPTED]
[ DESCRIPTION = <literal> ]
```

**Figure 62** Syntax of the CREATE DATASOURCE SAPBW statement

```
ALTER DATASOURCE ODBC <name:identifier>
[ DSN = <literal> ]
[ DATABASEURI = <literal> DRIVERCLASSNAME = <literal> ]
[ USERNAME=<literal> USERPASSWORD = <literal> [ ENCRYPTED ] ]
[ PROPERTIES = <literal> ]
[ CLASSPATH = <literal> ]
[ DATABASENAME = <literal>
DATABASEVERSION = <literal> ]
[ INITIALIZE = <integer>
MAXACTIVE = <integer>
VALIDATIONQUERY = <literal>
[ TESTONBORROW = { true | false } ]
]
[ DESCRIPTION = <literal> ]
[ SOURCECONFIGURATION ( [ <source configuration property>
[, <source configuration property> ]* ) ]
```

<source configuration property> ::= (see CREATE DATASOURCE ODBC for details)

**Figure 61** Syntax of the ALTER DATASOURCE ODBC statement
### 18.3.4 Data Sources for Web Services

To configure a Web service as data source, the following data must be specified:

- The URI to the WSDL file that defines the Web Service. The WSDL file defines one or several web services, where each service may be comprised of different ports with one or several operations each. A data source for web services will allow the creation of wrappers modeling any of the operations that they define.

- (Optional) If the WSDL points to an incorrect URL or does not contain one, we can make the new data source use another URL to connect to the source. If this parameter is not present, the new data source will use the URL of the WSDL. Otherwise, we can:
  - Indicate an URL with the parameter `ENDPOINT URI`.
  - Indicate the name of a variable (parameter `ENDPOINT VAR`). All the views created over this data source will have a field with the value of this parameter `[var_name]`. This is useful when the end point changes regularly or is obtained from another source at runtime.

- (Optional) Authentication information. The supported authentication methods are:
- **HTTP Basic** or **HTTP Digest** [HTTP_AUTH].

- **HTTP NTLM**. Uses the Microsoft's NT LAN Manager (NTLM) Authentication Protocol [MS_NLMP] to access Microsoft Windows servers. Virtual DataPort supports NTLM v1 and NTLM v2.

- **WSS Basic** and **WSS Digest**. Web Services Security [WSS] is a standard for the implementation of security features in applications using Web services. Currently, Denodo supports the authentication profile “Username Token” [WSS_UT].

When using the parameters `<credentials_with_vars>` to create the data source, the base views created over this data source will have two extra fields (or three, if using the NTLM authentication) which value will be used as credentials to access the Web service. The name of these fields will be the value of the parameters VAR.

For example, the views created over the following data source will have two extra fields: `login_var` and `password_var`, which value will be used as credentials.

```
CREATE DATASOURCE WS ... 
...

AUTHENTICATION HTTP BASIC ( 
    USER 'anonymous' VAR login_var
    PASSWORD 'anonymous' VAR password_var ) 
...;
```

- (Optional). Proxy information. If the Web Service access is performed through a proxy, the host name and port where the Proxy is running must be provided. If the Proxy required authentication, we also have to provide the proxy credentials The **ENCRYPTED** modifier indicates that the provided password is encrypted (usually, this modifier is only used by the export/import process of Denodo Platform). It is also possible to use the default http Proxy configuration (see the Administration Guide [ADMIN_GUIDE] to learn how to configure these default values) using the **DEFAULT** option.

Figure 66 shows the create syntax of a data source for a Web service.

The data source creation statement also allows specifying the OR REPLACE modifier. In this case, if there is already a data source with the same name, its definition will be replaced with the new one.
CREATE [ OR REPLACE ] DATASOURCE WS <name:identifier>
   WSDLURI = <literal>
   [ <endpoint> ]
   [ CHECKCERTIFICATES ]
   [ <authentication> ]
   [ <proxy> ]
   [ DESCRIPTION = <literal> ]

<endpoint> ::= 
   ENDPOINT URI = <literal>
   | ENDPOINT VAR = <var_name:identifier>

<authentication> ::= AUTHENTICATION {
   OFF
   | { HTTP BASIC | HTTP DIGEST | WSS BASIC | WSS DIGEST }
   ( { <credentials> | <credentials_with_vars> } )
   | HTTP NTLM ( <ntlm_credentials> )
}

<credentials> ::= USER <literal> PASSWORD <literal> [ ENCRYPTED ]

<credentials_with_vars> ::= {
   USER <literal> VAR <user:identifier> PASSWORD <literal> VAR
   <password:identifier>
}

<ntlm_credentials> ::= {
   <credentials> [ DOMAIN <literal> ]
   | <credentials_with_vars> [DOMAIN <literal> VAR <domain:identifier> ]
}

<proxy>::= PROXY {
   OFF | DEFAULT | ON ( HOST <literal> PORT <integer> [ <credentials> ] )
}

Figure 66  Syntax of the CREATE DATASOURCE WS statement

The modification statement of a data source of this type is similar.
**18.3.5 XML Data Sources**

Virtual DataPort allows using XML as data sources. To define an XML data source it is necessary to specify the access path to the XML document and, optionally, the access path to the file containing the schema or DTD of same.

- **SCHEMA** or **DTD** (optional): Path to the file that contains the metadata of the data source XML file. It may be an XML Schema or a DTD. If it is not specified, Virtual DataPort will try to infer an appropriate schema by analyzing the XML document structure indicated in the next parameter.

- **ROUTE**: Specification of the access path to the XML file that represents the data source. This may include interpolation variables to parameterize the access path depending on the conditions of the query executed on the data source (see section 19.5).

- **FILTER**: List of filters that will be applied to a file before processing it. They can be applied to the XML file and the XML Schema or DTD (see section 18.2.1)

The path specification and file filtering (UNZIP and DECRYPT) were described in section 18.2 and 18.2.1 respectively. The creation syntax can be seen in Figure 68:
**Figure 68** Syntax of the CREATE DATASOURCE XML statement

The data source creation statement also allows specifying the **OR REPLACE** modifier. In this case, if there is already a data source with the same name, its definition will be replaced with the new one.

The syntax of the modification statement of an XML data source is shown below.
18.3.6 JSON Data Sources

Virtual DataPort allows using JSON files as data sources. To define a JSON data source it is necessary to specify the access path to the document (ROUTE element). The path may include interpolation variables to parameterize the access path depending on the conditions of the query made on the data source (see section 18.4.1). Path specification and file filtering (UNZIP and DECRIPT) were described in section 18.2 and 18.2.1. The creation syntax can be seen in Figure 70:

```
CREATE [ OR REPLACE ] DATASOURCE JSON <name:identifier>
    ROUTE <route> [ <route_filters> ]
    [ DESCRIPTION = <literal> ]

<route> ::= {
    LOCAL <connection class name:literal> <uri:literal>
    | HTTP <connection class name:literal> { GET | POST } <uri:literal>
    | POSTBODY <post_body:literal> [ MIME <body mime_type:literal> ]
    [ CHECKCERTIFICATES ] [ <authentication> ] [ <proxy> ]
    | FTP <connection class name:literal> <uri:literal> <login:literal>
        <password:literal> [ ENCRYPTED ]
}
```

Figure 70 Syntax of the CREATE DATASOURCE JSON statement
If there is already a data source with the same name, the OR REPLACE modifier allows replacing its definition by the new one. Below is the syntax of the modification statement of a JSON data source.

```
ALTER DATASOURCE JSON <name:identifier>
   [ ROUTE <route> [ <route_filters> ] ]
   [ DESCRIPTION = <literal> ]

<route> ::= 
   LOCAL <connection class name:literal> <uri:literal>
   | HTTP <connection class name:literal> { GET | POST } <uri:literal>
   [ POSTBODY <post_body:literal> [ MIME <body mime_type:literal> ] ]
   [ CHECKCERTIFICATES ] [ <authentication> ] [ <proxy> ]
   | FTP <connection class name:literal> <uri:literal> <login:literal>
   [ password:literal] [ ENCRYPTED ]

<route_filters> ::= FILTER ( <filter> [, <filter> ]* )

<filter> ::= 
   DECRIPT PASSWORD = <literal> [ ENCRYPTED ]
   | UNZIP

<authentication> ::= AUTHENTICATION {
   OFF | { BASIC | DIGEST } ( <credentials> )
   | NTLM ( <ntlm_credentials> )
}

<proxy> ::= PROXY {
   OFF | DEFAULT | ON { HOST <literal> PORT <integer> [ <credentials> ] } }

<credentials> ::= USER <literal> PASSWORD <literal> [ ENCRYPTED ]

<ntlm_credentials> ::= <credentials> [ DOMAIN <literal> ]
```

Figure 70 Syntax of the creation statement of a JSON data source

Figure 71 Syntax of the modification statement of a JSON data source
18.3.7 DF Data Sources

This type of data source enables Denodo Virtual DataPort to access the data contained in flat files in CSV format (Comma Separated Values) and other flat text files with data that can be extracted through the use of regular expressions.

See section 19.2.4 to get details on how to improve the performance of the DF data sources.

To define a data source of a delimited file the following elements must be specified:

- **ROUTE**: The path to the delimited-type text file from which data are to be extracted (see section 18.2). This may include interpolation variables to parameterize the access path depending on the conditions of the query executed on the data source (see section 19.5).

- **FILTER**: List of filters that will be applied to the data file before processing it. The available filters are **UNZIP** and **DECRYPT** (see section 18.2.1)

- **CHARSET**: It specifies the charset encoding used by the file. Any charset encoding supported by JAVA can be used [JAVACHARSETS].

- **COLUMNDELIMITER**: Character string used as an element separator in the delimited file. It is only used if no Tuple Pattern is indicated.

- **TUPLEPATTERN**: Regular expression that specifies the format of the tuples that will be extracted from the delimited file. This regular expression has to match the whole line that wants to capture, not only part of it.

  The format used is that of regular expressions in JAVA language [REGEXP].

  The fields of the views will be the capturing groups of the regular expression.

  The section ‘Delimited File Sources’ of the Virtual DataPort Administration Guide [ADMIN_GUIDE] contains examples of tuple patterns.

  **Note**: the tuple pattern can contain interpolation variables.

- **ENDOFLINEDELIMITER**: Character string used as data tuple separator in the delimited file (the carriage return \n will be used by default).

- **BEGINDELIMITER**: A JAVA regular expression identifying the position in the file where the system must start searching for tuples (or searching for the header if the ‘header’ option was checked). If not value is specified, the search will start at the beginning of the file. If the ISDATA modifier is added, then the text matching with the regular expression will be considered as part of the search space. This may include interpolation variables to parameterize the access path depending on the conditions of the query executed on the data source (see section 19.5).

- **ENDDELIMITER**: A JAVA regular expression identifying the position in the file where the system must stop searching for tuples. If not value is specified, the search will continue until the end of the file. If the ISDATA modifier is added then the text matching with the regular expression will be considered as part of the search space. This may include interpolation variables to parameterize the access path depending on the conditions of the query executed on the data source (see section 19.5).

- **HEADER**: If given the **true** value, it is assumed that the first tuple extracted from the file data area contains the names of the fields. These names will be used to create the attributes of the base relation for Virtual DataPort.

- **HEADERPATTERN**: This indicates a regular expression-type pattern to be used to extract the name of the fields forming the header. This must only be specified if the pattern to be used to extract the header is different to that used to extract the tuples. The format of the regular expressions is the same as that used for the Tuple Pattern. This field can only be used when the Header check box is marked.
The data source creation statement also allows specifying the **OR REPLACE** modifier. In this case, if there is already a data source with the same name, its definition will be replaced with the new one.

```
CREATE [ OR REPLACE ] DATASOURCE DF <name:identifier>
  ROUTE <route>
  [ CHARSET = <literal> ]
  [ <route_filters> ]
  [ COLUMNDELIMITER = <literal>]
  | TUPLEPATTERN = <literal> [ HEADERPATTERN = <literal> ]
  }
  [ ENDOFLINEDELIMITER = <literal> ]
  [ BEGINDELIMITER = <literal> [ISDATA] ]
  [ ENDDELIMITER = <literal> [ISDATA] ]
  [ HEADER = <boolean> ]
  [ DESCRIPTION = <literal> ]

<route> ::= { LOCAL <connection class name:literal> <uri:literal>
  | HTTP <connection class name:literal> [ GET | POST ] <uri:literal>
  [ POSTBODY <post_body:literal> [ MIME <body mime_type:literal> ] ]
  [ CHECKCERTIFICATES ] [ [ <authentication> ] [ [ <proxy> ] ]
  | FTP <connection class name:literal> <login:literal> <password:literal>
  <uri:literal> [ ENCRYPTED ] }

<route_filters> ::= FILTER ( <filter> [, <filter> ]* )

<filter> ::= { DECRYPT PASSWORD = <literal> [ ENCRYPTED ]
  | UNZIP }

<authentication> ::= AUTHENTICATION { OFF | { BASIC | DIGEST } ( <credentials> )
  | NTLM ( <ntlm_credentials> ) }

<proxy>::= PROXY { OFF | DEFAULT | ON { HOST <literal> PORT <integer> [ <credentials> ] } }

<credentials> ::= USER <literal> PASSWORD <literal> [ ENCRYPTED ]

<ntlm_credentials> ::= <credentials> [ DOMAIN <literal> ]
```

**Figure 72** Syntax of the CREATE DATASOURCE DF statement

Figure 73 shows the syntax of the modification statement of a delimited file data source.
ALTER DATASOURCE DF <name:identifier>
    ROUTE <route> [ CHARSET = <literal> ] [ <route_filters> ]
    { COLUMNDELMITER = <literal>
        | TUPLEPATTERN = <literal> [ HEADERPATTERN = <literal> ]
    }
    [ ENDOFLINEDELMITER = <literal> ]
    [ BEGINDELMITER = <literal> [ISDATA] ]
    [ ENDDELIMITER = <literal> [ISDATA] ]
    [ HEADER = <boolean> ]
    [ DESCRIPTION = <literal> ]

<route> ::= { LOCAL <connection class name:literal> <uri:literal>
    | HTTP <connection class name:literal> { GET | POST } <uri:literal>
    [ POSTBODY <post_body:literal> [ MIME <body mime_type:literal> ] ]
    [ CHECKCERTIFICATES ] [<authentication>] [<proxy>]
    | FTP <connection class name:literal> <uri:literal> <login:literal>
        <password:literal> [ ENCRYPTED ]
}

<route_filters> ::= FILTER ( <filter> [, <filter> ]* )

<filter> ::= {
    DECRYPT PASSWORD = <literal> [ ENCRYPTED ]
    | UNZIP
}

<authentication> ::= AUTHENTICATION {
    OFF | { BASIC | DIGEST } ( <credentials> )
    | NTLM ( <ntlm_credentials> )
}

<proxy>::= PROXY {
    OFF | DEFAULT | ON ( HOST <literal> PORT <integer> [ <credentials> ] )
}

<credentials>::= USER <literal> PASSWORD <literal> [ ENCRYPTED ]

<ntlm_credentials>::= <credentials> [ DOMAIN <literal> ]

Figure 73 Syntax of the ALTER DATASOURCE DF statement

18.3.8 Denodo Aracne Data Sources

Virtual DataPort allows using a Denodo Aracne search server [ARCN] as a data source. The following parameters must be specified:

- **name**: Name to be given to the data source in Virtual DataPort.
- **ARNURI**: Access URI to the Aracne search server. The URI format is `host:port`, being `host` the name of the machine that hosts the search engine. The port is 9000 in the Aracne default installation.
- **LOGIN**: The user login to access the Denodo Aracne search/index engine server (this parameter must be specified for versions greater or equal to 4.5).
- **PASSWORD**: The user password to access the Denodo Aracne search/index engine server (this parameter must be specified for versions greater or equal to 4.5).

The creation syntax can be seen in Figure 74:
**18.3.9 Google Mini Data Sources**

Virtual DataPort can use a Google Enterprise Search [GMINI] search engine as data source. The following parameters must be specified:

- **name**: Name to be given to the data source in Virtual DataPort.
- **GSURI**: Access URI to the Google Enterprise search server. The URI format is `host:port`, being `host` the name of the machine that hosts the search engine.
- **Proxy Configuration**: If the http access is performed through a proxy, the host name and port where the Proxy is running must be provided. If the Proxy is authenticated, valid user identification and password must be also provided. It is also possible to use the default http Proxy configuration (see the Administration Guide [3] to learn how to configure these default values) by using the “DEFAULT” option.

The creation syntax can be seen in Figure 76:

```
CREATE [ OR REPLACE ] DATASOURCE GS <name:identifier>
GSURI = <literal>
[ PROXY [OFF |DEFAULT | ON ( HOST <literal> PORT <integer> [USER <literal>]
[ PASSWORD <literal> [ ENCRYPTED ]]) ] ]
[ DESCRIPTION = <literal> ]
```

*Figure 76* Syntax of the create statement of a Google Mini data source

If there is already a data source with the same name, the modifier OR REPLACE will allow for its definition to be replaced with the new one.

Below is the syntax of the modification statement for a Google Mini data source.

```
ALTER DATASOURCE GS <name:identifier>
GSURI = <literal>
[ PROXY [OFF |DEFAULT | ON ( HOST <literal> PORT <integer> [USER <literal>
[ PASSWORD <literal> [ ENCRYPTED ]]) ] ]
[ DESCRIPTION = <literal> ]
```

*Figure 77* Syntax of the modification statement of a Google Mini data source
18.3.10 LDAP Data Sources

Virtual DataPort allows using an LDAP server as a data source. Imported LDAP servers can be used to extract data from them and also to authenticate Virtual DataPort against them (see section 11.3.3). The following figure contains the syntax of the commands to deal with LDAP data sources:

```
CREATE [ OR REPLACE ] DATASOURCE LDAP <name:identifier>
  URI = <serverURI:literal>
  [ USERNAME = <userName:literal>]
  [ USERPASSWORD = <password:literal> [ ENCRYPTED ] ]
  [ USEPAGING = { TRUE | FALSE } [ MAXPAGESIZE = <integer> ] ]
  [ DESCRIPTION = <literal> ]
```

**Figure 78** Syntax of the create statement of an LDAP data source

- **name**. Name of the new data source in Virtual DataPort.
- **URI**. URI of the LDAP server. The URI format is `ldap://host:port`
- **USERNAME** / **USERPASSWORD**. Credentials to access the LDAP server (optional). The `ENCRYPTED` modifier indicates the password is provided encrypted. Usually, this modifier is only used by Virtual DataPort metadata import/export process (see section 12.1).
- **USEPAGING**. If `TRUE`, Virtual DataPort will do paged searches to obtain all the results of the queries, instead of obtaining all the results at once. `MAXPAGESIZE` is the number of results per page. This option is useful if the LDAP server has a limit on the number of results per query.

If there is already a data source with the same name, the modifier `OR REPLACE` will allow for its definition to be replaced with the new one.

Below is the syntax of the modification statement for an Aracne data source.

```
ALTER DATASOURCE LDAP <name:identifier>
  URI = <serverURI:literal>
  [ USERNAME = <userName:literal>]
  [ USERPASSWORD = <password:literal> [ ENCRYPTED ] ]
  [ USEPAGING = { TRUE | FALSE } [ MAXPAGESIZE = <integer> ] ]
  [ DESCRIPTION = <literal> ]
```

**Figure 79** Syntax of the modification statement of an LDAP data source

18.3.11 BAPI Data Sources

Virtual DataPort can invoke SAP BAPIs (Business Application Programming Interfaces) to obtain data stored in SAP ERP and other SAP applications.

**Important**: before creating any BAPI data source, we have to install the SAP Java Connector 3 in the system where Virtual DataPort is running. The appendix "Installing the connector for SAP ERP (BAPI data sources)" of the Administration Guide [ADMIN_GUIDE] explains how to do this.
Figure 80 and Figure 81 contain the syntax of the commands to create and modify data sources that connect to SAP systems:

**CREATE [ OR REPLACE ] DATASOURCE SAPERP <name:identifier>**

- `SystemName = <literal>`
- `HostName = <literal>`
- `ClientID = <literal>`
- `SystemNumber = <literal>`
- `USERNAME = <literal>`
- `USERPASSWORD = <literal> [ENCRYPTED]`
- `[ DESCRIPTION = <literal> ]`

**Figure 80** Syntax of the CREATE DATASOURCE SAPERP sentence

**ALTER DATASOURCE SAPERP <name:identifier>**

- `[ SystemName = <literal> ]`
- `[ HostName = <literal> ]`
- `[ ClientID = <literal> ]`
- `[ SystemNumber = <literal> ]`
- `[ USERNAME = <literal> ]`
- `[ USERPASSWORD = <literal> [ENCRYPTED] ]`
- `[ DESCRIPTION = <literal> ]`

**Figure 81** Syntax of the ALTER DATASOURCE SAPERP sentence

All the parameters of these two commands refer to the connection details to the SAP instance.

### 18.3.12 Custom Data Sources

Virtual DataPort allows creating wrappers ad-hoc for data sources for which no standard connector is available. To do so, two JAVA classes must be created to implement the required behavior (see section 18.4.15). Once these classes have been created, it is possible to import the data source to DataPort using a CUSTOM data source. The following parameters must be specified:

- **name**: Name to be given to the data source in Virtual DataPort.
- **CLASSNAME**: Name of the class implementing the specific wrapper for the source. It must extend `com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl`. See section 18.4.15.
- **CLASSPATH** (optional): Additional classpath required for running the wrapper.

The creation syntax can be seen in Figure 82:

**CREATE [ OR REPLACE ] DATASOURCE CUSTOM <name:identifier>**

- `CLASSNAME=<className:literal>`
- `[ CLASSPATH=<classpath:literal> ]`
- `[ JARS = <jar name:literal> [, <jar name:literal>]* ]`
- `[ DESCRIPTION = <literal> ]`

**Figure 82** Syntax of the create statement of a Custom data source

Below is the syntax of the modification statement for a Custom data source.
18.3.13 Data Source Configuration Properties

Data Source Configuration properties allow specifying certain characteristics of the underlying data sources such as the operations they support. Knowing the capacities of each data source is important for optimization reasons since it allows Virtual DataPort to delegate to the data source as much processing as possible to optimize response times and minimize traffic through the network.

**NOTE:** Typically, users do not need to edit this information since DataPort automatically uses suitable configurations for most common data sources.

The properties of each data source can be configured by adding parameter/value pairs to the data source creation statement or graphically using the administration tool (see Virtual DataPort Administration Guide [ADMIN_GUIDE]. The configurable properties are as follows:

- **Delegate All Operators** (**DELEGATEALLOPERATORS**, DS: JDBC, ODBC). This indicates whether the source allows for all operators to be delegated. The value is “false” by default.

- **Delegate Array Literal** (**DELEGATEARRAYLITERAL**, DS: JDBC, ODBC). This indicates whether the source allows for array-type compound constants to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate Compound Field Projection** (**DELEGATECOMPOUNDFIELDPROJECTION**, DS: JDBC, ODBC). This indicates whether the source allows projections on compound fields to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate GROUP BY** (**DELEGATEGROUPBY**, DS: JDBC, ODBC). This indicates whether the source allows the GROUP BY clause to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate HAVING clause** (**DELEGATEHAVING**, DS: JDBC, ODBC). This indicates whether the source allows the HAVING clause to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate Inner Join** (**DELEGATEINNERJOIN**, DS: JDBC, ODBC). This indicates whether the source allows for the Inner Join operator to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate Join** (**DELEGATEJOIN**, DS: JDBC, ODBC). This indicates whether the source allows for the Join operator to be delegated. The value is “true” by default for JDBC and ODBC sources.

- **Delegate Left Function** (**DELEGATELEFTFUNCTION**, DS: JDBC, ODBC). This indicates whether the source allows for conditions with functions on the left part to be delegated. The value is “true” by default for JDBC and ODBC sources.
• **Delegate Left Literal** (**DELEGATENATURALOUTERJOIN**, DS: JDBC, ODBC). This indicates whether the source allows for conditions with constants on the left part to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Natural Outer Join** (**DELEGATENOTCONDITION**, DS: JDBC, ODBC). This indicates whether the source allows the NOT condition to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate OR Condition** (**DELEGATENATURALOUTERJOIN**, DS: JDBC, ODBC). This indicates whether the source allows for the OR condition to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate ORDER BY** (**DELEGATEORDERBY**, DS: JDBC, ODBC). This indicates whether the source allows the ORDER BY clause to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Projection** (**DELEGATEPROJECTION**, DS: JDBC, ODBC). This indicates whether the source allows for projections to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Register Literal** (**DELEGATEREGISTERLITERAL**, DS: JDBC, ODBC). This indicates whether the source allows for the use of literals with register data type. The value is “false” by default for JDBC and ODBC sources.

• **Delegate Right Field** (**DELEGATERIGHTFIELD**, DS: JDBC, ODBC). This indicates whether the source allows for the use of fields on the right part of the conditions. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Right Function** (**DELEGATERIGHTFUNCTION**, DS: JDBC, ODBC). This indicates whether the source allows for conditions with functions on the right part to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Right Literal** (**DELEGATERIGHTLITERAL**, DS: JDBC, ODBC). This indicates whether the source allows for conditions with constants on the right part to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate Selection** (**DELEGATESELECTION**, DS: JDBC, ODBC). This indicates whether the source allows for conditions to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Delegate UNION** (**DELEGATEUNION**, DS: JDBC, ODBC). This indicates whether the source allows for the union operator to be delegated. The value is “true” by default for JDBC and ODBC sources.

• **Supports Modifier in Aggregate Function** (**SUPPORTSAGGREGATEFUNCTIONSOPTIONS**, DS: JDBC, ODBC). This indicates whether the source supports DISTINCT/ALL modifiers in aggregate functions. The value is “true” by default for JDBC and ODBC sources.
• **Supports Branch Outer Join** ([SUPPORTSBRANCHOUTERJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for (left | right) outer join to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Eq Outer Join** ([SUPPORTSEQOUTERJOINOPERATOR](#), DS: JDBC, ODBC). This indicates whether the source allows for the Equality Outer Join operator to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Explicit Cross Join** ([SUPPORTSEXPLICITCROSSJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for the Explicit Cross Join operator to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Full Eq Outer Join** ([SUPPORTSFULLEQOUTERJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for the Full Equality Outer Join operator to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Full NotEq Outer Join** ([SUPPORTSFULLNOTEQOUTERJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for the Full Not Equality Outer Join operator to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Fusing in using AND Natural Join** ([SUPPORTSFUSINGINUSINGANDNATURALJOIN](#), DS: JDBC, ODBC). This indicates if the source merges the same fields when running a natural join or a join with the USING clause. The value is “false” by default for JDBC and ODBC sources.

• **Supports Join On Condition** ([SUPPORTSJOINONCONDITION](#), DS: JDBC, ODBC). This indicates whether the source allows for the Join On clause to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Natural Join** ([SUPPORTSNATURALJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for the Natural Join clause to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Supports Using Join** ([SUPPORTSUSINGJOIN](#), DS: JDBC, ODBC). This indicates whether the source allows for the Using Join clause to be delegated. The value is “false” by default for JDBC and ODBC sources.

• **Delegate Aggregate Functions List** ([DELEGATEAGGREGATEFUNCTIONS](#), DS: JDBC, ODBC). This indicates the aggregation functions that can be delegated. In JDBC and ODBC sources, the list is made up of the [AVG](#), [COUNT](#), [MAX](#), [MIN](#) and [SUM](#) functions.

• **Delegate Scalar Functions List** ([DELEGATESCALARFUNCTIONS](#), DS: JDBC, ODBC). This indicates the scalar functions that can be delegated. In JDBC and ODBC sources.

• **Delegate Operators List** ([DELEGATEOPERATORSLIST](#), DS: JDBC, ODBC). This indicates the operators that can be delegated. In JDBC and ODBC sources, the list is made up of the [=](#), [<>](#), [<](#), [<=](#), [>](#), [>=](#), [in](#), [between](#), [contains](#), [containsor](#), [like](#), [is null](#), [is not null](#), [is true](#) and [is false](#) operators.
• **Operator Properties.** This allows specifying the support provided by the data source for a specific operator. For each operator, the name (operator_name attribute) and its list of properties are specified. Currently, these properties only exist for the `contains` operator (see section 18.3.13.1).

**Example:** Suppose that the creation of a data source from a MySQL relational source of a very old version does not allow for the `USING` clause in joins. VDP includes this parameter with a "true" value by default and, therefore, it must be changed. To do so, the value must be altered in the creation statement as follows:

```sql
CREATE DATASOURCE JDBC OldMySQL
    DRIVERCLASSNAME = 'com.mysql.jdbc.Driver'
    DATABASEURI = 'jdbc:mysql://localhost/vdp_demo'
    USERNAME = 'user'
    USERPASSWORD = 'userpwd'
#Configuration parameters ...
SOURCECONFIGURATION(
    SUPPORTSUSINGJOIN = false
);
```

**Figure 84** Example of altering a data source configuration

Virtual DataPort has default values for some specific relational databases (MySQL, Oracle, PostgreSQL, etc.) that may vary in relation to those described above.

18.3.13.1 **CONTAINS Operator Configuration Properties**

The `CONTAINS` operator allows executing complex Boolean keyword searches on text-type attributes from an external index of unstructured data (e.g. Aracne and/or Google Enterprise data sources).

The syntax of the search language on unstructured data is described in section 20. However, the search options available depend on the capacities natively provided by the data source. Section 20.3 provides exact details about the search capacities supported for Google Enterprise sources and Aracne sources.

Custom-type wrappers can also specify the search language capacities that are supported through Operator Configuration Properties. This way, other external indexes besides Aracne and Google Enterprise ones can be imported in DataPort. This section describes these properties.

• **Supports And.** This takes the value true, if searches with the logic operator AND are supported, and the value false, if they are not.

• **Supports OR.** This takes the value true, if searches with the logic operator OR are supported, and the value false, if they are not.

• **Supports Not.** This takes the value true if searches with the logic operator NOT are supported, and the value false, if they are not.

• **Supports Exact Search.** This takes the value true if searches by exact phrase are supported, and the value false, if they are not.

• **Supports One Wildcards First Position.** This takes the value true, if the wildcard matches with just one character (i.e. the wildcard '?' in the first position of a term are supported.)
• **Supports One Wildcards Rest Position.** This takes the value `true` if the wildcard matches with just one character (i.e. the wildcard '?') in the remaining positions of a term other than the first are supported.

• **Supports Multi Wildcards First Position.** This takes the value `true` if wildcards that match with multiple characters (i.e. the wildcard '*') are supported in the first position of a term.

• **Supports Multi Wildcards Rest Position.** This takes the value `true`, if wildcards that match with multiple characters (i.e. the wildcard '*') are supported in the remaining positions of a term other than the first.

• **Supports Fuzzy Terms Without Minimum Relevance.** This takes the value `true` if fuzzy searches without specifying a minimum similarity threshold are supported.

• **Supports Fuzzy Terms With Minimum Relevance.** This takes the value `true`, if fuzzy searches specifying a minimum similarity threshold are supported.

• **Supports Proximity Terms Without Maximum Distance.** This takes the value `true`, if searches by proximity without specifying a maximum distance among the terms are supported.

• **Supports Proximity Terms With Maximum Distance.** This takes the value `true`, if searches by proximity specifying a maximum distance among the terms are supported.

• **Supports Boosting Terms Without Boosting Factor.** This takes the value `true`, if the relevance boosting specification is supported for a term without specifying a specific boosting factor.

• **Supports Boosting Terms With Boosting Factor.** This takes the value true, if the relevance boosting specification is supported for a term specifying a specific boosting factor.

• **Supports Inclusive Range Search.** This takes the value `true`, if range searches are supported (inclusive).

• **Supports Exclusive Range Search.** This takes the value `true`, if range searches are supported (exclusive).

• **Supports Field Grouping.** This takes the value `true`, if the combination of logic operators AND and OR is supported using brackets. For example:

  \[ \text{title contains ' (term1 AND term2) OR (term3)' } \]

• **Supports Grouping.** This takes the value true, if the combination of logic operators AND and OR in different query conditions is supported. For example:

  \[ \text{title contains 'term1' AND (content contains 'term2' OR summary contains 'term3') } \]
18.4 Creating Wrappers

For each kind of wrapper supported by Virtual DataPort there exists a statement for creating wrappers. The following subsections detail the manual creation process for each kind of wrapper.

**NOTE**: It is strongly recommended that the wrapper and data source creation process be undertaken graphically using the DataPort administration tool (see [ADMIN_GUIDE]).

Previously, the concepts of execution context and interpolation strings, which will be used in creating some kinds of wrappers, are introduced in section 18.4.1 while general information about the schema metadata of the results returned by wrappers will be provided in section 18.4.2.

18.4.1 Execution Context and Interpolation Strings

As already mentioned in previous sections, the mission of a wrapper is to execute queries and/or updates on data sources.

When DataPort requests a wrapper to execute a query, it uses two different ways to provide the data on the query conditions that the wrapper should execute on the source:

- As a structured list of query conditions. This is the manner used by most wrapper types.
- As a series of interpolation variables included in a run context. This form of access is used by WWW-type wrappers using versions of Denodo ITPilot prior to 4.0 (see section Figure 96) and by JDBC wrappers using a pattern SQL query (see section 18.4.6.2). Details on the use of interpolation strings can be found in section 19.5.

18.4.2 Wrapper Metadata

In the case of most wrappers it is possible to specify metadata of the output schema (OUTPUTSCHEMA) they provide, i.e. the fields that will represent the data extracted from the source. These fields can be of three types:

- **SIMPLE**: fields belonging to basic data types such as text strings, integers, etc. Optionally, you can indicate if they can appear in query conditions of the wrapper. Query fields can be mandatory (every query must include a condition for such field) or optional. When specifying a simple field, its Java datatype is also specified. For that, the conversion tables specified in section 18.1.1 must be taken into account.
- **REGISTER**: formed by one or various fields, both simple and compound.
- **ARRAY**: lists composed by register-type fields.

Furthermore, a series of restrictions can be indicated for each output schema field:

- Whether the field can include null values (**NULL**) or cannot (**NOT NULL**). The **NULL** value is assumed by default.
- Whether the results can be ordered by the (**SORTABLE**) field or not (**NOT SORTABLE**). It is also possible to specify that the results can be sorted by the field but only in ascending (**SORTABLE ASC**) or descending (**SORTABLE DESC**) order. The **SORTABLE** value is assumed by default.
- Whether the field can be updated in an **UPDATE** statement (**UPDATEABLE**) or cannot (**NOT UPDATEABLE**). The **UPDATEABLE** value is assumed by default.
18.4.3 JDBC Wrappers

A JDBC wrapper extracts data from a remote Database via JDBC. The syntax for creating a wrapper of this type is shown in Figure 85.

```
CREATE [ OR REPLACE ] WRAPPER JDBC <name:identifier> 
  DATASOURCENAME=<name:identifier>
  { 
    [ SCHEMANAME = <name:literal> ] RELATIONNAME = <name:literal> 
    | SQLSENCE = <literal> 
  } 
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ] 
  [ ALIASES ( <alias> [, <alias>]* ) ] 
  [ SOURCECONFIGURATION ( [ <source configuration property> 
    [, <source configuration property> ]* ] ) ]

<field> ::= 
  <name:identifier> [ = <mapping:literal> ] : <type:literal> 
  [ ( { OBL | OPT } ) ] [ <inline constraints> ]* 
  | <name:identifier> [ = <mapping:literal> ] : ARRAY OF (<register field> ) 
    [ <inline constraints> ]* 
  | <name:name:register field>

<register field> ::= 
  <name:identifier> [ = <mapping:literal> ] : 
    REGISTER OF ( <field> [, <field> ]* ) [ <inline constraints> ]* 

<inline constraint> ::= 
  [ NOT ] NULL 
  | [ NOT ] UPDATEABLE 
  | { SORTABLE [ ASC | DESC ] | NOT SORTABLE } 
  | EXTERN 
  | MAXLEN = <max. length of the field:integer> (only for JDBC wrappers 
    obtaining data from Oracle PL/SQL. See below)

<source configuration property> ::= 
  ALLOWDELETE = { true | false | DEFAULT } 
  | ALLOWINSERT = { true | false | DEFAULT } 
  | ALLOWUPDATE = { true | false | DEFAULT } 
  | DATAINORDERFIELDSLIST = { ( <name:identifier> { ASC | DESC } 
    [, <name:identifier> { ASC | DESC } ]* ) | DEFAULT } 
  | SUPPORTSDISTRIBUTEDTRANSACTIONS = { true | false | DEFAULT }
```

Figure 85 Syntax of the CREATE WRAPPER JDBC statement
ALTER WRAPPER JDBC <name:identifier>
  [ DATASOURCENAME = <name:identifier>]
  [ ( SCHEMANAME = <name:literal> ] RELATIONNAME = <name:identifier> |
    SQLSENTENCE = <literal> ]
  [ OUTOUTSCHEMA ( <field> [, <field>]*) ]
  [ ALIASES ( <alias> [, <alias>]* ) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property>]* ] ) ]

<field> ::= (see CREATE WRAPPER JDBC for details)
<source configuration property> ::= (see CREATE WRAPPER JDBC for details)

Figure 86 Syntax of the ALTER WRAPPER JDBC statement

To specify a JDBC-type wrapper it is needed to indicate the name of the JDBC data source to be used (DATASOURCENAME). There are two mechanisms to indicate the wrapper where it has to retrieve the data from:

- Indicate the name of the table (RELATIONNAME) and its schema (SCHEMANAME) in the database.
- Specify a SQL statement (SQLSENTENCE). The SQL statement can be an interpolation string (see section 19.5).

The OUTPUTSCHEMA clause defines the output schema of the data that the wrapper will provide (see section 18.4.2). For each simple-type element the type must be specified. Furthermore, an association may be indicated between the name of the field returned by the wrapper and the name of the field in the database (as specified in the mapping). If this clause is not defined, the results returned by the wrapper must be compatible with the schema of the base relation the wrapper is associated with. More specifically, the names of the attributes obtained as results of the query must match those of the base relation, and their values must be compatible with their data types in relation base.

The ALIASES clause is useful when the SQLSENTENCE option and the special interpolation variable WHEREEXPRESSION (see section 18.4.3.2.1) are used.

The wrapper creation statement accepts the OR REPLACE modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (SOURCECONFIGURATION). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 85), and are explained in section 18.4.16.

18.4.3.1 Specification of a Table in the Remote Database

The first alternative for specifying the data to be obtained from the remote database is to indicate the name of the table or view in the database from which the data should be extracted.

18.4.3.2 Using a SQL Statement

The other mechanism for creating JDBC wrappers is defining a SQL statement that will be sent to the database when the wrapper is queried. We can use this mechanism to invoke stored procedures of the database or execute complex queries.
The specified SQL statement is an interpolation string susceptible to being parameterized with variables received from the execution context (see section 19.5 for details on same).

18.4.3.2.1 Using WHEREEXPRESSION

DataPort provides a predefined interpolation variable called WHEREEXPRESSION that simplifies the creation of base relations when the SQL statement method is required.

Besides, the use of WHEREEXPRESSION also has consequences regarding optimization. More specifically, if a join view uses the NESTED execution method (see section 19.2.1), and the view that acts as second relation is of the SQL statement type, it is highly advisable that such relation has been created by using WHEREEXPRESSION, because in that case Virtual DataPort can apply optimizations that are not possible with the rest of SQL statement-type base relations.

The use of WHEREEXPRESSION is explained now. WHEREEXPRESSION can be used in the SQL query specified to create the wrapper as a substitute of all or part of the WHERE query clause. At runtime, DataPort will replace the variable with a valid query condition, built from the query conditions received by the wrapper. For instance, let us suppose that a relation base named VIEW1 is created by the following SQL statement:

SELECT StorProc(FIELD1), FIELD2, FIELD3, FIELD4 ALIAS4
FROM TABLE1
WHERE @WHEREEXPRESSION

Notice that the query is using a stored procedure in the SELECT clause, being this the reason why the base relation must be created by using the SQL Statement method.

Also take into account that in the previously commented query the alias ALIAS4 defined is associated with the FIELD4 field. When the SQL Statement defines aliases, it is necessary to use the ALIASES clause (see its syntax in Figure 85) in order to adequately specify the alias utilized. In the example, the ALIAS4 attribute must be set to FIELD4.

Following the example, if once a view VIEW1 that uses the defined wrapper has been created, it is executed on DataPort with the following VQL query (NOTICE: in the example it is assumed that the user has not modified the names of the attributes when creating the base relation and, therefore, they match the ones specified in the SQL query that was used to create the wrapper):

SELECT * FROM VIEW1 WHERE FIELD2='f2' AND ALIAS4='f4'

In this case, Virtual DataPort will substitute the WHEREEXPRESSION variable at run time, by the value required to execute the equivalent query on the original database. In this case:

SELECT StorProc(FIELD1)AS ALIAS1, FIELD2, FIELD3, FIELD4 AS ALIAS4
FROM TABLE1
WHERE FIELD2='f2' AND FIELD4='f4'

18.4.3.2.2 Importing an Oracle PL/SQL Stored Procedure

To create a JDBC wrapper that invokes a stored procedure, we have to create the wrapper using an SQL statement. If we invoke a PL/SQL procedure from an Oracle database, we can define the maximum length of the returned fields.
CREATE WRAPPER JDBC pl_sql_sample
DATASOURCENAME=ds_jdbc_oracle_sample
SQLSENSETENCE='CALL sampleStoredProcedureWithTable(?)' ISPROCEDURE
OUTPUTSCHEMA ( 
  O_ID_RECORD: ARRAY OF ( 
    VALUE: REGISTER OF ( 
      VALUE:'java.lang.String' (OPT) NOT NULL
      NOT SORTABLE
      NOT UPDATEABLE
      MAXLEN=100
    ) NOT SORTABLE
    NOT UPDATEABLE
  ) NOT NULL
  NOT SORTABLE
  NOT UPDATEABLE
  MAXLEN=50
)
;

Figure 87 Example of JDBC wrapper that invokes an Oracle PL/SQL procedure

The VQL of Figure 87 creates a JDBC wrapper that invokes a PL/SQL procedure called sampleStoredProcedureWithTable. This procedure returns an element that is an array of registers. Each register has a String field. Each of these fields can have a maximum length of 100 characters (MAXLEN=100) and the array can have fifty elements at most (MAXLEN=50).

The following properties of the file $DENODO_HOME/conf/VDBConfiguration.properties define:

- com.denodo.vdb.engine.wrapper.raw.jdbc.adapter.plugins.OraclePlugin.storedProcedure.table.maxlen: default value for the maximum number of registers of an array. In this example, this value would be used if MAXLEN=50 was not defined.

- com.denodo.vdb.engine.wrapper.raw.jdbc.adapter.plugins.OraclePlugin.storedProcedure.register.maxlen: default value for the maximum length of the fields of a register. In this example, this value would be used if MAXLEN=100 was not defined.

18.4.4 Multidimensional Databases Wrappers

A wrapper for a multidimensional database connects to a multidimensional DB, through a multidimensional DB data source, execute a query and return the results.

As we mentioned in section 18.3.3, although the Administration Tool only lists ‘Multidimensional DB’ sources, the Server distinguishes between two types of multidimensional data sources: SAP data sources and generic ones (OLAP [OLAP4J]) such as Mondrian. It does the same with multidimensional wrappers.

Figure 88 and Figure 89 contain the syntax of the commands to create and modify SAP multidimensional DB wrappers.

CREATE [ OR REPLACE ] WRAPPER SAPBW <name:identifier>
DATASOURCENAME = <name:identifier>
MDXSENSETENCE = <name:literal>
[ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 88 Syntax of the CREATE WRAPPER SAPBW statement
ALTER WRAPPER SAPBW <name:identifier>  
  [ DATASOURCENAME = <name:identifier> ]  
  MDXSENTENCE = <name:literal>  
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 89 Syntax of the ALTER WRAPPER SAPBW statement

Figure 90 and Figure 91 contain the syntax of the commands to create and modify OLAP multidimensional DB wrappers.

CREATE [ OR REPLACE ] WRAPPER OLAP <name:identifier>  
  DATASOURCENAME = <name:identifier>  
  MDXSENTENCE = <name:literal>  
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 90 Syntax of the CREATE WRAPPER OLAP statement

ALTER WRAPPER OLAP <name:identifier>  
  [ DATASOURCENAME = <name:identifier> ]  
  MDXSENTENCE = <name:literal>  
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 91 Syntax of the ALTER WRAPPER OLAP statement

The appendix ‘Multidimensional to Relational Mapping’ of the Administration Guide [ADMIN_GUIDE] explains how the results of MDX queries are mapped to a relational structure.

18.4.5 ODBC Wrappers

ODBC wrappers allow querying ODBC data sources.

Figure 92 shows the syntax of creation of an ODBC wrapper. It follows the same structure as that defined for a JDBC wrapper. To create a wrapper of this type it is necessary to specify the ODBC data source, the table or SQL statement used to obtain the data from the source, and the output schema provided by the wrapper. For more information, see section 18.4.3.

CREATE [ OR REPLACE ] WRAPPER ODBC <name:identifier>  
  DATASOURCENAME=<name:identifier>  
  ( RELATIONNAME=<name:literal> | SQLSENTENCE=<literal> )  
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]  
  [ ALIASES ( <alias> [, <alias>]* ) ]  
  [ SOURCECONFIGURATION ( [ <source configuration property> [, <source configuration property> ]* ] ) ]
  <field> ::=  
    <name:identifier> [ = <mapping:literal> ] : <type:literal>  
      [ ( { OBL | OPT } ) ] [ <inline constraints> ]*  
    | <name:identifier> [ = <mapping:literal> ] : ARRAY OF <register field>  
      [ <inline constraints> ]*  
    | <name:register field>
The wrapper creation statement accepts the **OR REPLACE** modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (**SOURCECONFIGURATION**). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 92), and are explained in section 18.4.16.

**Figure 92** Syntax of the CREATE WRAPPER ODBC statement

| <field> ::= (see CREATE WRAPPER ODBC for details) |
| <source configuration property> ::= (see CREATE WRAPPER ODBC for details) |

**Figure 93** Syntax of the ALTER WRAPPER ODBC statement

### 18.4.6 WWW Wrappers

WWW wrappers are used to import semi-structured data sources (typically semi-structured web sources). These sources may be accessible in the web, through the local file system or through a FTP service. This kind of wrappers require Denodo ITPIlot [ITPILOT] to execute (ITPIlot also allows graphical creation of these wrappers).

It is important to note that the DataPort administrator does not have to create VQL statements to import these wrappers manually. ITPIlot includes options to automatically generate the necessary VQL for these tasks. The use of statements generated automatically by ITPIlot is strongly recommended.

The syntax for creating WWW wrappers is shown in Figure 94.
CREATE [ OR REPLACE ] WRAPPER ITP <name:identifier>
  [ MAINTENANCE { TRUE | FALSE } ]
  ([ OUTPUTSCHEMA ( <field> [, <field>])* ] SEQUENCE ( <sequence clause> )
  [ <substitution_clause> ]*
  | <scriptcode:literal> <xmlcontent:literal>)
  [ SOURCECONFIGURATION ( [ <source configuration property>
      [, <source configuration property> ]* ] ) ]
  | <name:register field>
  [ <register field> ::= <name:identifier>:REGISTER OF ( <field> [, <field>] )* ]
  [ <inline constraints> ]*
  | <name:identifier>:ARRAY OF ( <register field> ) [ <inline constraints> ]*
  | <name:register field>

<sequence clause> ::= CONNECTIONNAME=<connection class name:literal>
  CREATENEWINSTANCE=<boolean>
  ADD ROUTE <route>

<route> ::= LOCAL <connection class name:literal> <specification:literal>
  <uri:literal>
  | HTTP <connection class name:literal> <specification:literal>
    ( GET | POST ) <uri:literal>
  | FTP <connection class name:literal> <specification:literal>
    <uri:literal> <login:literal> <pwd:literal>

<substitution_clause> ::= ADD SUBSTITUTION <precondition_1> [,<precondition_i>]*
  ( <sequence_clause> )

<inline constraint> ::= [ NOT ] NULL
  | [ NOT ] UPDATEABLE
  | { SORTABLE [ ASC | DESC ] | NOT SORTABLE }

CREATE [ OR REPLACE ] WRAPPER ITP <name:identifier>
  [ MAINTENANCE { TRUE | FALSE } ]
  [ REGENERATE {TRUE | FALSE} ]
  [ AUTODEPLOY { TRUE | FALSE } ]
  [ JSCRIPT ] <script code:literal>
  [ [ MODEL ] <model xml:literal> ]
  [ [ SCANNERS ( <scanner name:literal> [, <scanner name:literal> ]* ) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
      [, <source configuration property> ]* ] ) ]
  [ DATAINORDERFIELDSLIST = { DEFAULT | ( <name:identifier> { ASC | DESC } ]
  [ <source configuration property> ::= ]

Generating Wrappers and Data Sources 135
The syntax for modifying WWW wrappers is similar (Figure 95).

```
ALTER WRAPPER ITP <name:identifier>
  [ MAINTENANCE { TRUE | FALSE } ]
  [ [ OUTPUTSCHEMA ( <field> [, <field>]* ) ] SEQUENCE ( <sequence clause> )
  [ <substitution_clause> ]*)
  | <scriptcode:literal> <xmlcontent:literal> ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property> )* ] ) ]

<field> ::= (see CREATE WRAPPER ITP for details)
<sequence clause> ::= (see CREATE WRAPPER ITP for details)
<substitution_clause> ::= (see CREATE WRAPPER ITP for details)
```

Figure 95 Syntax of the ALTER WRAPPER ITP statement

There are two alternative ways of creating a WWW wrapper, depending on whether the version of ITPilot used is before or after version 4.0. Section 18.4.6.1 deals with wrappers created using ITPilot 4.0 or after and section Figure 96 deals with wrappers created with previous versions (NOTE: wrappers created with ITPilot versions previous to 4.0 are considered obsolete and should not be used in new projects). The options common to both cases are described below.

The MAINTENANCE clause allows enabling or disabling the ITPilot automatic maintenance system for the wrapper. See the ITPilot documentation [ITPILOT] for further details.

The wrapper creation statement accepts the OR REPLACE modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (SOURCECONFIGURATION). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 94), and are explained in section 18.4.16.

18.4.6.1 WWW (ITPilot) Wrappers with ITPilot 4.0 and after

As of version 4.0, wrappers created using Denodo ITPilot are modeled as component flows compiled to JavaScript language. In this case, the wrappers will be created specifying the JavaScript code generated by ITPilot for the wrapper (scriptcode:literal in the syntax) and the description of the component flow forming the
wrapper, also generated by ITPilot (<xmlcontent:literal> in the syntax). Figure 96 shows an example (not the full JavaScript code or the full description of the component flow):

```
CREATE WRAPPER ITP AcmeWrapper
    MAINTENANCE FALSE
    "function getInit() {
        ... (rest of Javascript code)"
    "<?xml version='1.0' encoding='ISO-8859-1' ?>
    <InitComponent className='com.denodo.itp.model.components
        ... (rest of flow description)"
```

Figure 96 Example of ITPilot 4.0 wrapper

18.4.6.2 ITPilot Wrappers with Versions of ITPilot Prior to 4.0

**NOTE:** wrappers created with ITPilot versions previous to 4.0 are considered obsolete and should not be used in new projects.

In versions prior to ITPilot 4.0, wrapper creation requires the specifying of an access sequence. An access sequence represents a series of paths (routes to pages) where the system will search for the results to be extracted from the source consecutively and in order.

The access paths to resources from which data are extracted are specified through interpolation strings (see section 19.5).

An access sequence contains the following data:

- **CONNECTIONNAME:** Java class used to make the connection. A connection is created using a character string comprised of two parts: the connection name and the start parameters of same (optional). Both elements should be separated by commas. The class specified here acts as a default class for those wrapper paths that do not explicitly specify its connection class. `http.HTTPClientConnection` will be the default class used.

  Virtual DataPort includes various connection classes for the various available path types. The available connection classes are shown in the description of the syntax of each path type (see section 18.2).

- **CREATENEWINSTANCE:** If it is necessary to create a new connection for each request or an attempt should be made to reuse existing connections (this parameter is only taken into consideration in the paths that do not specify their own connection class).

- The list of paths that must be accessed to obtain the data from the external source. Paths are specified as seen in section 18.2, adding a data extraction specification able to extract the desired data from the page accessed through the defined path (specification should be written using the ITPilot DEXTL data extraction language [ITPILOT]). In addition, access patterns can be parameterized using context variables and interpolation functions (see section 19.5).

Another important consideration when building the wrapper is that the results returned by the query made should be compatible with the schema of the base relation to which said wrapper is linked in Virtual DataPort. More specifically, the attribute names obtained as a result of the data extraction should coincide with those of the base relation and their values should also be compatible with the data types in the base relation.
The metadata can also specify a regular expression in the simple-type fields which the results should match (those tuples in which the value for a field does not match its linked regular expression will be ruled out). In the case of WWW wrappers with versions prior to ITPilot 4.0, fields of the simple-type are all textual.

It is also possible to add an alias list for each wrapper field. These aliases can be used by ITPilot for automatic wrapper maintenance tasks (see [ITPILOT] for more information). Additionally, both in the creation statement and the ITPilot wrapper modification statement it is possible to indicate whether you wish to activate automatic maintenance for the wrapper.

18.4.6.3 Substitutions

A WWW wrapper used in versions prior to ITPilot 4.0 can be configured to use different access sequences, depending on the query conditions that Virtual DataPort includes in the query.

To achieve this, the administrator can specify a set of substitutions. A substitution defines:
- A list of preconditions on the attributes included in the query. A precondition represents a requirement which the query conditions must satisfy.
- A sequence, which will be executed if every substitution precondition is accomplished.

If the query conditions do not verify the preconditions of any substitution, the source will be accessed through the default sequence.

The format of the precondition list is comprised of a list of strings, where each one of them represents the name of a variable from the wrapper execution context. The condition of a substitution is verified if the referenced variable exists in the execution context (see section 19.5). The preconditions are specified as \texttt{<attribute>#$<operator>}. For example, let us suppose that a specific access sequence is to be used whenever a query against the wrapper contains a condition on the \texttt{TITLE} attribute and with the operator \texttt{containsor} (that is, "TITLE containsor 'values'"): to achieve it, a substitution would be created with a precondition "TITLE#containsor".

Figure 97 shows a WWW wrapper with a default sequence which uses an HTTP route, with a pattern access called \texttt{ACCESSPAT1} (compliant with any of the formats supported by ITPilot [ITPILOT]) and a data extraction specification \texttt{DATAEXTRACTSPEC1} (written in the ITPilot data extraction language, known as DEXTL [ITPILOT]).

Besides, a substitution is included, which is used in case the source is queried with the operator \texttt{containsor} over the \texttt{TITLE} attribute. Another sequence would be used, which consists in an HTTP route with \texttt{ACCESSPAT2} access pattern and an extraction specification called \texttt{DATAEXTRACTSPEC2}.

```
CREATE WRAPPER ITP shopview
  SEQUENCE (
    CONNECTIONNAME='http.HTTPClientConnection,120000'
    CREATENEWINSTANCE=TRUE
    ADD ROUTE HTTP '' 'DATAEXTRACTSPEC1' POST 'ACCESSPAT1'
  )
  ADD SUBSTITUTION 'TITLE#containsor' (
    CONNECTIONNAME='http.HTTPClientConnection,120000'
    CREATENEWINSTANCE=TRUE
    ADD ROUTE HTTP '' 'DATAEXTRACTSPEC2' POST 'ACCESSPAT2'
  )
```

Figure 97 Creation of a WWW wrapper
### 18.4.7 Web Services Wrappers

Virtual DataPort supports the creation of wrappers for SOAP Web services. Through the data contained in a WSDL specification file of a Web service (which was indicated when creating the Web service data source) the wrapper should select a specific operation to be modeled as a base relation, defining how the different parameters required for execution of the operation are established and which output data will form part of the wrapper result.

Figure 98 shows the syntax of the VQL statement for creating a Web services wrapper.

```sql
CREATE [ OR REPLACE ] WRAPPER WS <name:identifier>  
  DATASOURCENAME=<name:identifier>  
  SERVICENAME=<literal>  
  PORTNAME=<literal>  
  OPERATIONNAME=<literal>  
  [ INPUTMESSAGE=<literal> OUTPUTMESSAGE=<literal> ]  
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]  
  [ SOURCECONFIGURATION ( [ <source configuration property> 
    [, <source configuration property>] ]* ) ]  

<field> ::=  
  <name:identifier> = <mapping:literal> [ VALUE <literal> ]  
  [ ( { OBL | OPT } ) ] [ <inline constraints> ]*  
  | <name:identifier> = <mapping:literal> : ARRAY OF ( <register field> )  
    [ <inline constraints>]*  
  | <name:register field>  

<register field> ::=  
  <name:identifier> = <mapping:literal> :  
    REGISTER OF ( [ <field> [, <field>] ] ) [ <inline constraints> ]*  

<inline constraint> ::=  
  [ NOT ] NULL  
  | [ NOT ] UPDATEABLE  
  | ( SORTABLE [ ASC | DESC ] | NOT SORTABLE )  

<source configuration property> ::=  
  DATAINORDERFIELDSLIST = { DEFAULT | ( <name:identifier> { ASC | DESC }  
    [, <name:identifier> { ASC | DESC }] )* }  
  | DELEGATEOPERATORSLIST = { DEFAULT | ( <operator:identifier>  
    [, <operator:identifier>] )* }  
```

**Figure 98** Syntax of the CREATE WRAPPER WS statement

The modification syntax of a Web service wrapper is similar and is shown in Figure 99.
ALTER WRAPPER WS <name:identifier>
    [ DATASOURCENAME = <name:identifier> ]
    [ SERVICENAME = <name:literal> ]
    [ PORTNAME = <name:literal> ]
    [ OPERATIONNAME = <operation:literal> ]
    [ INPUTMESSAGE = <input:literal> OUTPUTMESSAGE = <output:literal> ]
    [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
    [ SOURCECONFIGURATION ( [ <source configuration property>
        [, <source configuration property>]* ] ) ]

<field> ::= (see CREATE WRAPPER WS for details)
<source configuration property> ::= (see CREATE WRAPPER WS for details)

Figure 99 Syntax of the ALTER WRAPPER WS statement

In addition to the Web service data source name that identifies the WSDL definition file, it is necessary to indicate other parameters that define the Web service operation to be used by the wrapper:

- **SERVICENAME**: Name of the Web service on which the operation is to be invoked. A WSDL file can contain the definition of various Web services.
- **PORTNAME**: Name of the port containing the specific operation.
- **OPERATIONNAME**: name of the operation. There may be several different operations with the same name, which are distinguished because of the input/output messages they allow. These are indicated in the following parameters.
- **INPUTMESSAGE**: Name of the message that defines the input parameters of the operation of the search method to be modeled (optional).
- **OUTPUTMESSAGE**: Name of the message that defines the output parameters of the operation of the search method to be invoked (optional).

The attributes of the messages of the selected operation define the Web services wrapper schema, i.e. a Web service wrapper has as a schema the input, output and input-output attributes with the names defined in the WSDL file.

**NOTE**: Operations can also use compound parameters in the input message. These parameters will be converted to DataPort compound types (see section 19.1) in the same way as those of the output message and you may specify conditions on them using the compound value constructors `ROW` and `'( ')` (see section 5.3.1).

From the list of conditions received the wrapper will create the parameters required to invoke the Web service and obtain the required results.

As with the other wrappers, it is possible to explicitly indicate the output schema of the wrapper (OUTPUTSCHEMA) together with the associations between the external attributes and the parameters of the Web service. The attribute "name" of a field of the OUTPUTSCHEMA indicates the name with which the wrapper will export the element. The "mapping" attribute indicates the name used by the Web service. To reference the different elements of a Web service in the mappings to be made the following notation is used:

- $<parameterNumber> \rightarrow \text{references the parameter of the indicated position of the Web service operation.}
- $$ \rightarrow \text{references the output parameter returned through invocation of the Web service operation.}

This is the notation used for the elements of the first level (input and output parameters and output of the Web service). For the other elements (fields of a result object or of a Web Service parameter) the mapping is obtained from the name of the property in the corresponding object.
The wrapper creation statement accepts the `OR REPLACE` modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (`SOURCECONFIGURATION`). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 98), and are explained in section 18.4.16.

### 18.4.8 XML Wrappers

Virtual DataPort supports the creation of wrappers from XML data sources. Figure 100 shows the syntax for creating an XML wrapper.

```sql
CREATE [ OR REPLACE ] WRAPPER XML <name:identifier>
   DATASOURCENAME=<name:identifier>
   [ TUPLEROOT <xmlnodeorpath:literal> ]
   [ OUTPUTSCHEMA ( <field> [, <field>]*) ]
   [ SOURCECONFIGURATION ( [ <source configuration property>
                             [ , <source configuration property> ]* ] ) ]
<field> ::=<name:identifier> [ = <mapping:literal> ] [: <type:literal>]
         [ ( [ OBL | OPT ] ) [ EXTERN ] ]
         [ [ ( <value:literal> [, <value:literal>] )* ] ]
         [ <inline constraints>* ]
         | <name:identifier> [ = <mapping:literal> ] : ARRAY OF ( <register field> )
         [ <inline constraints>* ]
         | <name:register field>
<register field> ::=<name:identifier> [ = <mapping:literal> ] :
               REGISTER OF ( <field> [, <field>] )* [ <inline constraints>* ]
<inline constraint> ::=[ NOT | NULL
                      | [ NOT ] UPDATEABLE
                      | [ SORTABLE [ ASC | DESC ] | NOT SORTABLE ]
<source configuration property> ::=DATAINORDERFIELDSLIST = { DEFAULT | ( <name:identifier> { ASC | DESC }
                   [ , <name:identifier> { ASC | DESC } ]* )
```
An XML wrapper is defined through an XML data source that identifies a local or remote XML resource.

The XML wrapper analyzes the structure of the XML document and returns as attributes the XML tags of the first level (using its name as attribute name), encapsulating the other elements in compound types.

Optionally, it is possible to indicate an XPath route [XPath] to an XML document node using the TUPLEROOT parameter. This is useful for DataPort to access only a portion of the document instead of the entire document. In this case, the node indicated by the path will be considered the root node for extraction. Each subelement of the indicated node will be considered a field in the tuples extracted. For example, if we import an RSS document and want the wrapper to return a tuple for each item element, the path /rss/channel/item may be used. Although an equivalent effect is possible by accessing the full XML document and subsequently using projection and flattening operations (see section 5.1.2) to get the required data, specifying the XPath route at the time of creation of the base relation will make the query process more efficient.

As with the other wrappers, the output schema of the data provided by the wrapper can be specified. This way it is possible to select only the elements of interest from the XML document to change their name (mapping represents the new name used in the wrapper; name is the original name in the XML document).

The wrapper creation statement accepts the OR REPLACE modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified [SOURCECONFIGURATION]. DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 100), and are explained in section 18.4.16.

### 18.4.9 JSON Wrappers

Virtual DataPort supports the creation of wrappers on documents in JSON format. To create a wrapper of this type the name of the data source must be indicated [DATASOURCENAME parameter].

The JSON wrapper analyzes the structure of the document and returns the JSON tags of the first level as attributes (using their name as the attribute name), encapsulating the other elements in compound types. As with the other wrappers, the schema returned by the wrapper may be specified [OUTPUTSCHEMA].

The wrapper creation statement also accepts the OR REPLACE modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified [SOURCECONFIGURATION]. DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 102), and are explained in section 18.4.16.

The following figure shows the creation syntax of a JSON wrapper.
CREATE [ OR REPLACE ] WRAPPER JSON <name:identifier>
  DATASOURCENAME=<name:identifier>
  [ TUPLOEROOT <jsonpath:literal> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property> ]* ] ) ]

<field> ::=<name:identifier> [ = <mapping:literal> ] [: <type:literal>]
  [ ( { OBL | OPT } ) [ EXTERN ] ]
  [ ( [ <value:literal> [, <value:literal> ]* ] ) ]
  [ <inline constraints> ]*
| <name:identifier> [ = <mapping:literal> ] : ARRAY OF ( <register field> )
  [ <inline constraints> ]*
| <name:register field>

<register field> ::=<name:identifier> [ = <mapping:literal> ] :
  REGISTER OF ( <field> [, <field> ]* ) [ <inline constraints> ]*

<inline constraint> ::=[ NOT ] NULL
  | [ NOT ] UPDATEABLE
  | ( SORTABLE [ ASC | DESC ] | NOT SORTABLE )

<source configuration property> ::=<name:identifier> [ ASC | DESC ]
  [, <name:identifier> ASC | DESC ]]*

Figure 102 Syntax for creating a JSON wrapper

The syntax of the modification statement of a JSON wrapper is similar.

ALTER WRAPPER JSON <name:identifier>
  [ DATASOURCENAME=<name:identifier> ]
  [ TUPLOEROOT <jsonpath:literal> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property> ]* ] ) ]

<field> ::= (see CREATE WRAPPER JSON for details)
<source configuration property> ::= (see CREATE WRAPPER JSON for details)

Figure 103 Syntax for modifying a JSON wrapper

18.4.10 DF wrappers

Virtual DataPort supports the creation of wrappers for CSV-delimited files and other flat text files with data that can
be extracted by applying regular expressions. To create a wrapper of this type the name of the data source must be
indicated (DATASOURCENAME). Optionally, as with the other wrappers, the schema of data returned by the
wrapper may be specified (OUTPUTSCHEMA).
The `wrapper` creation statement accepts the `OR REPLACE` modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (SOURCECONFIGURATION). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 104), and are explained in section 18.4.16.

**NOTE:** In this type of wrappers, registers or arrays are not supported as elements of the output schema.

The following figure shows the creation syntax of a wrapper of delimited files.

```
CREATE [ OR REPLACE ] WRAPPER DF <name:identifier>
  DATASOURCENAME=<name:identifier>
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ SOURCECONFIGURATION ( [ [ <source configuration property>
      [, <source configuration property> ]* ] ) ]
<field> ::=<name:identifier> [ = <mapping:literal> ]
  [ ( [ OBL | OPT ] ) [ EXTERN ] ]
  [ ( [ <value:literal> [, <value:literal> ]* ) ]
  [ <inline constraints> ]*]
  | <name:register field>
<register field> ::=<name:identifier> [ = <mapping:literal> ] :
  REGISTER OF ( <field> [, <field> ]* ) [ <inline constraints> ]*
<inline constraint> ::=[ NOT ] NULL
  | [ NOT ] UPDATEABLE
  | { SORTABLE [ ASC | DESC ] | NOT SORTABLE }
<source configuration property> ::=DATAINORDERFIELDSLIST = { DEFAULT | ( <name:identifier> { ASC | DESC } [, <name:identifier> ( ASC | DESC ) ]* )
```

*Figure 104* Syntax of the CREATE WRAPPER DF statement

The syntax of the modification statement of a delimited file wrapper is similar.

```
ALTER WRAPPER DF <name:identifier>
  DATASOURCENAME=<name:identifier> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ SOURCECONFIGURATION ( [ [ <source configuration property>
      [, <source configuration property> ]* ] ) ]
<field> ::= (see CREATE WRAPPER DF for details)
<source configuration property> ::= (see CREATE WRAPPER DF for details)
```

*Figure 105* Syntax of the ALTER WRAPPER DF statement
18.4.11 Denodo Aracne Wrappers

Virtual DataPort supports the creation of wrappers on indexes of unstructured data created using Denodo Aracne [ARCN].

To create a wrapper of this type, the name of the data source – DATASOURCENAME – must be indicated along with the name of the Aracne index handler – HANDLERNAME – used to create the wrapper.

As with the other wrappers, it is possible to specify the schema of the data returned by the wrapper (OUTPUTSCHEMA). In this case, the schema must contain a series of fixed attributes that are always returned by Aracne index handlers. Only the name of these fixed attributes may be modified. Furthermore, the schema may also include specific attributes corresponding to other additional fields exported by the Aracne handler.

Below is a description of the fixed attributes (see [ARCN] for further details):

- TASK. Name of the Aracne task that obtained and indexed this document. This is of string type.
- PUBDATE. Document publication date. This only appears in RSS-type documents. This is of string type.
- TITLE. Title generated by Aracne for the document. This is of string type.
- ANCHORTEXT. For documents obtained by Aracne using a Web crawling process, it contains the text associated to the link used to reach to this document. This is of string type.
- SUMMARY. Summary generated by Aracne for the document. This is of string type.
- URL. In the case of documents obtained by a web crawling process, this contains the original document URL. In RSS documents, this corresponds to the link field value of the RSS item. In the case of documents obtained from a local file system, this contains the path to it. In the case of documents obtained from an e-mail server, it contains the name of the e-mail server and the name of the account to which the e-mail belongs. This is of string type.
- IDENTIFIER. Standardized URL. This is of string type.
- CONTENT. “Useful” contents of the document generated by Aracne. See the Aracne Administration Guide [ARCN] for further details. This is of string type.
- DESCRIPTION. This only appears in RSS-type documents. In this case, it takes the value of the DESCRIPTION element from the RSS document. This is of string type.
- MODIFIED. Date on which the document in the index was last modified.
- SEARCHABLECONTENT. Field added by DataPort that concatenates the contents of the main textual fields of the document (title, summary, contents, anchortext, etc.) and the specific text fields that the index may contain. This is the field on which searches are normally made.
- LEVEL. Crawling depth level at which the document was obtained. This is of string type.
- TYPE. Content type: html, pdf, rss, etc. This is of the character string type.
- TITLEXML. Title of the document in XML with information on the view structure of the contents (paragraphs). This field is used to visually represent the title and not for searches. This is of string type.
- SUMMARYXML. Summary of the document in with information (encoded in XML) about how the text was visually distributed in paragraphs. This field is used to visually represent the summary and not for searches. This is of the character string type.
- PATH. Path where the Aracne server saved a local copy of the document. This is of string type.
- SCORE. Indication of the relative relevance of the document for the query. The results of a search are normally returned in decreasing order by SCORE. This is of float type.
- MAXDOCS. Attribute added by DataPort to restrict the maximum number of results returned by a search. This is of integer type.
- CATEGORIES. This only appears in RSS-type documents that contain a CATEGORIES element. In this case, it takes the value of this element from the RSS document. This is of string type.

Denodo Aracne is also capable of automatically generating the most relevant words of a document or a field according to the TFIDF (Term Frequency Inverse Document Frequency) relevance measurement. These terms can be included in additional fields of the DataPort wrapper schema. The use of the FILTERMAINTERMS clause is related to this function. See section 18.4.11.1.
The wrapper creation statement also accepts the `OR REPLACE` modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one. The creation syntax is shown in Figure 106.

```
CREATE [ OR REPLACE ] WRAPPER ARN <name:identifier>
    DATASOURCENAME=<name:identifier>
    HANDLERNAME=<literal>
    [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
    [ FILTERMAINTERMLIST ( <literal> [, <literal>]* ) ]

<field> ::=<name:identifier> = <mapping:literal> [ VALUE <literal> ] :
    <type:literal>
        [ ( { OBL | OPT } ) ]
    | <name:identifier> = <mapping:literal> : ARRAY OF ( <register
        field> )
        [ <inline constraints>]*
    | <name:register field>

<register field> ::=<name:identifier> = <mapping:literal> :
    REGISTER OF ( [ <field> [, <field>]* ] )

<inline constraint> ::=<name:identifier>,<num_of_mainterms_integer> [, { <literal> [, <literal>]* } ],
```

**Figure 106** Creation syntax of a Denodo Aracne wrapper

The following figure shows an example of the creation of an Aracne wrapper. The wrapper fields must include the aforementioned. In these fields, for the wrapper to work correctly, the only modification possible is the change of name. In the example, the name of the `TITLE` field is changed to `DOCNAME`. In the example, a field is also added to contain the most relevant terms of the document (see section 18.4.11.1).
CREATE WRAPPER ARN aracneview3
DATASOURCENAME=aracnesearch
HANDLERNAME='default'
OUTPUTSCHEMA (
  TASK : 'java.lang.String' (OPT),
  PUBDATE : 'java.lang.String' (OPT),
  DOCNAME='TITLE' : 'java.lang.String' (OPT),
  ANCHORTEXT : 'java.lang.String' (OPT),
  SUMMARY : 'java.lang.String' (OPT),
  IDENTIFIER : 'java.lang.String' (OPT),
  URL : 'java.lang.String' (OPT),
  CONTENT : 'java.lang.String' (OPT),
  DESCRIPTION : 'java.lang.String' (OPT),
  MODIFIED : 'java.lang.String' (OPT),
  SEARCHABLECONTENT : 'java.lang.String' (OPT) EXTERN,
  LEVEL : 'java.lang.String' (OPT),
  TYPE : 'java.lang.String' (OPT),
  TITLEXML : 'java.lang.String' (OPT),
  SUMMARYXML : 'java.lang.String' (OPT),
  PATH : 'java.lang.String' (OPT),
  SCORE : 'java.lang.Float',
  MAXDOCS : 'java.lang.Integer' (OPT) EXTERN,
  SEARCHABLECONTENT_MAIN_TERM = 'SEARCHABLECONTENT_MAIN_TERM' :
  ARRAY OF ( 
    SEARCHABLECONTENT_MAIN_TERM_REG: REGISTER OF ( 
      SEARCHABLECONTENT_SCORE : 'java.lang.Integer',
      SEARCHABLECONTENT_TERM : 'java.lang.String'
    )
  )MAINTERMS (SEARCHABLECONTENT ,10,( 'usualterm1' ,
  'usualterm2') )
);

Figure 107  Example of creating a Denodo Aracne wrapper

The syntax of the wrapper modification statement is similar and is shown in Figure 108.
ALTER WRAPPER ARN <name:identifier>
DATASOURCENAME=<name:identifier>
HANDLERNAME=<literal>
[ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
[ FILTERMAINTERMLIST ( <literal> [, <literal>]* ) ]
<field> ::=<name:identifier> = <mapping:literal> [ VALUE <literal> ] :
<type:literal>
   [ ( ( OBL | OPT ) ) ]
| <name:identifier> = <mapping:literal> : ARRAY OF ( <register field> )
   [ <inline constraints>*
| <name:register field>
<register field> ::=<name:identifier> = <mapping:literal> :
   REGISTER OF ( [ <field> [, <field>]* ] )
<inline constraint> ::=MAINTERMS ( <name:identifier>,<num_of_mainterms_integer> [, { ( <literal> [, <literal>]* ) } ] )

Figure 108  Modification syntax of a Denodo Aracne wrapper

18.4.11.1 Adding Fields with the Most Relevant Terms

Denodo Aracne is capable of automatically generating the most relevant words of a document or a field according to
the TFIDF (Term Frequency Inverse Document Frequency) relevance measurement. These terms can be accessed via
additional fields in the DataPort wrapper, as described in this section.

For example, in Figure 107 a new attribute known as SEARCHABLECONTENT_MAIN_TERM is added to contain
the most relevant terms of the SEARCHABLECONTENT index field. The new attribute must be of array of
records-type (see section 19.1). Each record must contain two fields:

- The relevant term. In this example, this takes the name of the index field, adding the suffix _TERM (SEARCHABLECONTENT_TERM).
- Its position in the list of the most relevant. In this example, this takes the name of the index field, adding
  the suffix _SCORE (SEARCHABLECONTENT_SCORE). This is of integer type. The most relevant
term will take position 1.

The modifier MAINTERMS must also be used to specify the contents of the new field. To do so, the following
parameters can be specified:

- Name (Mandatory). Name of the field involved. In this example, SEARCHABLECONTENT.
- Number of main terms (Mandatory). Maximum number of relevant terms to be included for each document.
- Filter main terms words (Optional). List of “usual words” (separated by commas) that must not appear
  among the most relevant terms for this field. Where Aracne generates any of those appearing in this list
  among the most relevant terms for the attribute contents, this would be eliminated from the list of relevant
terms. It is important to note that only usual words specific to the application must be specified. The usual
words in the language used such as articles, pronouns, etc. (commonly known as “stopwords”) are already
eliminated by Denodo Aracne.

Furthermore, the Aracne wrapper creation syntax includes the FILTERMAINTERMS clause (see Figure 106). This
clause allows for a list of usual words common to all fields in the base view to be specified. Once again, you do not
have to worry about specifying usual words in the language used such as articles, pronouns, etc. (commonly known as "stopwords"), as they are already eliminated by Denodo Aracne.

18.4.12 Google Enterprise / Google Mini Wrappers

Virtual DataPort supports the creation of wrappers on search engines created using the Google Enterprise tools [GMINI].

As usual, to create a wrapper of this type the name of the data source – DATASOURCENAME - must be indicated. It is also possible to specify the following parameters:

- **SITECOLLECTIONS**: This parameter is mandatory. It specifies, within the Google Enterprise server, the collections on which to make the search. The collections are created by the Google Enterpriseserver administrator. Its name is case-sensitive. It is possible to specify several collections separated by commas. In this case, the search will be made on all of them. Where an external server is accessed, the collection to be sought can normally be obtained by examining the value of the site parameter on the invocation URLs.

- **CLIENT**: This parameter is optional. It identifies the client making the queries. The Google Enterprise server can be configured to behave in a different manner, depending on the client to have issued the query.

- **LANGUAGES**: This parameter is optional. If specified, only documents in the specified language will be returned. The language must be a value of those listed in the Google documentation [GMINILANG].

- **NUMKEYMATCH**: This parameter is optional. Google Enterprise allows the administrator to manually determine the priority of the pages. This parameter receives an integer value of between 0 and 5, where 5 is the maximum priority. If this value is established, the searches made will only return the pages having the specified priority or higher.

As with the other wrappers, the schema of data returned by the wrapper may be specified [OUTPUTSCHEMA]. In this case, the schema must include a series of fixed fields, and only their name may be modified. Each field is described below:

- **TITLE**: Title of the document. This is of string type.
- **SUMMARY**: Summary generated by Google Enterprise for the document. This is of string type.
- **URL**: Document URL. This is of string type.
- **MIMETYPE**: MIME type of the document. This is of string type.
- **RATING**: Priority assigned manually by the Google Enterprise administrator for the document. This may take values of between 0 and 5, where 5 is the maximum priority. This is of integer type.
- **MAXDOCS**: Field added by DataPort to restrict the maximum number of results returned by a search. This is of integer type.
- **METAS**: Attribute of array of records-type (see section 19.1) that contains the metatags for the document. Each record has two string-type fields to indicate the name of the metatag (metakey) and its value (metavalue).
- **CONTENT**: Contents of the document. This is the field normally used for searches. This is of string type.
- **SITE**: This allows restricting the documents returned to those belonging to a certain domain (e.g. 'acme.com'). This is of string type.
- **FILETYPE**: Extension of the document file. This is of string type.

The wrapper creation statement also accepts the **OR REPLACE** modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one. The creation syntax is shown in Figure 109.
CREATE [ OR REPLACE ] WRAPPER GS <name:identifier>
    DATASOURCENAME=<name:identifier>
    SITECOLLECTIONS ( <literal> [, <literal>*] )
    [ CLIENT=<literal> ]
    [ LANGUAGES ( <literal> [, <literal>*] ) ]
    [ NUMKEYMATCH=<integer> ]
    [ OUTPUTSCHEMA ( <field> [, <field>*] ) ]

    <field> ::= <name:identifier> = <mapping:literal> [ VALUE <literal> ] :
    <type:literal>
    [ { OBL | OPT } ]
    | <name:identifier> = <mapping:literal> : ARRAY OF ( <register field> )
    | <name:identifier> = <mapping:literal> :
        REGISTER OF ( [ <field> [, <field>*] ] )

Figure 109  Creation syntax of a Google Mini wrapper

The following figure shows an example of the creation of a Google Mini wrapper. The wrapper fields must be those specified. For the statement to work correctly, it is only possible to change the name of the output fields. In the example, the name of the TITLE field is changed to DOCNAME.

CREATE WRAPPER GS acme_com
    DATASOURCENAME=acme_com
    SITECOLLECTIONS ( 'Acme_com' )

OUTPUTSCHEMA ( DOCNAME='TITLE' : 'java.lang.String' (OPT),
    SUMMARY : 'java.lang.String',
    URL : 'java.lang.String' (OPT),
    MIMETYPE : 'java.lang.String',
    RATING : 'java.lang.Integer',
    MAXDOCS : 'java.lang.Integer' (OPT) EXTERN,
    METAS: ARRAY OF ( METAS: REGISTER OF ( METAKEY : 'java.lang.String',
        METAVALUE : 'java.lang.String' ),
    ),
    CONTENT : 'java.lang.String' (OPT) EXTERN,
    SITE : 'java.lang.String' (OPT) EXTERN,
    FILETYPE : 'java.lang.String' (OPT) EXTERN,
    LANGUAGE : 'java.lang.String' )

Figure 110  Example of creating a Google Mini wrapper

The syntax of the wrapper modification statement is similar and is shown in Figure 110.
Virtual DataPort supports the creation of wrappers for the extraction of data contained in LDAP servers. To create a wrapper of this type, it is needed to indicate the data source name encapsulating the access data to the LDAP server (DATASOURCENAME parameter).

To identify the data to extract, there are two options. You can use the OBJECTCLASSES parameter to specify the list of Object Classes in the LDAP server that the wrapper will access. With this option the requests to the server will be automatically generated by DataPort from the user queries.

Also, the wrapper can be created from an expression (LDAPEXPRESSION) which is directly delegated to the source. This expression can have interpolation variables including the predefined interpolation variable WHEREEXPRESSION (see section 18.4.3.2.1). Remember that this variable can optimize NESTED joins delegating OR conditions in a simple query to a LDAP server.

To create a base view with this option, at least one object accessible with the expression must be selected.

Optionally, the kind of search can be selected. There are two kinds of search: recursive or in the level of root node only (the root node is specified in the data source URL). Recursive search is selected by default.

Optionally, as with the other wrappers, the schema of data returned by the wrapper may be specified (OUTPUTSCHEMA).

The wrapper creation statement also accepts the OR REPLACE modifier. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified (SOURCECONFIGURATION). DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 112), and are explained in section 18.4.16.
The following figure shows the creation syntax of an LDAP wrapper.

```
CREATE [ OR REPLACE ] WRAPPER LDAP <name:identifier>
  DATASOURCENAME=<name:identifier>
  OBJECTCLASSES = <name:literal> [, <name:literal>]*
  [ LDAPINDEX = <name:literal> ]
  [ RECURSIVESEARCH = TRUE | FALSE ]
  [ OUTPUTSCHEMA ( <field> [, <field>]*) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property> ]* ] ) ]

<field> ::=<name:identifier> [ = <mapping:literal> ] [: <type:literal>]
  [ ( { OBL | OPT } ) ]
  [ ( [ <value:literal> [, <value:literal> ]* ] ) ]
  [ <inline constraints> ]*
  | <name:identifier> [ = <mapping:literal> ] : ARRAY OF ( <register field> )
  | ( <register field> )**

<register field> ::=<name:identifier> [ = <mapping:literal> ] :
  REGISTER OF ( <field> [, <field>]* )

<source configuration property> ::=<name:identifier> [ = <mapping:literal> ]
  DATAORDERFIELDSLIST = { DEFAULT | ( <name:identifier> ASC | DESC )
  [, <name:identifier> ASC | DESC ]* }]
```

**Figure 112** Syntax for creating an LDAP wrapper

The syntax of the modification statement of an LDAP wrapper is similar.

```
ALTER WRAPPER LDAP <name:identifier>
  [ DATASOURCENAME= <name:identifier> ]
  [ OBJECTCLASSES=<name:literal> [, <name:literal>]* ]
  [ LDAPINDEX = <name:literal> ]
  [ RECURSIVESEARCH = TRUE | FALSE ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ SOURCECONFIGURATION ( [ <source configuration property>
    [, <source configuration property> ]* ] ) ]

<field> ::= (see CREATE WRAPPER LDAP for details)
<source configuration property> ::= (see CREATE WRAPPER LDAP for details)
```

**Figure 113** Syntax for modifying an LDAP wrapper

### 18.4.14 BAPI Wrappers

BAPI wrappers can connect to a SAP system, using a BAPI data source, execute a BAPI and return its results.

Figure 114 and Figure 115 contain the syntax of the commands to create and modify BAPI wrappers.
CREATE [ OR REPLACE ] WRAPPER SAPERP <name:identifier>
  DATASOURCENAME = <name:identifier>
  BAPINAME = <name:literal>
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 114 Syntax of the command to create BAPI wrappers: CREATE WRAPPER SAPERP

ALTER WRAPPER SAPERP <name:identifier>
  [ DATASOURCENAME = <name:identifier> ]
  [ BAPINAME = <name:literal> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]

Figure 115 Syntax of the command to modify BAPI wrappers: ALTER WRAPPER SAPERP

### 18.4.15 CUSTOM Wrappers

Custom wrappers provide access to a source through a specific implementation. The CUSTOM wrappers are associated with a CUSTOM data source. In the creation process for this type of data sources (see section 18.3.12), a class implementing the wrappers of this type must be specified. As explained below, this class must extend com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl (see section 19.3.3).

Figure 116 shows the syntax for creating a CUSTOM-type wrapper. The only mandatory parameter received in its creation – as well as a name to identify it by – is the name of the data source from which it will be created (see section 18.3.12).

Where the data source wrappers accept configuration parameters, the PARAMETERS clause allows specifying them.

The OR REPLACE modifier is also accepted. Where specified, if there is already a wrapper with the same name, its definition is replaced by the new one.

Lastly, certain wrapper properties can be specified [SOURCECONFIGURATION]. DataPort will take them into account to determine the operations that can be made on the wrapper. The applicable properties are indicated in the corresponding statement declaration (Figure 116), and are explained in section 18.4.16.

CREATE [ OR REPLACE ] WRAPPER CUSTOM <name:identifier>
  DATASOURCENAME=<name:identifier>
  [ PARAMETERS ( <paramName:identifier>=<paramValue:literal>
               [,<paramName:identifier>=<paramValue:literal>]* ) ]

Figure 116 Syntax of the CUSTOM wrapper creation statement

Figure 117 shows an example of creating a CUSTOM wrapper. The wrapper is given the name testcustom and is associated with the CUSTOM data source known as testcustomds. The testcustomds data source wrappers receive two configuration parameters known as ENTERPRISE and YEAR. The new wrapper is configured using the values 'enterprise1' and '2006', respectively.
Virtual DataPort 4.6
Advanced VQL Guide

CREATE WRAPPER CUSTOM testcustom
DATASOURCENAME=testcustomds
PARAMETERS ( ENTERPRISE='enterprise1', YEAR='2006' ) ;

Figure 117
Example of creating a CUSTOM

The modification statement syntax of a CUSTOM wrapper is that shown in Figure 118. The options available are the same as for the creation of the wrapper.

ALTER WRAPPER CUSTOM <name:identifier>
[ DATASOURCENAME=<name:identifier> ]
[ PARAMETERS ( <paramName:identifier>=<paramValue:literal> [,<paramName:identifier>=<paramValue:literal>]* ) ]

Figure 118
Syntax of the CUSTOM wrapper update

18.4.16 Wrapper Configuration Properties

Wrapper Configuration Properties allow indicating specific characteristics of the underlying data sources such as their distributed transaction support capacity or whether inserting operations are permitted. Section 18.3.13 indicated the configuration properties of the data sources. This section describes the configurable properties in each wrapper, depending on the type of data source they have come from.

NOTE: Typically, users do not need to edit this information since DataPort automatically uses suitable configurations for most common data sources.

The properties of each wrapper can be configured in the wrapper creation statement by adding parameter/value pairs or from the Virtual DataPort administration tool (see VDP Administration Guide [ADMIN_GUIDE] for further information). The configurable properties are as follows:

- **Allow Insert** ([ALLOWINSERT]): This indicates whether the underlying data source accepts insert operations. It is applicable to relational databases (accessible via JDBC and ODBC) and CUSTOM wrappers. The possible values are:
  - Default: VDP assigns a default value depending on the source type. In the case of relational sources, the default value is "true".
  - true: The data source allows for insert operations.
  - false: The data source does not allow for insert operations.

- **Allow Delete** ([ALLOWDELETE]): This indicates whether the underlying data source accepts delete operations. It is applicable to relational databases (accessible via JDBC and ODBC) and CUSTOM wrappers. The possible values are:
  - Default: VDP assigns a default value depending on the source type. In the case of relational sources, the default value is "true".
  - true: The data source allows for delete operations.
  - false: The data source does not allow for delete operations.

- **Allow Update** ([ALLOWUPDATE]): This indicates whether the underlying data source accepts update operations. It is applicable to relational databases (accessible via JDBC and ODBC) and CUSTOM wrappers. The possible values are:
  - Default: VDP assigns a default value depending on the source type. In the case of relational sources, the default value is "true".
  - true: The data source allows for update operations.
  - false: The data source does not allow for update operations.
- **Delegate All Operators** (**DELEGATEALLOPERATORS**): This indicates whether the source allows for all operators to be delegated. Applicable to CUSTOM wrappers. The value is "false" by default.
  
  **Note:** If this property is "true", the property **DELEGATEOPERATORSLIST** will be ignored and all the operators will be delegated.

- **Delegate AND Condition** (**DELEGATEANDCONDITION**): This indicates whether the source allows for the AND condition to be delegated. The value is "true" by default for CUSTOM wrappers.

- **Delegate Array Literal** (**DELEGATEARRAYLITERAL**): This indicates whether the source allows for array-type compound constants to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Compound Field Projection** (**DELEGATECOMPOUNDFIELDPROJECTION**): This indicates whether the source allows projections on compound fields to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Left Function** (**DELEGATELEFTFUNCTION**): This indicates whether the source allows for conditions with functions on the left part to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Left Literal** (**DELEGATELEFTLITERAL**): This indicates whether the source allows for conditions with constants on the left part to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate NOT Condition** (**DELEGATENOTCONDITION**): This indicates whether the source allows the NOT condition to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate OR Condition** (**DELEGATEORCONDITION**): This indicates whether the source allows for the OR condition to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate ORDER BY** (**DELEGATEORDERBY**): This indicates whether the source allows the ORDER BY clause to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Register Literal** (**DELEGATEREGISTERLITERAL**): This indicates whether the source allows for register-type compound constants to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Right Field** (**DELEGATERIGHTFIELD**): This indicates whether the source allows for conditions with fields on the right part to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Right Function** (**DELEGATERIGHTFUNCTION**): This indicates whether the source allows for conditions with functions on the right part to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Delegate Right Literal** (**DELEGATERIGHTLITERAL**): This indicates whether the source allows for conditions with constants on the right part to be delegated. Applicable to CUSTOM wrappers. The value is "true" by default.

- **Supports Distributed Transactions** (**SUPPORTSDISTRIBUTEDTRANSACTIONS**): This indicates whether the underlying data source can take part in anXA [XA] distributed transaction. It is applicable to relational databases (accessible via JDBC and ODBC) and CUSTOM wrappers. The possible values are:
  
  - Default: VDP assigns a default value depending on the source type. In the case of relational sources, the default value is "true".
  - true: The data source meets the XA specification.
  - false: The data source does not meet the XA specification.

- **Data in Order Field List** (**DATAINORDERFIELDSLIST**): This property determines the list of fields by which the data is sorted (where applicable). Furthermore, it is needed to specify for each field whether sorting is ascending (**ASC**) or descending (**DESC**). Each field name pair with its sort criterion is separated by a comma. This property is applicable in all data sources.

- **Delegate Operators List** (**DELEGATEOPERATORSLIST**): This property determines the list of operators that can be delegated to the data source. This allows for VDP to optimize the query plan by delegating part of the processing to the native source. While VDP carries out this action automatically on relational databases, other source types do not provide this information in their metadata, despite this sometimes being possible. VDP allows indicating the list of operators that can be delegated in the Web Service ("=" by default) and CUSTOM ("=" by default) wrapper types.

**Example:** In the following example we create a DF wrapper (see section 18.3.7) indicating that the data of the file is ordered by the ‘id’ column (property **DATAINORDERFIELDSLIST**). By adding this property, a base view created over this wrapper can participate in a MERGE JOIN. Otherwise it can’t.
18.5 QUERY WRAPPER STATEMENTS

Virtual DataPort allows directly querying wrappers (without having to define base relations on them).

The general syntax of the statement to execute queries on wrappers is shown in Figure 120. It is needed to indicate the type and name of the wrapper and an optional list of conditions in the format <value>, binary operator, <value> (see general syntax of condition values in section 3.8.1). Unary operators and multivalued binary operators are not allowed.

```plaintext
QUERY WRAPPER \{ ARN \| DF \| GS \| JDBC \| JSON \| LDAP \| CUSTOM \| ODBC \| SAPBW \| SAPERP \| WS \| XML \} <name:identifier> 
[ 
   <value> <binary operator> <value> 
   [, <value> <binary operator> <value> ]* 
 ]
<value>::= (see section 3.8)
```

Figure 120 Syntax of the QUERY WRAPPER statements

The query statement syntax of a WWW wrapper is slightly different and is shown in Figure 121. Only a list of key=value pairs can be indicated separated by commas, which will be directly received by the wrapper as input parameters.

```plaintext
QUERY WRAPPER ITP <name:identifier>
[ 
   { <name:identifier> = <value:literal> 
   [, <name:identifier> = <value:literal>]* 
   },
]
```

Figure 121 Syntax of the WWW QUERY WRAPPER statements
19 ADVANCED CHARACTERISTICS

This section describes some advanced characteristics of Virtual DataPort which, although not always necessary in the most common administration tasks, are of interest in certain cases.

19.1 MANAGEMENT OF COMPOUND-TYPE VALUES

Virtual DataPort has two classes of data types: simple types and compound types. Compound types (array and register) represent hierarchical data in the DataPort base relations and views.

NOTE: In Virtual DataPort, an array-type element must be viewed as a subrelation. Actually, a DataPort array will always have a register type internally associated. Each subelement contained in the array will belong to this register data type. Hence, the fields of this register may be seen as the schema of the subrelation being modeled. It is important to bear this in mind when applying operators to subelements of a compound field.

Each attribute value of a view in the server can be uniquely identified within a tuple using an expression called URI. The URI associated with the value of an attribute belonging to a simple type simply consists of the name of the attribute. On the other hand, the value of a compound-type attribute is represented using a tree, in which the leaves are atomic values (i.e. belonging to simple data types). Two types of non-leaf nodes exist in these trees:

- **Arrays (array type):** From these an arch runs to each of the nodes that represent the subelements that comprise the array (all belong to the same register data type). Each arch is tagged with the position index of the array subelement being indicated, written between the symbols "[" and "]".

- **Registers (register type):** From these an arch runs to each of the nodes that represent the subelements that comprise the register (each subelement is related to a field of the record that can belong to a different data type). Each arch is tagged with the name of the field.

Furthermore, an arch with the attribute name indicates the root of the tree.

Given this tree, a URI that identifies a node of same is obtained starting with the root and moving down the tree, concatenating (separated by the character "," except in the case of array indexes, in which it will be indicated the index value between brackets) the names of the different arches until arriving at the required node. Finally, the name of the attribute is concatenated at the beginning of the string. Furthermore, if in a URI for an array-type node no index is specified, then the URI indicates the list of values of the array.

Therefore, we can distinguish two types of URIs:

- Those that indicate a simple type or a register-type value.

- Those that indicate a list of values. These URIs correspond to DataPort array-type values and, therefore, can be seen as a subrelation, where each array element is a tuple and the schema of this tuple is defined by the register element fields associated with the array.

URIs of the first type can always be used in the SELECT clause of the queries or as group-by attributes in a GROUP-BY clause. If, in addition, a simple type value is pointed, then this URI can be used in the same manner as any other simple-type attribute in a query statement: in the clauses SELECT, WHERE, GROUP BY, etc. It is also possible to use the ROW and '{' '}' constructors (see section 5.3.1) to build compound values and use them in the right side of a condition. In this case, the operators '=' and '<>' are the only ones allowed, and the datatypes of the URIs
on the right and left side of the condition must be compatible (that is, their trees must be equal except for the arc names).

URIs of the second type may appear in the following cases:

- In conditions of the WHERE clauses. When these URIs appear on the left of a condition with a URI of the first type on the right. In this case, the conditions are evaluated, as if they were a condition on the subrelation modeled by the URI.

- In a FLATTEN VIEW used in the FROM clause. See section 5.1.2.

- Aggregation functions (see section 5.4.1) support this type of URIs.

19.1.1 Processing of Compound Types: Example

Imagine that you want to define a relation that models books with title and various authors. We could have the attributes:

- TITLE, simple type (text)

- AUTHOR, compound type. More specifically, we can have various authors and, for each author, we want to represent his/her name, surname and a list of contact addresses. As explained earlier, an array type models a subrelation, whereby it is necessary to indicate using a register type the schema of this relation. The subrelation AUTHOR thus has an associated register type with subattributes of the simple type NAME, SURNAME and other compound attributes of the array type to contain the list of contact addresses (CONTACT). CONTACT represents another subrelation, with a schema comprised of the subattributes MAIL and ADDRESS; MAIL has a simple type and ADDRESS is a register comprised of the subattributes STREET, PLACE and COUNTRY.

The tree of the type AUTHOR is shown in Figure 122. The data type to represent elements of the type AUTHOR can be created with the following statements:

```sql
CREATE TYPE address AS REGISTER OF (
    STREET:text,
    CITY:text,
    COUNTRY:text
);
CREATE TYPE contactAddress AS REGISTER OF (
    MAIL:text,
    ADDRESS:address
);
CREATE TYPE contactAddressArray AS ARRAY OF contactAddress;
CREATE TYPE author AS REGISTER OF (
    NAME:text,
    SURNAME:text,
    CONTACTADDRESS:contactAddressArray
);
CREATE TYPE authorArray AS ARRAY OF author;
```
Figure 123 shows an example of a tuple of this view and its internal representation:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>AUTHOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book1</td>
<td>Name1</td>
</tr>
<tr>
<td></td>
<td>Surname1</td>
</tr>
<tr>
<td></td>
<td>MAIL</td>
</tr>
<tr>
<td></td>
<td><a href="mailto:Author1@authors.com">Author1@authors.com</a></td>
</tr>
<tr>
<td></td>
<td>Street1</td>
</tr>
<tr>
<td></td>
<td>MAIL</td>
</tr>
<tr>
<td></td>
<td><a href="mailto:Author2@authors.com">Author2@authors.com</a></td>
</tr>
<tr>
<td></td>
<td>Street2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SURNAME</th>
<th>CONTACTADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name3</td>
<td>Surname3</td>
<td>MAIL</td>
</tr>
<tr>
<td></td>
<td><a href="mailto:Author3@authors.com">Author3@authors.com</a></td>
<td>STREET</td>
</tr>
<tr>
<td></td>
<td>Street3</td>
<td>City3</td>
</tr>
<tr>
<td></td>
<td>MAIL</td>
<td>ADDRESS</td>
</tr>
<tr>
<td></td>
<td><a href="mailto:Author4@authors.com">Author4@authors.com</a></td>
<td>STREET</td>
</tr>
<tr>
<td></td>
<td>Street4</td>
<td>City4</td>
</tr>
</tbody>
</table>

Figure 122 Trees of compound elements
The structure of the value tree is shown in Figure 124.

Now a base relation that models this relation can be created:

```
CREATE TABLE BOOK I18N es_euro {
    TITLE:text \{ SEARCH \},
    AUTHOR:authorArray
};
```

It will also be necessary to create a wrapper for the relation. Note that, as always, the schema of the data returned by the wrapper should be compatible with the schema of the relation, which in this case means that the wrapper requires that the data be returned in the form of compound values.

**NOTE:** Remember that it is strongly recommended that you use the Virtual DataPort graphical administration tool to import data sources and create base views. This way, the appropriate sentences for creating compound types, wrappers and base views will be automatically created.
For example, the following figure shows part of a VQL sentence to create an ITPilot wrapper to obtain the required data. Note how the output schema defined is compatible with that of the relation:

```vql
CREATE WRAPPER ITP BOOK_sm1
OUTPUTSCHEMA {
    TITLE,
    AUTHOR:ARRAY OF
        AUTHOR:REGISTER OF {
            NAME,
            SURNAME,
            CONTACTADDRESS:ARRAY OF
                CONTACTADDRESS:REGISTER OF {
                    MAIL,
                    ADDRESS:ARRAY OF
                        ADDRESS:REGISTER OF {
                            STREET,
                            CITY,
                            COUNTRY
                        }
                    }
                }
        }
}

... Wrapper definition ...;
```

![Figure 126](image)

Creating a wrapper with compound types

Once the wrapper has been created, a search method can be defined for the `BOOK` relation (see section 4.2). In most cases, query restrictions will only be defined for URIs that indicate simple data types (this is consistent with the fact that compound-type attributes are considered as though they were subrelations). However, it is also possible to add restrictions for URIs indicating compound types (in this case, remember that the operands on the right of the conditions will be built using the constructors `ROW` and `{' '}` and that only operators `=' and `<>` may be used). The following sentence adds a possible search method (note that a restriction has been included for the compound URI `AUTHOR.CONTACTADDRESS`):

```vql
ALTER TABLE BOOK
ADD SEARCHMETHOD BOOK_SM1 (
    CONSTRAINTS {
        ADD TITLE                                    NOS ZERO ()
        ADD AUTHOR.NAME                              NOS ZERO ()
        ADD AUTHOR.SURNAME                           NOS ZERO ()
        ADD AUTHOR.CONTACTADDRESS                    NOS ZERO ()
        ADD AUTHOR.CONTACTADDRESS.MAIL               NOS ZERO ()
        ADD AUTHOR.CONTACTADDRESS.ADDRESS.STREET     NOS ZERO ()
        ADD AUTHOR.CONTACTADDRESS.ADDRESS.CITY       NOS ZERO ()
        ADD AUTHOR.CONTACTADDRESS.ADDRESS.COUNTRY    NOS ZERO ()
    }
    OUTPUTLIST (TITLE, AUTHOR)
    WRAPPER (itp book)
);
```

![Figure 127](image)

Adding a search method with compound types
**NOTICE:** In the specification of URIs of compound attributes in condition queries, and in order to avoid ambiguities between the name of the table and the attribute name, the attribute names will be specified between parentheses.

Finally, some examples of queries that could be made on the relation are shown:

1. Obtain the title and the authors’ names of all the books that contain in their title the word ‘java’.

   ```sql
   SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
   FROM BOOK
   WHERE TITLE like '%java%'
   GROUP BY TITLE;
   ```

2. Find the title and the list of contact addresses for each of the authors of the books that contain in their title the word ‘java’.

   ```sql
   SELECT TITLE, LIST((AUTHOR).CONTACTADDRESS) AS AUTHORLIST
   FROM BOOK
   WHERE TITLE like '%java%'
   GROUP BY TITLE;
   ```

3. Find the title and the first e-mail address of each of the authors of all the books that contain in their title the word ‘java’.

   ```sql
   SELECT TITLE, LIST((AUTHOR).CONTACTADDRESS[0].MAIL) AS AUTHORLIST
   FROM BOOK
   WHERE TITLE like '%java%'
   GROUP BY TITLE;
   ```

4. Find the title and the name of each of the authors of all the books that contain the word ‘java’ in their title and that have at least one author with an e-mail address that contains the word ‘.es’.

   ```sql
   SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
   FROM BOOK
   WHERE (TITLE like '%java%')
   AND ((AUTHOR).CONTACTADDRESS.MAIL like '%.es%')
   GROUP BY TITLE;
   ```

5. Find the title and the name of each of the authors of all the books that contain the word ‘java’ in their title and that have at least one author with an address in the street ‘Real’.

   ```sql
   SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
   FROM BOOK
   WHERE (TITLE like '%java%')
   AND ((AUTHOR).CONTACTADDRESS.ADDRESS.STREET like '%Real%')
   GROUP BY TITLE;
   ```

6. Find the books written by an author with a single contact address, the e-mail john@mail.com and who lives in Real street in the city of Madrid (Spain).

   ```sql
   SELECT TITLE, AUTHOR
   FROM BOOK
   WHERE (AUTHOR).CONTACTADDRESS =
   {ROW('john@mail.com'),{ROW('Real', 'Madrid', 'Spain')}}
   ```
19.2  OPTIMIZING QUERIES

This section describes different aspects of interest in relation to the optimizing of queries in Virtual DataPort.

The possible strategies for executing join operations and how to choose the most suitable strategy for a view or a query are first discussed. The options for configuring the DataPort cache for a specific view are then discussed. Finally, it is described how to configure the DataPort swapping to disk policy.

19.2.1  Optimizing Join Operations

A key aspect of query optimization in Virtual DataPort is the most appropriate choice of strategy for join operations. Although Virtual DataPort will try to use the most appropriate strategy in each case based on internal cost data, a specific execution strategy may be forced for the required join operation.

An execution strategy for a join consists of two elements: the method used to implement the join operation and the order in which the join input relations must be considered. Virtual DataPort supports the following execution methods:

- **MERGE**: This can only be executed in cases in which the input relation data are ordered by the join attributes. In this case, this strategy is often the most efficient and the one to consume least memory. In the case of the data not being ordered, the join technique may be used if the sources involved are all databases (accessed through JDBC or ODBC wrappers), as in this case DataPort can retrieve the data ordered from the original sources. If the use of this strategy is forced in a case in which it is not applicable, DataPort will produce an error message.

- **NESTED**: This run method firstly obtains the tuples from the first input relation that verify the join condition and then, for each combination of values obtained for the attributes taking part in the join, a subquery is issued to obtain the tuples corresponding to this combination of values in the second input relation. In case the second input relationship comes from a database, DataPort will optimize this process by emitting a single subquery which retrieves all required data from the second relationship. This method is often extremely efficient when the first input view is relatively small in relation to the second and the latency per query of the second source is low. On using this method, the order of the input relations is particularly important: the first relation should be the one with the smallest expected size.

**IMPORTANT NOTE:** In the case that the second input relation in the **NESTED** join imports data from a database, DataPort will optimize that process generating a single subquery that retrieves all required data of the second relation. Besides, if such relation has been created by means of the SQL Statement method (see section 18.4.3.2 or the Administration Guide [ADMIN_GUIDE]), then it is necessary to use the variable **WHEREEXPRESSION** (see section 18.4.3.2.1 or the Administration Guide [ADMIN_GUIDE]), so that DataPort can use that optimization option.

- **NESTED PARALLEL**: This run method is similar to the **NESTED** method. The difference is that the subqueries issued on the second input relation may be issued in parallel to each other. It accepts an additional parameter that specifies the maximum number of subqueries issued in parallel to each other. Notice that if the second relation is of database type, the use of **NESTED PARALLEL** is usually unnecessary and less efficient and the **NESTED** option can be used instead, since DataPort will optimize the process by generating a single subquery that retrieves all required data.

- **HASH**: This type of join is often the most efficient when the data in the input relations are not ordered and are large. It is also often the most effective when the query latency times for the data sources are high (e.g. Web sources), as this type of join minimizes the number of sub-queries made on the sources.
On creating a join-type view or on writing a query, it is possible to specify the run method required by indicating the modifiers NESTED, NESTED PARALLEL, MERGE or HASH. **Examples:**

```
FROM view1 HASH JOIN view2 ON (joinCondition)
FROM view1 MERGE LEFT OUTER JOIN view2 ON (joinCondition)
FROM view1 NESTED NATURAL INNER JOIN view2 ON (joinCondition)
FROM view1 NESTED PARALLEL JOIN 5 view2 ON (joinCondition)
```

Note how, in the last example, the maximum number of subqueries parallel to each other run using the NESTED PARALLEL method is limited to 5.

It is also possible to establish the required order of the input relations using the ORDERED modifier (this indicates that the input relations must be considered in the order specified by the join clause) or the REVERSEORDER modifier (this indicates that the input relations must be considered in the reverse order to that specified by the join clause). **Examples:**

```
FROM view1 NESTED ORDERED JOIN view2 ON (joinCondition)
FROM view1 NESTED REVERSEORDER LEFT OUTER JOIN view2 ON (joinCondition)
```

### 19.2.1.1 Dynamic Choice of Join Strategy

When a query that uses derived views in its `FROM` clause, is executed and the definition of these views involves join operations, it is possible to dynamically specify an execution strategy for each operation (which changes the strategy specified when the view was created, only for this specific query).

To dynamically choose the join strategy, the `CONTEXT` clause with the option `QUERYPLAN` must be used. It is also possible to use the `ALTER VIEW` sentence (see section 6.1) to modify the execution strategy of the joins taking part in defining a specific view. The formal syntax of the `QUERYPLAN` option can be seen in Figure 128.

```
QUERYPLAN = <query_plan>

<query plan> ::= { } |
                [ [<view name:identifier> : <view plans>] ]+

<view plans> ::= <view plan> |
                [ ( [ <view plan> ] ) ]+

<view plan> ::= <any method type> <any order type>
               | NESTED PARALLEL [nestedParallelNumber:integer] <any order type>

<any method type> ::= <method type> | ANY
<any order type>  ::= <order type> | ANY

<method type> ::= HASH | NESTED | MERGE
<order type> ::= ORDERED | REVERSEORDER
```

**Figure 128** QUERYPLAN syntax

Observe the following example. Suppose there are three base relations V1, V2 and V3. V1 is made up of attributes A and B, V2 by attributes B and C and V3 by attributes C, D and E. Now suppose that the following VQL sentences are executed:

```
CREATE VIEW V4 AS
SELECT A,B,C
FROM V1 MERGE JOIN V2 USING (B)
```
Figure 129 shows the definition tree for view V5 (this tree can be easily obtained with the help of the Virtual DataPort graphic administration tool. See [ADMIN_GUIDE]). As can be seen, there are two join operations that form part of the tree: that used on creating the intermediate view V4 (where the MERGE execution method is forced) and that used to create V5 (where the NESTED execution method is forced with V4 as first relation).

Now suppose that the following VQL query is to be executed:

```
SELECT * FROM V5 WHERE D=d
```

In this case, a different execution strategy may be desirable for the join operations comprising the V5 tree. For example, there may be very few tuples in V3 that verify the new condition D=d. Therefore, less tuples would be expected to enter the V5 creation join from V3 than from V4. Under these conditions and only for this query, it would be wise to change the order of input relations so that V3 is considered the first relation and V4 the second.

This may be done using the QUERYPLAN option of the CONTEXT clause. The name of the intermediate view used, and the preference for the execution method and order of input relations can be specified for each join operation in the tree of this query. ANY is used to indicate that the choice is to be made by DataPort.

Hence, in this example, the V5 creation join can be forced to be run in the desired order:

```
SELECT * FROM V5 WHERE D=d
CONTEXT (QUERYPLAN = V5:NESTED REVERSEORDER)
```

It is also possible to set the desired execution strategy of the join used to create V4. For example, if you wish to set this strategy to use the HASH method, allowing DataPort to choose the order of the input relations, write:

```
SELECT * FROM V5 WHERE D=d
```
As indicated above, the \texttt{QUERYPLAN} option is also available in the \texttt{ALTER VIEW} sentence to modify the execution strategies of the joins involved in defining a specific view. For example, if you want to modify the execution strategies of the joins in view \texttt{V5}, write:

\begin{verbatim}
ALTER VIEW V5 QUERYPLAN = (V5:NESTED REVERSEORDER V4:HASH ANY);
\end{verbatim}

\section*{19.2.2 Using the Cache}

The commands for modifying a base relation (\texttt{ALTER TABLE. See section 4.1}) and modifying a view (\texttt{ALTER VIEW. See section 6.1}) allow enabling the cache system (\texttt{CACHE} option) for a base relation or a derived view, respectively. In this case, the tuples obtained as a result of executing queries on the view will be materialized in the local database acting as a cache. The \texttt{ALTER DATABASE} command (see section 11.3.2) allows establishing the default configuration for the base relations and the views of a certain database.

Note that if this option is activated in a view, it can also be used to run periodic preloads of source data by simply making a query to a relation that obtains the data to be preloaded at the required intervals.

The cache system allows two different types of behavior to be configured:

- Exact query cache: In this case the system will use the cache data to answer a query only if an identical query to the current one has already been executed. This is the mode used, when the \texttt{ON} parameter is selected for the cache.

- More general query cache: (\texttt{POST} cache parameter). If this option is enabled, the system will detect if a given query can be answered on the basis of another previous query (even if this is not the same as the new query) by applying a series of post-processing operations. For example, if the results of a previous query
\begin{verbatim}
select * from view where (field1 = a)
\end{verbatim}
are in the cache and the system receives the query
\begin{verbatim}
select * from view where (field1 = a and field2 = b)
\end{verbatim}
, it would be possible to answer it taking as a basis the results of the first query and applying a post-processing operation that eliminates those tuples in which the \texttt{field2 = b} condition is not fulfilled.

Use of this option may not be appropriate if a wrapper does not always return all the results of a query made to a specific source. For example, if a wrapper that accesses a Web source returns only the first 100 results returned by the source for the
\begin{verbatim}
select * from view where (field1 = a)
\end{verbatim}
query, then the result of applying the post-processing condition (\texttt{field2 = b}) to the results of the query can be different to the result obtained executing directly on the source
\begin{verbatim}
select * from view where (field1 = a and field2 = b).
\end{verbatim}

In case caching is not desired in a base relation, the \texttt{CACHE OFF} option must be used. The cache data expiration timeout can also be modified by using the \texttt{TIMETOLIVEINCACHE} property (in seconds).

\subsection*{19.2.2.1 Cache Invalidation}

The cache of a view can be invalidated using the \texttt{ALTER VQL} sentence (see section 8.1).

There are two types of cache invalidation: “full” invalidation that removes all the cached data for a view; and the “partial” invalidation that only removes the cached data verifying a specified condition. Both invalidation types support the \texttt{CASCADE} option, which causes the operation to propagate to the views participating in the definition of the view specified in the invalidation sentence.

The next sentence would invalidate the cache for the \texttt{sampleView} view and for all the views participating in the \texttt{sampleView} definition:
ALTER VIEW sampleView
CACHE INVALIDATE CASCADE;

The next sentence is a partial invalidation. Only the tuples matching with the specified condition will be invalidated. In this case, the CASCADE option is not used and, therefore, the sentence will affect the sampleView view only:

ALTER VIEW sampleView
CACHE INVALIDATE WHERE field1 = 'value';

19.2.3 Configuring Swapping Policies

DataPort may require the automatic execution of swapping to disk operations to avoid possible memory overflow errors, while executing queries involving the processing and combination of large volumes of data.

The commands for modifying a base relation (ALTER TABLE is explained in section 4.1), modifying a view (ALTER VIEW is explained in section 6.1) and for executing a query (CONTEXT clause of the SELECT command is explained in section 5.9) specify whether Virtual DataPort is allowed to swap intermediate results to disk using the SWAP ON or SWAP OFF option. The ALTER DATABASE command (see section 11.3.2) allows establishing the default configuration for base relations and the views of a certain database.

DataPort will swap, when SWAP ON is chosen and where an intermediate result produced while the query or view is being executed exceeds a certain maximum size. This size may be indicated (in megabytes) using the SWAPSIZE option of the aforementioned commands (the default value is 50 Mb).

To avoid unnecessary access to disk operations that may slow down the execution, it is advised to disable swapping for views or queries where no memory overflow is foreseen.

It may also be wise to increase the SWAPSIZE value for a view or query. This is useful when an intermediate result may exceed the default value but, even in this case, the system is known to have enough memory so as not to overflow. As a general rule, the SWAPSIZE value should be no greater than one third the memory available for the JAVA virtual machine on which the DataPort server is run.

Examples:

1) Disabling swapping in a view:
   
   ALTER VIEW V SWAP OFF;

2) Enabling swapping in a view, establishing a SWAPSIZE of 100 Mb:
   
   ALTER VIEW V SWAP ON SWAPSIZE 100;

3) Running a query and disabling swapping:
   
   SELECT ... CONTEXT ('SWAP' = 'OFF')

4) Running a query with swapping enabled and a SWAPSIZE of 100 Mb:
   
   SELECT ... CONTEXT ('SWAP' = 'ON', 'SWAPSIZE' = '100')
19.2.4 Optimize DF Data Sources

It is possible to decrease the processing time of a delimited file by filtering its rows using the TUPLEPATTERN parameter instead of COLUMNDELIMITER.

The TUPLEPATTERN parameter is a regular expression that matches the rows that we want to obtain. The advantage over using COLUMNDELIMITER is that the lines that don’t match this regular expression are immediately discarded and will not be processed. On the other hand, by using COLUMNDELIMITER, every line of the file is parsed.

By using interpolation variables, we can establish filtering conditions on every row. I.e. a data source with the parameter:

```
TUPLEPATTERN = '([^\d\{2\}) (\d\{2\}) \d\{2\} (\d\{1,\}) \s+ (@NUMBER) \s+ (\w\{4\}\d
```

A view created over this data source will have a required field named NUMBER. Every line of the file that doesn’t match that regular expression will be immediately discarded.

19.3 PROGRAMMING 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. Please read the README in $DENODO_HOME/tools/denodo4e for more information.

19.3.1 Creation of 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 10.3). These custom functions can be used in the same way as every other function like MAX, MIN, SUM, etc.

Virtual DataPort allows the creation of 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.

It is recommended to create custom functions using Java annotations (see section 19.3.1.1); although it is also possible to use name conventions (see section 19.3.1.2).

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, then nothing in that Jar will be loaded in the server.
- 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 defining the custom function.
- Functions can have arity \( n \) but only the last parameter of the signature can be repeated \( n \) times.

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:

<table>
<thead>
<tr>
<th>Java Type</th>
<th>VDP Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Integer</td>
<td>int</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>long</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>float</td>
</tr>
</tbody>
</table>
Note: The parameters of a custom functions cannot be basic types: int, long, double, etc.

Note: to use custom functions that rely on external jars, we have to:

a) Copy the required jars to the directory $DENODO_HOME/extensions/thirdparty/lib.

b) 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.

19.3.1.1 Creating Custom Functions with Annotations

A Custom function created with annotations is a JAVA class with a few annotations that indicate Virtual DataPort which methods needs to execute.

Note: To compile new custom functions, the library %DENODO_HOME%/lib/contrib/denodo-custom.jar has to be added to the classpath.

The available annotations are:

- com.denodo.common.custom.annotations.CustomElement. Class annotation. Marks a class as a custom function. The parameters of this annotation are:
  - name. Required. Name of the function.
  - type. Required. Type of the function. It can be either:
    - CustomElementType.VDPFUNCTION (condition function) or
    - CustomElementType.VDPAGGREGATEFUNCTION (aggregation function)

- com.denodo.common.custom.annotations.CustomExecutor. Method annotation. Marks a method as a function signature. This method will be executed when invoking the function with the appropriate arguments. The annotation has an optional variable syntax, in order to specify the syntax of the function signature when presenting it to the user at the Administration Tool.

- com.denodo.common.custom.annotations.CustomExecutorReturnType. Method annotation. The method will be invoked to compute the return type of a function before executing a query (see 19.3.1.4 for more details). A method with this annotation is required in the following scenarios, otherwise it is optional:
  - 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.

- com.denodo.common.custom.annotations.CustomParam. Parameter annotation. Provides a user friendly name to a parameter of a function when presenting it to the user at the Administration Tool. If this annotation is not used, the syntax of the parameter will be displayed as arg1, args2…

- com.denodo.common.custom.annotations.CustomGroup. Parameter annotation used in aggregation functions. Defines the type of a CustomGroupValue in a function, using the groupType variable. The annotation also has the optional variable name as CustomParam.
19.3.1.2 Creating Custom Functions Using Name Conventions

Although we recommend developing custom functions using annotations, 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.

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 (see 19.3.1.4 for more details):

- 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.

19.3.1.3 Compound Types

Compound types and compound values are represented in custom functions by using the following Java classes:

- `com.denodo.common.custom.elements.CustomRecordType`. Represents a register data type. It stores the type name and a set of name-type pairs where the type is a `java.lang.Class` of some of the Java classes used for simple types or a compound type (CustomRecordType or CustomArrayType).

- `com.denodo.common.custom.elements.CustomArrayType`. Represents an array data type. It stores the type name and an instance of CustomRecordType with the type of the elements of the array.

- `com.denodo.common.custom.elements.CustomRecordValue`. Represents a register data value. It stores a set of name-value pairs where the value is an instance of a simple type (`java.lang.String`, `java.lang.Integer`, etc.) or a compound value (CustomRecordValue or CustomArrayValue).


- `com.denodo.common.custom.elements.CustomGroupValue`. Represents the list of values coming from a non-aggregation field in an aggregation function.
19.3.1.4 Custom Function Return Type

Custom functions which return type depends on input values or functions returning compound types must implement an additional method in order to allow Virtual DataPort to compute the return type before executing the function. This additional method must follow a few rules:

1. When the `execute` method returns a compound type or a `java.lang.Object`, the additional method must be implemented. Otherwise, it is optional (the return type is obtained directly from the method).
2. The additional method should have the same number of parameters as the `execute` method.
3. Each parameter of the additional method must have the same type or an equivalent one, as its respective parameter in the `execute` method:
   - If the `execute` method returns a basic Java type, the additional method has to return the same basic Java class.
   - I.e. If the `execute` method returns a `String` object, the additional method has to return `java.lang.String.class`.
   - If the `execute` method returns a `CustomRecordValue` object, the additional method has to return a `CustomRecordType` object.
   - If the `execute` method returns a `CustomArrayValue` object, the additional method has to return a `CustomArrayType` object.

   See table ‘Equivalency between Java and Virtual DataPort data types’ in section 19.3.1 to know the type that these return parameters will have in VDP.

4. 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 doesn’t have name, the type will be created with a random name.

Following, there are two examples of functions implementing the additional method:

**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:

```java
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 new CustomArrayValue(arrayValues);
    }
}
```
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 130 Example of function without annotations with return type depending on the input

Aggregation Function using annotations

Implementation of a function FIRST_RECORD which output is 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 execute(CustomRecordType recordType) { 
        return recordType;
    }
}

Figure 131 Example of aggregation function using annotations

Virtual DataPort also distributes a few more examples of custom functions, located at $DENODO_HOME/samples/vdp/customFunctions

There is a README file explaining how to compile and use these example custom functions.
19.3.2 Creation of Stored Procedures

The necessary classes and interfaces for creating new stored procedures are located in the com.denodo.vdb.engine.storedprocedure package. This section describes briefly the use of its main classes. See the Javadoc documentation [JAVADOC] for further details on these classes and operations.

A stored procedure has to extend the AbstractStoredProcedure class. The following methods can be overridden:

- **public void initialize(DatabaseEnvironment environment)**. Method invoked when initializing the stored procedure. The stored procedure can optionally override this method. The object DatabaseEnvironment has methods that can be used to perform certain actions on the Virtual DataPort server. See Javadoc documentation [JAVADOC] for further details about them:
  - Execute VQL statements on the DataPort server (executeQuery, executeUpdate methods).
  - Obtain references to stored procedures in the server (lookupProcedure method) in order to execute them.
  - Obtain references to server functions (lookupFunction method) in order to execute them.
  - Create transactions (createTransaction method),
  - Add a stored procedure to the current transaction (joinTransaction method),
  - Write a message in the server log (log method),
  - Obtain the value of a server property (getDatabaseProperty method). The currently accessible properties with this method are CURRENT_USER and CURRENT_DATABASE, pointing out the current user name and database name, respectively.

- **public String getDescription()**. Return the description of the stored procedure.

- **public String getName()**. Return the name of the stored procedure.

- **void prepare()**. The stored procedure can optionally override this method to prepare the current transaction.

- **void commit()**. The stored procedure can optionally override this method to confirm the current transaction.

- **void rollback()**. The stored procedure can optionally override this method to undo the current transaction.

- **public StoredProcedureParameter[] getParameters()**. Method that must specify the input and output parameters of the stored procedure. These parameters are returned as an array of StoredProcedureParameter objects. Each StoredProcedureParameter object specifies the name, type, direction (input or output) and nullability (if accepts a NULL value or not) of a parameter. If the parameter is a compound type, an array of StoredProcedureParameter objects must be specified to describe its fields. See Javadoc documentation [JAVADOC] for more details.

- **public void doCall(Object[] inputValues) throws StoredProcedureException**. Method invoked to execute the stored procedure.
• public int getNumOfAffectedRows(). Return the number of tuples affected by the execution of the procedure.

The AbstractStoredProcedure class also provides the following methods:

• public StoredProcedureResultSet getProcedureResultSet(). Method used to obtain a StoredProcedureResultSet object associated with the current stored procedure. This object contains the results that will be returned by the stored procedure and, therefore, the implementation of the doCall method will normally require a call to getProcedureResultSet() to obtain it and add to it the required results.

• protected static java.sql.Array createArray(Collection values, int type). Method that creates an SQL array-type object. This is required when the stored procedure returns compound-type values.

• protected static java.sql.Struct createStruct(Collection values, int type). Method that creates a struct SQL-type object. This is required when the stored procedure returns compound-type values.

The Virtual DataPort distribution contains examples of stored procedures (including their source code) located in DENODO_HOME/samples/vdp/storedProcedures. The README file in this path contains instructions to compile and install these samples.

19.3.2.1 Required Libraries to Develop Stored Procedures

To develop stored procedures for Virtual DataPort, add the following .jar files to the CLASSPATH of your environment:

- %DENODO_HOME%/lib/vdp-server-core/denodo-vdp-server-base.jar
- %DENODO_HOME%/lib/vdp-server-core/denodo-vdp-server-ext.jar
- %DENODO_HOME%/lib/contrib/commons-logging.jar
- %DENODO_HOME%/lib/contrib/jta-spec.jar
- %DENODO_HOME%/lib/contrib/denodo-util.jar

**Note:** to use stored procedures that rely on external jars, we have to:

a) Copy the required jars to the directory $DENODO_HOME/extensions/thirdparty/lib

b) Or, copy the contents of the required jars into the jar that contains the stored procedure. We have to copy the contents of the required jars, not the jars themselves.

c) Or, import the external jars into DataPort (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)

19.3.3 Creation of Custom Wrappers

To create a new CUSTOM-type wrapper, two Java classes must be extended:

• com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl. This class has to be extended to define the output schema of the new wrapper and certain additional metadata.

• com.denodo.vdb.engine.wrapper.raw.my.MyAccessImpl. This class is extended to implement the actual behavior of the wrapper.
The following sections explain how to implement these classes.

DataPort includes a series of sample CUSTOM wrappers in the path $DENODO_HOME/samples/vdp/wrappersCustom. The README file in this path contains instructions on how to compile, install and use them.

### 19.3.3.1 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-base.jar
- %DENODO_HOME%/lib/vdp-server-core/denodo-vdp-server-ext.jar
- %DENODO_HOME%/lib/contrib/commons-lang.jar
- %DENODO_HOME%/lib/contrib/commons-logging.jar
- %DENODO_HOME%/lib/contrib/denodo-interpolator.jar
- %DENODO_HOME%/lib/contrib/denodo-util.jar

**Note:** to use custom wrappers that rely on external jars, we have to:

a) Copy the required jars to the directory $DENODO_HOME/extensions/thirdparty/lib

b) Or, copy the contents of the required jars into the jar that contains the custom wrapper. We have to copy the contents of the required jars, not the jars themselves.

c) Or, import the external jars into DataPort (see section Importing Extensions of the Administration Guide [ADMIN_GUIDE]) and when importing the new stored procedure, select the jar with the custom wrapper and the also the external jars (see section Importing Stored Procedures of the Administration Guide).

### 19.3.3.2 Defining the Metadata of the CUSTOM Wrapper

The abstract class com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl must be extended to define the metadata of the new CUSTOM wrapper. The following methods must be overridden (see the Javadoc documentation [JAVADOC] and the examples for more details):

- **public abstract MyAccessImpl doCreate() throws CreateWrapperException.** Method responsible for creating the wrapper class that will execute the query. The following section contains more details about this class.

- **public com.denodo.vdb.catalog.wrapper.metadata.MetaRegisterRaw getOutputSchema() throws LoadWrapperException.** This method must return the schema of the data obtained through the queries made by the wrapper. For each of the attributes contained in the response tuples the following should be indicated:
  - The data type of the attribute.
  - If the attribute can be queried in the source (that is, if the wrapper can apply selection conditions to said attribute in the source). If the attribute can be queried, it may also be obligatory. This indicates that the wrapper will only be capable of executing queries that include at least one selection condition for said attribute.

- **public List getWrapperParameters() (Optional) This method must return a list containing the wrapper configuration parameters. Each parameter is represented by an object com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperParameter. This class has two parameters, the parameter name and a boolean parameter that indicates if the
parameter is mandatory or optional. If this method is not implemented, the wrapper will have no configuration parameters.

- `public com.denodo.vdb.catalog.wrapper.SourceConfiguration getSourceConfiguration()` (Optional) This method can be overridden to specify the configuration properties of the CUSTOM data source (see section 18.3.13). The implementation of this method may invoke this method in the superclass to obtain the default configuration properties. If this method is not overridden, the wrapper will use the default configuration properties (i.e: only the equality operand ("=") is delegated).

To ease the process of developing a custom wrapper, a default implementation is provided for the class hierarchy that defines the schema of a CUSTOM wrapper (see `com.denodo.vdb.catalog.wrapper.my.metadata.MyMetaRegisterRaw` in the javadoc documentation [JAVADOC]).

### 19.3.3.3 Creating the Wrapper

Once the class that encapsulates the wrapper metadata has been defined, the class that actually defines the behavior of the wrapper must be created. This class will extend `com.denodo.vdb.engine.wrapper.raw.my.MyAccessImpl` and it will be returned by the method `doCreate` of the class `com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl` (see the Javadoc documentation [JAVADOC] for more details).

The following methods can be overridden:

- `doRun (List conditions)` (Mandatory) This method will be invoked by Virtual DataPort to execute a query on the wrapper. The conditions list is formed by objects of the type `com.denodo.vdb.engine.wrapper.condition.WrapperCondition` [see Javadoc documentation [JAVADOC]].

- `doInsert`. If the wrapper supports inserts, this method will be invoked by Virtual DataPort to execute the `INSERT` statements. Its first parameter is a list of attribute names and the second one, a list with the values to insert.

- `doUpdate`. If the wrapper supports updates, this method will be invoked by Virtual DataPort to execute an `UPDATE` statement. Its parameters are a list of the attributes to alter, a list with the new values and a list of the query conditions formed by `com.denodo.vdb.engine.wrapper.condition.WrapperCondition` objects (see Javadoc documentation [JAVADOC]).

- `doDelete`. If the wrapper supports deletions, this method will be invoked by Virtual DataPort to execute the `DELETE` statements on the wrapper. It has one parameter that is a list of query conditions formed by `com.denodo.vdb.engine.wrapper.condition.WrapperCondition` objects (see Javadoc documentation [JAVADOC]).

- `prepare`. If the wrapper supports transactions, this method will be invoked to prepare a transaction.

- `commit`. If the wrapper supports transactions, this method will be invoked to confirm a transaction.
• **rollback.** If the wrapper supports transactions, this method will be invoked to undo the changes to a transaction.

• **stop.** (mandatory) This method will be invoked to stop the execution of a wrapper.

The implementation of these methods may access the value of the wrapper configuration parameters through the `getParameters()` method.

Execution of the wrapper should provide the results in accordance with the interface `com.denodo.vdb.engine.IRawResult` [see Javadoc documentation](JAVADOC).

To add tuples to this result the wrapper will follow these steps:

• Invoke the method `createRawRow` in the object `MyAccessImpl` to create a new empty tuple (which will be a `com.denodo.vdb.engine.IRawRow` object).

• Fill in the tuple with the data obtained by the wrapper.

• Add it to the result by invoking the method `addRawRow` of the `MyAccessImpl` object.

**Important:** the results returned by a wrapper must be compatible with the schema of the base view that it is associated with.

### 19.4 CREATING NEW INTERNATIONALIZATION CONFIGURATIONS

Virtual DataPort can work with data from a group of different countries/locations. An internationalization configuration, represented by a map, exists for each of the countries/locations from which data managed by DataPort may come. Various configurable parameters exist for each of the locations contemplated. Some examples of configurable parameters are: currency, symbols used as separators into decimal numbers and into thousands for currency, date format, etc.

Although Virtual DataPort includes internationalization configurations for the most common situations, creating new configurations is a very simple process. This section describes this process in detail.

The internationalization parameters of a location can be divided into various groups. The different groups are mentioned below, and each of the parameters comprising same are described in detail:

**NOTE:** The internationalization parameters are case-insensitive. For instance, “TimeZone” and “timezone” correspond to the same key.

• **Generic parameters**

  • **language** – Indicates the language used in this location. It is a valid ISO language code. These codes contain two letters in lower case as defined in ISO-639 [LANGUAGE_ISO]. Examples: `es` (Spanish), `en` (English), `fr` (French).

  • **country** – Specifies the country associated with this location. It is a valid ISO country code. These codes contain two letters in upper case, as defined by ISO-3166 [COUNTRY_ISO]. Examples: `ES` (Spain), `ES_EURO` (Spain with EURO currency), `GB` (England), `FR` (France), `FR_EURO` (France with EURO currency), `US` (United States).
- **timeZone** – Indicates the time zone of the location (e.g. Europe/Madrid for Spain = GMT+01:00 = MET = CET).

- **Currency configuration**: Allows configuring different properties for the *money*-type values.

  - **currencyDecimalPosition** – Number of decimals for the currency in the location. For example, for the euro this value is 2.

  - **currencyDecimalSeparator** – Character used as a decimal separator in the currency. For example, the decimal separator for the euro is the comma.

  - **currencyGroupSeparator** – Group separator in the currency used for the location. For example, for the euro the group separator is the full stop.

  - **currency** – Name of the currency. Example: EURO, POUND, FRANC.

  - **moneyPattern** – Specifies the currency format. In currency formats the comma is always used as a separator for thousands and the full stop as a separator for decimal numbers. The character ‘¤’ represents the currency symbol and indicates in which place the character or characters that represent it should be positioned. Example: ###,###,###.## ¤. The patterns defined by the java.text.DecimalFormat class in the API standard Java Developer Kit are used to analyze the currencies (see Javadoc [JDKJAVADOC] documentation for more information).

- **Configuration of dates**: Configuration of type *date*.

  - **datePattern** – Indicates the format for dates. To specify the format for dates ASCII characters are used to indicate the different units of time. The following table lists the meaning of each of the reserved characters used in a date format, their arrangement and an example of use. Example of a date format: `d-MMM-yyyy H'h' m'm'`. For more information, read the Javadoc for classes java.text.DateFormat and/or java.text.SimpleDateFormat [JDKJAVADOC].
Reserved Characters for Date Format

In the above table, different values are used to indicate the arrangement of reserved characters. The specific output format depends on the number of times the different elements are repeated in each position:

- **Text**: use 4 or more characters to specify complete form; less than 4 characters to use the abbreviated form. For instance, if a date pattern specifies EEEE in the day of the week position, it indicates that day of the week should be shown using the complete form (e.g. ‘Monday’) instead of the abbreviated form (e.g. ‘Mon’).

- **Number**: it always uses the minimum number of digits possible. 0s are added to the left of the shortest numbers if required. The year is a special case: if the number of ‘y’ is 2, the year is shortened to 2 digits.

- **Text & Number**: 3 or more characters to represent it as text; otherwise a number is used. For instance, if a date pattern specifies MMM in the month position, it indicates that months should be shown using the text name (e.g. ‘Jul’). If the pattern specifies MM, the month will be shown as a number.

In a date format the characters that are not found in the ranges [‘a’..‘z’] or [‘A’..‘Z’] are considered constants, i.e. characters such as ‘:’, ‘.’, ‘ ‘, ‘,’ ‘#’ and ‘@’ appear in the resulting date, although they are not in inverted commas in the format pattern.

- **Configuration of real numbers**: Facilitates the configuration of the data types float and double.

  - **doubleDecimalPosition** – Indicates the number of decimal positions to be used to represent a double-type or float-type value (real numbers).

  - **doubleDecimalSeparator** – Represents the decimal separator used in a real number.

  - **doubleGroupSeparator** – Specifies the group separator for real numbers.
The statement required to create the internationalization configuration `es_euro`, which contains the most frequently used values in Spain, is shown below:

```sql
CREATE MAP I18N i18n_us_pst (
    'country' = 'US'
    'currency' = 'DOLAR'
    'currencydecimalposition' = '2'
    'currencydecimalseparator' = ''
    'currencygroupseparator' = ''
    'currencysymbol' = ''
    'datepattern' = 'd-MMM-yyyy H''h'' m''m'''
    'doubledecimalposition' = '2'
    'doubledecimalseparator' = ''
    'doublegroupseparator' = ''
    'language' = 'en'
    'moneypattern' = '###,###,###.##'
    'timepattern' = 'DAY'
    'timezone' = 'PST'
);
```

**Figure 132**  
Internationalization configuration `es_euro`

### 19.5 EXECUTION CONTEXT OF A QUERY AND INTERPOLATION STRINGS

This section describes the concepts of execution context and interpolation string. These instruments are used in Virtual DataPort to parameterize certain expressions used by the wrapper or the data source associated with a specific base relation depending on the queries made on this relation (see section 18).

The execution context of a query is made up of a series of variables that take the form of key/value pairs, where both the key and the value are strings. When a specific query is executed, a variable is added to the context for each query condition. The name associated with this variable is the attribute name and the operator used in the condition, separated by the character ‘#’ [ATTRIBUTE#operator]. The value associated with the variable will be the value indicated in the right side of the condition. Where the query only includes one query condition for this attribute, the name of the ATTRIBUTE variable can also be used, without specifying the operator.

**NOTE:** The variables may not work properly when the wrapper receives more than one query condition using the same attribute and operator.

The variables contained in the execution context can be used in the so-called interpolation strings.

An interpolation string is an expression using execution context variables, which generates a string as a result. A variable in an interpolation string must be specified by prefixing it with the symbol ‘@’ followed by the name of the variable, provided that this name is a string of alphanumeric characters (letters and the characters ‘#’ and ‘_’). Variables with a name that includes any other character can be specified by including the name between the symbols ‘@{‘ and ‘}’.

**NOTE:** When any of the symbols '@', '\', '^', '(' or ')' appear in the constant parts of the interpolation string, they must be escaped by the character '\' (i.e. '\@', '\\', '\^', '{(', '}'). Note that this implies that, on specifying local file-type paths in Windows Operating Systems, the character '\' must be escaped as '\\'.

**Example:** Suppose you have a Web server that allows accessing to certain reports from the departments of a particular company encoded into XML. The path to access the report from each department is the same, except for...
the name of the file that matches the name of the department (e.g.
http://examplesite.com/exampleroute/reports/DPT1.xml
http://examplesite.com/exampleroute/reports/DPT2.xml ...).

Now suppose that you want to build a DataPort base relation that allows accessing to these reports. To do so you
must create an XML-type data source (see section 18.3.5) and an XML-type wrapper (see section 18.4.8). This base
relation (we will term it as DPT_REPORTS) is to contain a tuple for each department. Each tuple will have two
attributes: DPT_NAME [text type] and REPORT [that will contain the report data. This attribute will normally be
a DataPort compound type (see section 19.1).

When creating the data source for this base relation, the problem arises that the data file to be accessed depends on
the department referred to by the query. To solve this problem, an http path could be specified in the ROUTE
parameter with a connection string such as:

http://examplesite.com/exampleroute/reports/@{DPT_NAME}.xml

Hence, queries such as the following can be executed:

SELECT REPORT FROM DPT_REPORTS WHERE DPT_NAME = 'DptName'

And the system would transparently access the file data corresponding to the department specified to answer the
query. For example, the path accessed for the previous query would be:

http://examplesite.com/exampleroute/reports/DptName.xml

Lastly, when an interpolation variable has a list of elements as a value (this happens in the cases of operators
allowing for a list of values as operands), the value associated with the variable will be the linking of the single
elements separated by the character '+' This can be used in the parameterization of certain aspects of the WWW
wrappers (see section 18.4.6).

19.6 ADDING VARIABLES TO SELECTION CONDITIONS (GETVAR AND SETVAR)

There are situations where we want to create an aggregation view with a condition in it. That is, creating a view
with a WHERE condition and a GROUP BY. The limitation of this is that the WHERE condition is static and cannot be
changed at runtime.

For example, if we have two views:

1. A base view CLIENT with these fields: name, income and state.

2. And a view WEALTHY_CLIENT_BY_STATE defined as:
   CREATE VIEW WEALTHY_CLIENT_BY_STATE AS
   SELECT state, COUNT(*)
   FROM client
   WHERE income > 1000000
   GROUP BY state

There is a limitation in the second view: the limit of income to consider a client wealthy is static. So, we have to
know this limit before creating the view. If we wanted to change this limit at runtime we could remove the WHERE
condition and add the field income to the GROUP BY fields. But then, we would be grouping by this field and we
might not want to do that. Besides, if income is not in the output of the base view you cannot add income to the
GROUP BY.

To avoid this problem, you can use the function GETVAR in the definition of the query. The syntax of this function is
GETVAR('<name of the variable>', '<type of the variable>', '<default value>')</n
Figure 133  Syntax of the function GETVAR

GETVAR tries to obtain the value of the variable <name of the variable> from the CONTEXT of the query. If it does not find it, it returns <default value>.

For example, you could define the view WEALTHY_CLIENT_BY_STATE like this:

```
CREATE VIEW WEALTHY_CLIENT_BY_STATE AS
SELECT state, COUNT(*)
FROM client
WHERE income >= GETVAR('_var_wealthy_client_income_limit', 'int', 1000000)
GROUP BY state
```

Figure 134  Definition of a view with a variable in the selection condition (GETVAR)

With this change, the limit of income is no longer static and we can query the view defining this value at runtime. For example:

```
SELECT * FROM WEALTHY_CLIENT_BY_STATE
CONTEXT ('VAR _var_wealthy_client_income_limit' = '250000')
```

Figure 135  Invoking a view defined with a variable in the selection condition

If we do not put a value for the variable in the CONTEXT of the query, the value used in the selection condition is the <default value> of the GETVAR function: 1000000.

Another option is obtaining the value of a variable from another view at runtime and putting this value in the CONTEXT with the function SETVAR. The syntax of this function is:

```
SETVAR('<name of the variable>', '<value of the variable')
```

Figure 136  Syntax of the function SETVAR

E.g. we have a DF base view INCOME_LIMIT that returns one row with the value that we want to use for the variable 'var_wealthy_client_income_limit'.

```
SELECT WEALTHY_CLIENT_BY_STATE.*
FROM
  (SELECT SETVAR('_var_wealthy_client_income_limit', limit)
   FROM INCOME_LIMIT WHERE type='wealthy')
NESTED ORDERED JOIN
WEALTHY_CLIENT_BY_STATE;
```

Figure 137  Invoking a view defining a variable in the selection condition

We execute a NESTED JOIN between the two views because in this type of join, the left branch is executed first. That means that the Server queries the view INCOME_LIMIT first and the function SETVAR puts the value of the
variable in the CONTEXT. Then, when the right branch is executed, `GETVAR` will find the value of the variable `_var_wealthy_client_income_limit` in the CONTEXT.

**Note**: if the query of the "left side" branch of the join returns more than one row, the `SETVAR` function will only take into account the value of the field of the first row.
For example, if we have three views:

1. A base view `CLIENT` with the fields `name, income, address and state`.

2. A view `WEALTHY_CLIENT` defined as a projection of the view `CLIENT` with the condition `(income > 1000000)`.

3. A view `WEALTHY_CLIENT_BY_STATE` defined as an aggregation view of `WEALTHY_CLIENT`:

   ```sql
   CREATE VIEW WEALTHY_CLIENT_BY_STATE AS
   SELECT state, COUNT(*)
   FROM wealthy_client
   GROUP BY state
   ```

The problem is that in these views, the limit to consider a client wealthy is static. If we want to change that limit in every query, we can consider using an interpolation variable in the definition of the view `WEALTHY_CLIENT`. But if we do that, the views `WEALTHY_CLIENT` and `WEALTHY_CLIENT_BY_STATE` will have an extra field. As an alternative, we can use the `GETVAR` function in the definition of the view `WEALTHY_CLIENT`:

```sql
CREATE VIEW WEALTHY_CLIENT AS
SELECT * FROM CLIENT
WHERE income >= GETVAR('_var_wealthy_client_income_limit', 'int', 1000000)
```

**Figure 138** Definition of a view with a variable in the selection condition (GETVAR)

With this change, we can query the view `WEALTHY_CLIENT_BY_STATE` defining at runtime the value of the limit of income:

```sql
SELECT * FROM WEALTHY_CLIENT_BY_STATE
CONTEXT ('VAR _var_wealthy_client_income_limit' = '250000')
```

**Figure 139** Invoking a view defined with a variable in the selection condition

If we do not put a value for the variable in the `CONTEXT` of the query, the value used in the selection condition is the `<default value>` of the `GETVAR` function: `1,000,000`.

Another option is obtaining the value of a variable from another view at runtime and putting this value in the `CONTEXT` with the function `SETVAR`.

E.g. we have a DF base view `INCOME_LIMIT` that returns one row with the value that we want to use for the variable `_var_wealthy_client_income_limit`.

```sql
```
SELECT WEALTHY_CLIENT_BY_STATE.*
FROM
  (SELECT SETVAR('_var_wealthy_client_income_limit', limit)
   FROM INCOME_LIMIT WHERE type='wealthy')
NESTED ORDERED JOIN
  WEALTHY_CLIENT_BY_STATE;

**Figure 140** Invoking a view defining with a variable in the selection condition

We execute a **NESTED JOIN** between the two views because in this type of join, the left branch is executed first. That means that the Server queries the view `INCOME_LIMIT` first and the function `SETVAR` puts the value of the variable in the **CONTEXT**. Then, when the right branch is executed, `GETVAR` will find the value of the variable `_var_wealthy_client_income_limit` in the **CONTEXT**.
20 APPENDICES

20.1 SYNTAX OF CONDITION FUNCTIONS

This section describes the syntax of condition functions (that can also be used to define derived attributes). These functions can be grouped into different types based on the data type to which they are applied:
- Arithmetic functions
- Text processing functions.
- Date processing functions
- XML processing functions.
- Type conversion functions.

20.1.1 Arithmetic Functions

20.1.1.1 SUM

Description
The SUM function adds its arguments.

Syntax
SUM (number1, number2, [ number3, ... ]):numeric
  number1. Required. First number to be added.
  number2. Required. Second number to be added.
  number3. Optional. Third number to be added.

Example 1
SELECT sum(1, cast('double', 2.5), 4.6) as sumValue
FROM Dual();

<table>
<thead>
<tr>
<th>sumValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
</tr>
</tbody>
</table>

Example 2
SELECT (1 + cast('int', 2.9) + 4.6) as sumValue
FROM Dual();

<table>
<thead>
<tr>
<th>sumValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6</td>
</tr>
</tbody>
</table>

20.1.1.2 SUBTRACT

Description
This function subtracts two numbers.

Syntax
SUBTRACT(number1, number2):numeric
  number1. Required. First number to be subtracted from.
  number2. Required. Second number to be subtracted.
Example 1

```
SELECT subtract(10, 2.5) as subtractValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>subtractValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5</td>
</tr>
</tbody>
</table>

Example 2

```
SELECT (10 - cast('int', 2.5)) as subtractValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>subtractValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

20.1.1.3 MULT

**Description**
The MULT function multiplies its arguments.

**Syntax**

```
MULT (number1, number2, [ number3, ... ]):numeric
```

- number1. Required. First number to be multiplied.
- number2. Required. Second number to be multiplied.
- number3. Optional. Third number to be multiplied.

**Example 1**

```
SELECT mult(10, 2.5) as multValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>multValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.0</td>
</tr>
</tbody>
</table>

**Example 2**

```
SELECT (10 * 2.5) as multValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>multValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.0</td>
</tr>
</tbody>
</table>

20.1.1.4 DIV

**Description**
Divides two numbers.

**Syntax**

```
DIV (dividend:numeric, divisor:numeric):numeric
```

- dividend. Required. The dividend of the operation.
- divisor. Required. The divisor of the operation.

**Example 1**

```
SELECT div(10, 2.5) as divValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>divValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.0</td>
</tr>
</tbody>
</table>
Example 2

```
SELECT (10 / cast('double', 2)) as divValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>divValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
</tr>
</tbody>
</table>

20.1.1.5 MIN

**Description**
Returns the minimum value in a list of arguments.

**Syntax**

```
MIN (number1, number2, [ number3, ... ]):numeric
```
- number1. Required.
- number2. Required.
- number3. Optional.

**Example**

```
SELECT min(5, 10, 3.2) as minValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>minValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
</tr>
</tbody>
</table>

20.1.1.6 MAX

**Description**
Returns the maximum value in a list of arguments.

**Syntax**

```
MAX (number1, number2, [ number3, ... ]):numeric
```
- number1. Required.
- number2. Required.
- number3. Optional.

**Example**

```
SELECT max(5, 10, 3.2) as maxValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>maxValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
</tr>
</tbody>
</table>

20.1.1.7 ABS

**Description**
Returns the absolute value of a number.

**Syntax**

```
ABS(value:numeric):numeric
```
- value. Required. The number of which absolute value will be calculated.
Example

```sql
SELECT abs(-5) as absoluteValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>absoluteValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

20.1.1.8 MOD

**Description**

Returns the result of the module operation: the remainder of the integer division of the first and second arguments. This function has an infix version and its operator is '%'.

**Syntax**

\[
\text{MOD}(\text{dividend}:\text{int}, \text{divisor}:\text{int}):\text{int} \\
\text{MOD}(\text{dividend}:\text{long}, \text{divisor}:\text{long}):\text{long}
\]

- \text{dividend}: Required. Integer or field of type integer.
- \text{divisor}: Required. Integer or field of type integer.

**Examples**

Consider the following view \( V \):

<table>
<thead>
<tr>
<th>INTSAMPLE</th>
<th>LONGSAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>-4</td>
<td>-55</td>
</tr>
<tr>
<td>8</td>
<td>70</td>
</tr>
</tbody>
</table>

And the view \( \text{modView} \) created with the command:

```sql
CREATE VIEW modView AS
    SELECT intsample, mod(intsample, 2) as s1, longsample, mod(longsample, 2) as s2
    FROM \( V \);
```

**Example 1**

```sql
SELECT * FROM modView
```

<table>
<thead>
<tr>
<th>INTSAMPLE</th>
<th>S1</th>
<th>LONGSAMPLE</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>-55</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>70</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example 2**

```sql
SELECT 10\%2 FROM modView
```

<table>
<thead>
<tr>
<th>MOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

20.1.1.9 CEIL

**Description**

Returns the smallest integer not less than the argument.
Syntax
CEIL(value:decimal):long
CEIL(value:int):int
CEIL(value:long):long
CEIL(value:money):money

- value. Required. The value to round off.

Example

```sql
SELECT ceil(5.08) as ceilValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>ceilValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

20.1.1.10 FLOOR

Description
Returns the largest integer not greater than the argument.

Syntax
FLOOR(value:decimal):long
FLOOR(value:int):int
FLOOR(value:long):long
FLOOR(value:money):money

- value. Required. Value to round off.

Example

```sql
SELECT floor(5.98) as floorValue
FROM Dual();
```

<table>
<thead>
<tr>
<th>floorValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

20.1.1.11 ROUND

Description
Rounds a number to the nearest integer.

Syntax
ROUND(value:numeric):numeric

- value. Required. Value to round off.

Example

```sql
SELECT round(5.98) as roundValue1,round(5.08) as roundValue2
FROM Dual();
```

<table>
<thead>
<tr>
<th>roundValue1</th>
<th>roundValue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

20.1.1.12 POWER

Description
Returns the result of a number raised to a power.
Syntax
POWER(number:numeric, power:int):double
  • number. Required. Base number.
  • power. Required. Exponent to which the base number is raised.

Example
SELECT power(5,2) as powerValue
FROM Dual();

<table>
<thead>
<tr>
<th>powerValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

20.1.1.13 SQRT

Description
Returns a positive square root.

Syntax
SQRT(value:numeric):double
  • value. Required. Number for which you want the square root.

Example
SELECT sqrt(25) as sqrtValue
FROM Dual();

<table>
<thead>
<tr>
<th>sqrtValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
</tr>
</tbody>
</table>

20.1.1.14 LOG

Description
Returns the logarithm of a number in a base-ten.

Syntax
LOG(value:numeric):double
  • value. Required. Positive real number for which you want the logarithm.

Example
SELECT log(100) as logValue
FROM Dual();

<table>
<thead>
<tr>
<th>logValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
</tr>
</tbody>
</table>

20.1.1.15 RAND

Description
Returns a random value between zero and one.

Syntax
It doesn’t receive any parameter.

Example
Consider the view V:
### 20.1.2 Text Processing Functions

Text processing functions are used for text manipulation and transformation.

#### 20.1.2.1 TEXTCONSTANT

**Description**
Parse a parameter as a text.

**Syntax**

\[
\text{TEXTCONSTANT(text):text}
\]

- text. Required. Text to be displayed as is in the result.

**Example**

```sql
SELECT originalText, textconstant('I like to fly to') as constantText
FROM myTable;
```

<table>
<thead>
<tr>
<th>originalText</th>
<th>constantText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco, CA</td>
<td>I like to fly to</td>
</tr>
<tr>
<td>San Jose, CA</td>
<td>I like to fly to</td>
</tr>
<tr>
<td>Birmingham , AL</td>
<td>I like to fly to</td>
</tr>
<tr>
<td>NY, NY</td>
<td>I like to fly to</td>
</tr>
</tbody>
</table>

#### 20.1.2.2 CONCAT

**Description**
Concatenates parameters into one string.

**Syntax**

\[
\text{CONCAT(text1, text2, [text3], ...):text}
\]

- text1. Required. The first text item to be concatenated.
- text2. Required. The second text item to be concatenated.
- text3. Optional. Another text item to be concatenated.

**Example**

```sql
SELECT originalText, concat('I like to fly to ', originalText, ' every month') as concatText
FROM myTable;
```

<table>
<thead>
<tr>
<th>originalText</th>
<th>concatText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco, CA</td>
<td>I like to fly to every month</td>
</tr>
<tr>
<td>San Jose, CA</td>
<td>I like to fly to every month</td>
</tr>
<tr>
<td>Birmingham , AL</td>
<td>I like to fly to every month</td>
</tr>
<tr>
<td>NY, NY</td>
<td>I like to fly to every month</td>
</tr>
</tbody>
</table>
20.1.2.3  INSTR

Description
Returns the index of a string within another string.

Syntax
INSTR(str1:text, str2:text):int

Returns the index of the first character of the first occurrence of 'str2' within 'str1'. If 'str2' is not present within 'str1', it returns -1. The index of the first character is 0.

Example
SELECT originalText, instr(originalText, 'i') as result FROM myTable;

<table>
<thead>
<tr>
<th>originalText</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco, CA</td>
<td>9</td>
</tr>
<tr>
<td>San Jose, CA</td>
<td>-1</td>
</tr>
<tr>
<td>Birmingham, AL</td>
<td>1</td>
</tr>
<tr>
<td>NY, NY</td>
<td>-1</td>
</tr>
</tbody>
</table>

20.1.2.4  LEN

Description
Returns the number of characters in a text string

Syntax
LEN(value:text):int
  - value. Required. The text whose length you want to find. Spaces count as characters.

Example
SELECT originalText, len(originalText) as lenText FROM myTable;

<table>
<thead>
<tr>
<th>originalText</th>
<th>lenText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco, CA</td>
<td>18</td>
</tr>
<tr>
<td>San Jose, CA</td>
<td>13</td>
</tr>
<tr>
<td>Birmingham, AL</td>
<td>15</td>
</tr>
<tr>
<td>NY, NY</td>
<td>7</td>
</tr>
</tbody>
</table>

20.1.2.5  REPLACE

Description
Substitutes new text for old text in a text string

Syntax
REPLACE(value:text, from:text, to:text):text
  - value. Required. Text which you want to replace some/all of it.
• from. Required. All occurrences to be replaced.
• to. Required. Text which will replace all the occurrences of old_text.

Example

```sql
SELECT originalText, replace(originalText,'CA','California') as replaceText FROM myTable;
```

<table>
<thead>
<tr>
<th>originalText</th>
<th>replaceText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco , CA</td>
<td>San Francisco , California</td>
</tr>
<tr>
<td>San Jose , CA</td>
<td>San Jose , California</td>
</tr>
<tr>
<td>Birmingham , AL</td>
<td>Birmingham , AL</td>
</tr>
<tr>
<td>NY, NY</td>
<td>NY, NY</td>
</tr>
</tbody>
</table>

20.1.2.6 REPLACEMAP

Description

Substitutes new text for old text in a text string based on key/value pairs from another view or map.

Syntax

```sql
REPLACEMap(searchText:{text|enumerated}, mapName{text|enumerated}):text
REPLACEMap(searchText:text, viewName:text, keyField:text, valueField:text):text
```

• searchText. Required. Text which you want to replace some/all of it.
• mapName. Required. Map that contains key/value pairs.
• viewName. Required. VirtualDataPort view which contains key/value pairs.
• keyField. Required. A field from view_name contains keys.
• valueField. Required. A field from view_name contains values.

Examples

Example 1

Consider the following map:

```sql
CREATE MAP simple daysOfTheWeek(
'Sun' = 'Sunday'
'Mon' = 'Monday'
'Tus' = 'Tuesday'
'Wed' = 'Wednesday'
'Thur'= 'Thursday'
'Fri' = 'Friday'
'Sat' = 'Saturday'
'Sun' = 'Sunday'
);
```

Now consider the following query:

```sql
SELECT textblock ,
replacemap (textblock, 'daysOfTheWeek') as textblockWithFullName
FROM V;
```

<table>
<thead>
<tr>
<th>Textblock</th>
<th>textblockWithFullName</th>
</tr>
</thead>
<tbody>
<tr>
<td>I like to travel on Sun</td>
<td>I like to travel on Sunday</td>
</tr>
<tr>
<td>I am available to travel on Mon</td>
<td>I am available to travel on Monday</td>
</tr>
<tr>
<td>My best day of vacation is Sat because I see my relatives on Wed</td>
<td>My best day of vacation is Saturday because I see my relatives on Wednesday</td>
</tr>
</tbody>
</table>
Example 2
Consider the following VirtualDataPort view `days_of_the_week`:

<table>
<thead>
<tr>
<th>FULL_DAY_NAME</th>
<th>ABBREVIATED_FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>Sun</td>
</tr>
<tr>
<td>Monday</td>
<td>Mon</td>
</tr>
<tr>
<td>Tuesday</td>
<td>Tus</td>
</tr>
<tr>
<td>Wednesday</td>
<td>Wed</td>
</tr>
<tr>
<td>Thursday</td>
<td>Thur</td>
</tr>
<tr>
<td>Friday</td>
<td>Fri</td>
</tr>
<tr>
<td>Saturday</td>
<td>Sat</td>
</tr>
<tr>
<td>Sunday</td>
<td>Sun</td>
</tr>
</tbody>
</table>

Now consider the following query:

```sql
SELECT textblock, replacemap (textblock, 'days_of_the_week', 'abbreviated_format', 'full_day_name') AS textblockWithFullName
FROM V;
```

<table>
<thead>
<tr>
<th>Textblock</th>
<th>textblockWithFullName</th>
</tr>
</thead>
<tbody>
<tr>
<td>I like to travel on Sun</td>
<td>I like to travel on Sunday</td>
</tr>
<tr>
<td>I am available to travel on Mon</td>
<td>I am available to travel on Monday</td>
</tr>
<tr>
<td>My best day of vacation is Sat because I see my relatives on Wed</td>
<td>My best day of vacation is Saturday because I see my relatives on Wednesday</td>
</tr>
</tbody>
</table>

20.1.2.7 LOWER

Description
Converts text to lowercase.

Syntax
`LOWER(value:text):text`
  - value. Required. Text to convert to lower case.

Example

```sql
SELECT originalText, lower(originalText) as lowerText
FROM Mytable;
```

<table>
<thead>
<tr>
<th>originalText</th>
<th>lowerText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco , CA</td>
<td>san francisco , ca</td>
</tr>
<tr>
<td>San Jose , CA</td>
<td>san jose , ca</td>
</tr>
<tr>
<td>Birmingham , AL</td>
<td>birmingham , al</td>
</tr>
<tr>
<td>NY, NY</td>
<td>ny, ny</td>
</tr>
</tbody>
</table>

20.1.2.8 LTRIM

Description
Returns a copy of the string without its leading whitespaces.

Syntax
`LTRIM(value:text):text`
  - value. Required.
20.1.2.9  UPPER

**Description**
Converts text to uppercase.

**Syntax**
UPPER(value: text): text
- value. Required. Text to convert to upper case.

**Example**

```sql
SELECT originalText , upper (originalText) as upperText
FROM Mytable;
```

<table>
<thead>
<tr>
<th>originalText</th>
<th>upperText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco , CA</td>
<td>SAN FRANCISCO , CA</td>
</tr>
<tr>
<td>San Jose , CA</td>
<td>SAN JOSE , CA</td>
</tr>
<tr>
<td>Birmingham , AL</td>
<td>BIRMINGHAM , AL</td>
</tr>
<tr>
<td>NY, NY</td>
<td>NY, NY</td>
</tr>
</tbody>
</table>

20.1.2.10  SUBSTRING

**Description**
Returns a substring that begins at the specified startIndex of the input string and extends to the character at index endIndex-1. Thus the length of the result is endIndex - startIndex.

**Syntax**
SUBSTRING(value: text, startIndex: int, endIndex: int): text
- value. Required. Text string containing the characters to extract.
- startIndex. Required. Index of the first character of the new substring. The startIndex of the first character of the input string is 0.
- endIndex. Required. Index of the last character. The result does not include this last character.

**Example**

```sql
SELECT city, substring(city, 0, 3) as substringCity
FROM locations;
```

<table>
<thead>
<tr>
<th>city</th>
<th>substringCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>San</td>
</tr>
<tr>
<td>San Francisco</td>
<td>San</td>
</tr>
<tr>
<td>Birmingham</td>
<td>Bir</td>
</tr>
<tr>
<td>NY</td>
<td>null</td>
</tr>
</tbody>
</table>

20.1.2.11  REGEXP

**Description**
Text transformation based on regular expressions. REGEXP uses a regular expression to search for a string and another regular expression to output the result.

**Syntax**
REGEXP(original_text: text, regexp: text, replacement_regexp: text): text
- original_text. Required. Text which you want to replace some/all of it.
- regexp. Required. Regular expression used to search for a specific text in original_text.
- replacement_regexp. Required. Regular expression used to format the output of old_text_regexp.
Example

```
SELECT REGEXP('Shakespeare,William','(\w+),\s+(\w+)',' hello $1 you are the man Mr. $2') as mytext
FROM Dual();
```

```
Mytext
hello Shakespeare you are the man Mr. William
```

20.1.2.12  RTRIM

**Description**
Returns a copy of the string without its trailing whitespaces.

**Syntax**
```
RTRIM (value: text): text
```

- **value. Required.**

20.1.2.13  TRIM

**Description**
Removes all spaces and carriage returns from text except for single spaces between words.

**Syntax**
```
TRIM (value: text): text
```

- **value. Required.** Text from which you want spaces and carriage returns to be removed from.

**Example**
```
SELECT originalText, trim (originalText) as trimText
FROM Mytable;
```

```
originalText | trimText
-------------|-------------
San Francisco, CA | San Francisco, CA
San Jose, CA | San Jose, CA
Birmingham, AL | Birmingham, AL
NY, NY | NY, NY
```

20.1.2.14  REMOVEACCENTS

**Description**
Replaces all characters with an accent with the same characters without accent.

**Syntax**
```
REMOVEACCENTS (value: text): text
```

- **value. Required.** Text you want to remove accents from.

**Example**
```
SELECT removeaccents('bёёáéíóúàèìòù') as textWoAccent
FROM Dual();
```

```
textWoAccent
Bёёaeiou aeiou
```
20.1.2.15 SIMILARITY

Description
Calculates the textual similarity between two text strings based on a given textual similarity algorithm.

Syntax
SIMILARITY(value1: text, value2: text [ , algorithm: text ]): double
- value1. Required. Text to be compared.
- value2. Required. Text to be compared with value1.
- algorithm. Optional. Algorithm to use. VirtualDataPort provides the following textual similarity algorithms:

<table>
<thead>
<tr>
<th>Algorithms based on distance between text strings</th>
<th>Algorithms Based on the appearance of common terms in the texts</th>
<th>Combinations of both</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScaledLevenshtein</td>
<td>TFIDF</td>
<td>JaroWinklerTFIDF</td>
</tr>
<tr>
<td>JaroWinkler</td>
<td>Jaccard</td>
<td>JaroWinkler</td>
</tr>
<tr>
<td>Jaro</td>
<td>UnsmoothedJS</td>
<td>Jaro</td>
</tr>
<tr>
<td>Level2 Jaro</td>
<td></td>
<td>Level2Jaro</td>
</tr>
<tr>
<td>MongeElkan</td>
<td></td>
<td>MongeElkan</td>
</tr>
<tr>
<td>Level2MongeElkan</td>
<td></td>
<td>Level2MongeElkan</td>
</tr>
</tbody>
</table>

Example
SELECT city, similarity (city, 'San') as measure
FROM doclist
ORDER by measure DESC;

<table>
<thead>
<tr>
<th>City</th>
<th>upperText</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>0.71</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.71</td>
</tr>
<tr>
<td>NY</td>
<td>0.00</td>
</tr>
<tr>
<td>Birmingham</td>
<td>0.00</td>
</tr>
</tbody>
</table>

20.1.2.16 SPLIT

Description
Splits strings around matches of a given regular expression and returns an array containing these substrings.

Syntax
split(regexp: text, value: text): array
- regexp. Required. A regular expression. The substrings that match this regular expression are not included in the result.
- value. Required. Field name or text to split.

Examples
Consider the following view V:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.10</td>
<td>I am some text</td>
<td>21-ene-2005 0h 0m 0s</td>
</tr>
<tr>
<td>-80.10</td>
<td>Text is % needed always</td>
<td>12-mar-2005 12h 30m 0s</td>
</tr>
<tr>
<td>20.50</td>
<td>Text for a living</td>
<td>01-feb-2006 16h 45m 0s</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Example 1
SELECT split(' ', 'hello bye') FROM V
Example 2

```sql
SELECT split(' ', B) FROM V
```

Example 3

```sql
SELECT split('e', C) FROM V
```

Example 4

```sql
SELECT split('\.\.', A) FROM V
```

### 20.1.3 Date Processing Functions

#### 20.1.3.1 ADDDAY

**Description**

It returns the date passed as parameter with its field day rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**

```sql
ADDDAY(date: date, increment: int): date
```

- **date**: Required. The date field.
- **increment**: Required. The amount to increase the field day. If the number is negative, the field is decreased.

**Example**

```sql
SELECT time, ADDDAY(time, 8)
FROM v
```
### 20.1.3.2 ADDHOUR

**Description**
It returns the date passed as parameter with its field hour rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**
```
ADDHOUR(date:date, increment:int):date
```
- `date`: Required. The date field.
- `increment`: Required. The amount to increase the field hour. If the number is negative, the field is decreased.

**Example**
```
SELECT time, ADDHOUR(time, -2)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDHOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jul, Thu 7, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Dec, Fri 31, 2010 22h 59m 56s</td>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
</tr>
</tbody>
</table>

### 20.1.3.3 ADDMINUTE

**Description**
It returns the date passed as parameter with its field minute rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**
```
ADDMINUTE(date:date, increment:int):date
```
- `date`: Required. The date field.
- `increment`: Required. The amount to increase the field minute. If the number is negative, the field is decreased.

**Example**
```
SELECT time, ADDMINUTE(time, 10)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDMINUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 29, 2005 19h 29m 41s</td>
</tr>
<tr>
<td>Jun, Thu 30, 2005 01h 0m 0s</td>
<td>Jul, Wed 29, 2005 23h 0m 0s</td>
</tr>
</tbody>
</table>

### 20.1.3.4 ADDMONTH

**Description**
It returns the date passed as parameter with its field month rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**
```
ADDMONTH(date:date, increment:int):date
```
- `date`: Required. The date field.
- `increment`: Required. The amount to increase the field month. If the number is negative, the field is decreased.

**Example**
```
SELECT time, ADDMONTH(time, 10)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDMONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 29, 2005 19h 29m 41s</td>
</tr>
<tr>
<td>Jun, Thu 30, 2005 22h 59m 0s</td>
<td>Jun, Thu 30, 2005 23h 09m 0s</td>
</tr>
</tbody>
</table>
Example

```sql
SELECT time, ADDMONTH(time, -12)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDMONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Tue 29, 2004 19h 19m 41s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Jan, Fri 8, 2010 22h 59m 56s</td>
</tr>
</tbody>
</table>

20.1.3.5  ADDSECOND

**Description**
It returns the date passed as parameter with its field second rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**
```
ADDSECOND(date:date, increment:int):date
```
- **date**: Required. The date field.
- **increment**: Required. The amount to increase the field second. If the number is negative, the field is decreased.

**Example**

```sql
SELECT time, ADDSECOND(time, 5)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDSECOND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 29, 2005 19h 19m 46s</td>
</tr>
<tr>
<td>Jun, Thu 30, 2005 22h 59m 56s</td>
<td>Jun, Thu 30, 2005 23h 0m 1s</td>
</tr>
</tbody>
</table>

20.1.3.6  ADDWEEK

**Description**
It returns the date passed as parameter with its field week rolled up (or down, if the integer is negative) by the amount specified. That is, rolled up or down in multiples of 7 days.

**Syntax**
```
ADDWEEK(date:date, increment:int):date
```
- **date**: Required. The date field.
- **increment**: Required. Number of times to increase the field day, 7 days. If the number is negative, the field is decreased.

**Example**

```sql
SELECT time, ADDWEEK(time, -2)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDWEEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 15, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Dec, Sat 25, 2010 22h 59m 56s</td>
</tr>
</tbody>
</table>

We can see that the date is rolled back fourteen days (2 weeks). It rolls back, instead of rolling up, because the parameter `increment` is a negative number.
20.1.3.7  ADDYEAR

**Description**
It returns the date passed as parameter with its field year rolled up (or down, if the integer is negative) by the amount specified.

**Syntax**
```
ADDYEAR(date:date, increment:int):date
```
- **date**: Required. The date field.
- **increment**: Required. The amount to increase the field year. If the number is negative, the field is decreased.

**Example**
```
SELECT time, ADDYEAR(time, 7)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDYEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Fri 29, 2012 19h 19m 41s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Jan, Mon 8, 2018 22h 59m 56s</td>
</tr>
</tbody>
</table>

20.1.3.8  FIRSTDAYOFMONTH

**Description**
It returns the date passed as parameter, with the field day rolled down to the first day of the month.

**Syntax**
```
FIRSTDAYOFMONTH(date:date):date
```
- **date**: Required.

**Example**
```
SELECT time, FIRSTDAYOFMONTH(time)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>FIRSTDAYOFMONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 1, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Jan, Sat 1, 2011 22h 59m 56s</td>
</tr>
</tbody>
</table>

20.1.3.9  FIRSTDAYOFWEEK

**Description**
It returns the date passed as parameter, with the field day rolled down to the first day of the week. The first day of the week is Monday.

**Syntax**
```
FIRSTDAYOFWEEK(date:date):date
```
- **date**: Required.

**Example**
```
SELECT time, FIRSTDAYOFWEEK(time)
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>FIRSTDAYOFWEEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Mon 27, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Jan, Mon 10, 2011 22h 59m 56s</td>
<td>Jan, Mon 10, 2011 22h 59m 56s</td>
</tr>
</tbody>
</table>

We can see that in the second row the day is already the first day of the week, so the output of the function is the same as the input.
20.1.3.10  FORMATDATE

Description
Returns a string containing a date-type formatted using the given pattern.
This function relies on the date and time formatting system of Java. Document [JAVADATEFORMAT] has a detailed explanation of it.

Syntax
FORMATDATE(date_pattern:text, date:date, locale:text):text

- date_pattern. Required. Pattern used to format the date passed in the second parameter (see section 20.5 for more information about date patterns format).
- date. Required. The date value to be formatted.
- locale. Optional. Internationalization configuration. Certain strings, like the names of the months, depend on the value of this parameter. I.e. “us_pst”, “es”, “gb”, ...

Examples
Example 1

<table>
<thead>
<tr>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>June, Wed 29, 2005</td>
</tr>
<tr>
<td>June, Wed 29, 2005</td>
</tr>
<tr>
<td>June, Wed 29, 2005</td>
</tr>
<tr>
<td>June, Wed 29, 2005</td>
</tr>
</tbody>
</table>

SELECT formatdate('MMMM, EE dd, yyyy', ttime, 'us_pst') AS DATE FROM internet_inc

Example 2

<table>
<thead>
<tr>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005.06.29 AD at 19:19:41</td>
</tr>
<tr>
<td>2005.06.29 AD at 19:19:41</td>
</tr>
<tr>
<td>2005.06.29 AD at 19:19:41</td>
</tr>
<tr>
<td>2005.06.29 AD at 19:19:41</td>
</tr>
</tbody>
</table>

SELECT formatdate("yyyy.MM.dd G 'at' HH:mm:ss", ttime, "us_pst") AS DATE FROM internet_inc

Text between single quotes is not interpreted (see ‘at’) and is copied to the output as it is. To add a single quote to the output write two single quotes "".

Example 3

<table>
<thead>
<tr>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:19 PM</td>
</tr>
<tr>
<td>7:19 PM</td>
</tr>
<tr>
<td>7:19 PM</td>
</tr>
<tr>
<td>7:19 PM</td>
</tr>
</tbody>
</table>

SELECT formatdate("h:mm a", ttime, "us_pst") AS DATE FROM internet_inc

Example 4

SELECT formatdate("yyMMddHHmmss", ttime, "us_pst") AS DATE FROM internet_inc
20.1.3.11 GETDAY

**Description**
Returns the day of a given date. The function returns a long data-type ranging from 1 to 31.

**Syntax**
```sql
GETDAY(date:date):long
```
- **date**: Required. Date to retrieve the day from.

**Example**
```sql
SELECT getday(to_date('M dd yyyy' , '3 05 2008') ) as theDayOfMonth
FROM Dual();
```

```
theDayOfMonth
5
```

20.1.3.12 GETHOUR

**Description**
It returns the hour of a given date. The function returns a long data-type, ranging from 0 (12:00 A.M.) to 23 (11:00 P.M.).

**Syntax**
```sql
GETHOUR(date:date):long
```
- **date**: Required. Date to retrieve the hour from.

**Example**
```sql
SELECT getHour(to_date('M dd yyyy HH:mm:ss' , '3 05 2008 21:17:05') )
as theHourOfTime
FROM Dual();
```

```
theHourOfTime
21
```

20.1.3.13 GETMINUTE

**Description**
It returns the minute of a given date. The function returns a long data-type, ranging from 0 to 59.

**Syntax**
```sql
GETMINUTE(date:date):long
```
- **date**: Required. Date to retrieve the minute from.

**Example**
```sql
SELECT getMinute(to_date('M dd yyyy HH:mm:ss' , '3 05 2008 21:17:05') )
as theMinuteOfTime
FROM Dual();
```
20.1.3.14 GETSECOND

**Description**
It returns the second of a given date. The function returns a long data-type, ranging from 0 to 59.

**Syntax**
`GETSECOND(date:date):long`

- `date`. Required. Date to retrieve the second from.

**Example**
```
SELECT getSecond (to_date('M dd yyyy HH:mm:ss' , '3 05 2008 21:17:05') )
as theSecondOfTime
FROM Dual();
```

```
theSecondOfTime
5
```

20.1.3.15 GETTIMEINMILLIS

**Description**
It returns the number of milliseconds from January 1, 1970, 00:00:00 GMT to the date passed as parameter.

**Syntax**
`GETTIMEINMILLIS(date:date):long`

- `date`. Required.

**Example**
```
SELECT getTimeInMillis (to_date('M dd yyyy HH:mm:ss' , '1 01 2010 00:00:00') )
as theMillisOfTime
FROM Dual();
```

```
theMillisOfTime
1262300400000
```

20.1.3.16 GETMONTH

**Description**
It returns the number of month in a year of a given date. The function returns a long data-type, ranging from 1 (January) to 12 (December).

**Syntax**
`GETMONTH(date:date):long`

- `date`. Required. Date to retrieve the number of month from.

**Example**
```
SELECT getMonth (to_date('M dd yyyy HH:mm:ss' , '3 05 2008 21:17:05') )
AS theMonthOfDate
FROM Dual();
```

```
theMonthOfDate
3
```
20.1.3.17 GETYEAR

Description
It returns the year of a given date.

Syntax
GETYEAR(date:date):long
  • date. Required. Date to retrieve the year from.

Example
SELECT getYear (to_date('M dd yyyy HH:mm:ss' , '3 05 2010 21:17:05') )
as theYearOfDate
FROM Dual();

theYearOfDate
2010

20.1.3.18 LASTDAYOFMONTH

Description
It returns the date passed as parameter with the field day rolled up to the last day of the month.

Syntax
LASTDAYOFMONTH(date:date):date
  • date. Required.

Example
SELECT time, LASTDAYOFMONTH (time)
FROM v

TIME                  | LASTDAYOFMONTH
---------------------|-----------------|
Jun, Wed 30, 2005 19h 19m 41s | Jun, Wed 30, 2005 19h 19m 41s |
Feb, Thu 10, 2011 22h 59m 56s  | Feb, Mon 28, 2011 22h 59m 56s |

We can see that in the first row the day is already the last day of the month, so the output of the function is the same as the input.

20.1.3.19 LASTDAYOFWEEK

Description
It returns the date passed as parameter with the field day rolled up to the last day of the week. The last day of the week is Sunday.

Syntax
LASTDAYOFWEEK(date:date):date
  • date. Required.

Example
SELECT time, LASTDAYOFWEEK(time)
FROM v

TIME                  | LASTDAYOFWEEK
---------------------|-----------------|
Jun, Wed 30, 2005 19h 19m 41s | Jul, Sun 3, 2005 19h 19m 41s |
Feb, Thu 10, 2011 22h 59m 56s  | Feb, Sun 13, 2011 22h 59m 56s |
20.1.3.20  MAX

**Description**
See appendix 20.1.1.6.

20.1.3.21  MIN

**Description**
See Appendix 20.1.1.5.

20.1.3.22  NEXTWEEKDAY

**Description**
It returns this date with its field day rolled up to the day of the next week specified by the parameter `weekDay`. The days of the week are: Sunday = 0, Monday = 1, Tuesday = 2 …

**Syntax**
```
NEXTWEEKDAY (date:date, weekDay:int):date
```
- `date`. Required.
- `weekDay`. Required. The day of the week that the date will be rolled up to.

**Example**
```
SELECT time, NEXTWEEKDAY(time, 3)
FROM v
```
```
<table>
<thead>
<tr>
<th>TIME</th>
<th>NEXTWEEKDAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 30, 2005 19h 19m 41s</td>
<td>Jul, Wed 6, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Feb, Thu 10, 2011 22h 59m 56s</td>
<td>Feb, Wed 16, 2011 22h 59m 56s</td>
</tr>
</tbody>
</table>
```

20.1.3.23  NOW

**Description**
Returns the current date and time.

**Syntax**
```
NOW():date
```

**Example**
```
SELECT now() as DateAndTimeNow
FROM Dual();
```
```
<table>
<thead>
<tr>
<th>DateAndTimeNow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-jan-2010 0h 0m 0s</td>
</tr>
</tbody>
</table>
```

20.1.3.24  PREVIOUSWEEKDAY

**Description**
It returns this date with its field day rolled down to the day of the past week specified by the parameter `weekDay`. The days of the week are: Sunday = 0, Monday = 1, Tuesday = 2 …

**Syntax**
```
PREVIOUSWEEKDAY (date:date, weekDay:int):date
```
- `date`. Required.
- `weekDay`. Required. The day of the week that the date will be rolled down to.
### Example

```
SELECT time, PREVIOUSWEEKDAY(time, 2) 
FROM v 
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>PREVIOUSWEEKDAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 30, 2005 19h 19m 41s</td>
<td>Jul, Tue 21, 2005 19h 19m 41s</td>
</tr>
<tr>
<td>Feb, Thu 10, 2011 22h 59m 56s</td>
<td>Feb, Tue 1, 2011 22h 59m 56s</td>
</tr>
</tbody>
</table>

### 20.1.3.25 TO_DATE

**Description**

Converts a date in the form of a string to a date data-type based on a date pattern and a locale. This function relies on the date and time formatting system of Java. Document [JAVADATEFORMAT] has a detailed explanation of it.

**Syntax**

```
TO_DATE(datePattern:text, dateValue:text [ , locale:text ]):date
```

- **datePattern.** Required. Pattern describing the date and time format of dateValue (see section 20.5 for more information)
- **dateValue.** Required. Date string which will be converted to data-type.
- **locale.** Optional. Internationalization configuration.

**Examples**

**Example 1**

```
SELECT to_date('M dd yyyy HH:mm:ss' , '3 05 2010 21:17:05') as dateAsDate 
FROM Dual();
```

<table>
<thead>
<tr>
<th>dateAsDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri Mar 05 21:17:05 2010</td>
</tr>
</tbody>
</table>

**Example 2**

```
SELECT to_date('yyyyMMddHHmmss', '20100701102030') as dateAsDate 
FROM Dual();
```

<table>
<thead>
<tr>
<th>dateAsDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu Jul 01 10:20:30 2010</td>
</tr>
</tbody>
</table>

**Example 3**

```
SELECT to_date("yyyy-MM-dd'T'HH:mm:ss.SSS", "2001-07-04T12:08:56.235") as dateAsDate 
FROM Dual();
```

<table>
<thead>
<tr>
<th>dateAsDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed Jul 04 12:08:56 2001</td>
</tr>
</tbody>
</table>

### 20.1.3.26 TRUNC

**Description**

It returns the date passed as parameter, truncated to a specific unit of measure. This function has the same syntax as the function TRUNC(date) of the Oracle database. The parameter `pattern` also has the same syntax.
Syntax

**TO_DATE** (date:date [pattern:text]):date
  
  • date. Required. Date to be truncated.
  
  • pattern. The date is truncated to the unit specified by this parameter. The syntax of these pattern is the same as the Oracle function **TRUNC** (date) [ORCL_TRUNC]. If pattern is missing, date is truncated to the nearest day.

Examples

Example 1

```sql
SELECT time, TRUNC(time, 'MONTH')
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDYEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Jun, Wed 1, 2005 0h 0m 0s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Jan, Sat 1, 2011 0h 0m 0s</td>
</tr>
</tbody>
</table>

Example 2

```sql
SELECT time, TRUNC(time, 'Q')
FROM v
```

<table>
<thead>
<tr>
<th>TIME</th>
<th>ADDYEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun, Wed 29, 2005 19h 19m 41s</td>
<td>Apr, Fri 1, 2005 0h 0m 0s</td>
</tr>
<tr>
<td>Jan, Sat 8, 2011 22h 59m 56s</td>
<td>Jan, Sat 1, 2011 0h 0m 0s</td>
</tr>
</tbody>
</table>

The pattern ‘Q’ means that the date will be truncated to the quarter.

20.1.4 XML Processing Functions

20.1.4.1 XMLQUERY

**Description**

Extracts information from an XML document using the XQuery language [XQUERY]

**Syntax**

XMLQUERY(XQueryExpression:text, isXQueryFile:boolean):xml
XMLQUERY(XQueryExpression:text, isXQueryFile:boolean, xmlValue:xml):xml
XMLQUERY(XQueryExpression:text, isXQueryFile:boolean, xmlValue:text, boolean isXMLFile:boolean):xml

  • XQueryExpression. XQuery expression used to query xml data.
  
  • isXQueryFile. True, if the parameter ‘XQueryExpression’ is a path to a file containing an XQuery expression. False, if ‘XQueryExpression’ is a literal or is the name of a field that contains an expression.
  
  • xmlValue. The XML to manipulate.
  
  • isXMLFile. True, if the parameter ‘xmlValue’ is a path to a file containing an XML document. If ‘isXMLFile’ is false or is missing, ‘xmlValue’ is a literal or the name of an XML field.

**Examples**

Consider the view v that only has one column of type ‘xml’ and one row:
Consider the file 'C:/books_info.xml' with the same content as the view V.

Consider the file 'C:/books.xq' with the following XQuery expression:

```xml
<ul>
  { for $b in //BOOKS/ITEM
  order by $b/TITLE return
    <li>
      { string($b/TITLE) } by { string($b/AUTHOR) }
    </li>
  }
</ul>
```

And consider the file 'C:/books2.xq' with an XQuery expression that transform the XML document of the file 'C:/books_info.xml':

```xml
<ul>
  { for $b in doc('c:/books_info.xml')//BOOKS/ITEM
  order by $b/TITLE return
    <li>
      { string($b/TITLE) } by { string($b/AUTHOR) }
    </li>
  }
</ul>
```

**Example 1**
The following queries have the same result:
Query 1:

```
SELECT XMLQUERY("<ul>
    { for $b in doc('C:/books_info.xml')//BOOKS/ITEM
        order by $b/TITLE return
        <li>
            { string($b/TITLE) } by { string($b/AUTHOR) }
        </li>
    }
</ul>", false)
FROM Dual();
```

Query 2:

```
SELECT XMLQUERY ('C:/books2.xq', true)
FROM Dual();
```

Query 3:

```
SELECT XMLQUERY("<ul>
    { for $b in //BOOKS/ITEM
        order by $b/TITLE return
        <li>
            { string($b/TITLE) } by { string($b/AUTHOR) }
        </li>
    }
</ul>", false, booksxml) from xQuerySampleView;
```

Query 4:

```
SELECT XMLQUERY('C:/books.xq', true, booksxml, false)
FROM xQuerySampleView
```

In ‘Query 1’ the XQuery expression is passed as a parameter and in ‘Query 2’ the parameter is the path to a file containing the same expression. That is why in the ‘Query 2’, the second parameter is `true`. This expression reads the content of the file ‘C:/books_info.xml’.

In ‘Query 3’ and ‘Query 4’ the XML document is obtained from the field `booksxml` of the view `V`.

20.1.4.2 XPATH

Description
Applies an XPath expression on a specific XML type field.

Syntax
```
XPATH(xmlValue:xml, XPathExpression:text [, xmlHeader:boolean]):xml
```

- `xmlValue`. Required. XML data-type which you want to apply the XPath expression on.
• XPathExpression. Required. XPath expression.
• xmlHeader. Optional. If true, the result will include the XML declaration (<?xml version="1.0"...)

Example 1

```sql
SELECT xpath ( cast ('XML', ' <?xml version="1.0" encoding="ISO-8859-1"?> 
  <a>
    <b>Hello</b> 
    <b>World</b>
  </a>' ) , '/a/b[1]/text()' , false) as xpathResults
FROM Dual();
```

<table>
<thead>
<tr>
<th>xpathResults</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
</tr>
</tbody>
</table>

Example 2

```sql
SELECT xpath ( cast ('XML', ' <?xml version="1.0" encoding="ISO-8859-1"?> 
  <a>
    <b>Hello</b> 
    <b>World</b>
  </a>' ) , '/a/b//text()' , false) as xpathResults
FROM Dual();
```

<table>
<thead>
<tr>
<th>xpathResults</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
</tr>
</tbody>
</table>

20.1.4.3 XSLT

**Description**

Returns the result of applying an XSL transformation to an XML.

**Syntax**

```
XSLT(xmlValue:xml, xslValue:xml):xml
XSLT(xmlValue:{xml|text}, xslValue:{xml|text}, [, boolean isPathToXML ] [, boolean isPathToXSLT ]):xml
```

- xmlValue. Required. XML literal, XML field or file to transform.
- xslValue. Required. XSL literal, field containing an XSL or file containing an XSL.
- isPathToXML. Required only if the type of xmlValue is text. true if xmlValue is a path to the xml file. false otherwise.
- isPathToXSLT. Required only if the type of xslValue is text. true if is a path to the xsl file. false otherwise.

**Examples**

Consider the view `v`:

<table>
<thead>
<tr>
<th>xmldata</th>
<th>xslData</th>
</tr>
</thead>
</table>
| <?xml version='1.0' encoding='UTF-8'?>
<shop>
  <products>
    <product>
      <id>1</id>
      <name>Virtual DataPort</name>
    </product>
  </products>
</shop> | <?xml version='1.0' encoding='UTF-8'?><xsl:transform version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match='/'>
    <shop>
      <xsl:for-each select='product'>
    </xsl:template>
  </xsl:transform> |
Example 1

```
SELECT XSLT(xmlsample, xslsample, false, false)
FROM V
```

The same result could be obtained with the following queries:

```
SELECT XSLT(xmlsample, CAST('xml', xslsample), false)
FROM V
SELECT XSLT(CAST('xml', xmlsample), xslsample, false)
FROM V
SELECT XSLT(CAST('xml', xmlsample), CAST('xml', xslsample))
FROM V
```

Example 2

Convert the file `books.xml` using the XSL file `books.xsl`. Note the two last parameters indicating that the first and the second parameters are paths to files.

```
SELECT XSLT ('../test/xml/books.xml', '../test/xml/books.xsl', true, true)
FROM V
```

Example 3

Convert the cells of the columns `xmlsample` using the XSL file `books.xsl`.

```
SELECT XSLT (CAST('xml', xmlsample), '../test/xml/books.xsl', true)
FROM V
```

Example 4

Convert the file `books.xml` using the XSL of the column `xslsample`.

```
SELECT XSLT ('../test/xml/books.xml', CAST('xml', xslsample), true)
FROM V
```
20.1.5  Type Conversion Functions

20.1.5.1  ARRAY_TO_STRING

Description
Converts an array field to a string that contains the elements of the array separated by a character. This function has two signatures. With the first one, the array is surrounded by braces ("{" and "}") and if the array contains other arrays they will also be surrounded by braces. If the array contains registers, they will be surrounded by parentheses ("(" and ")").

With the second signature, the user can indicate the characters that surround the array and its inner registers and arrays.

Syntax
ARRAY_TO_STRING(separator:text, array_value:array):text

ARRAY_TO_STRING(separator:text, array_begin_delimiter:text, array_end_delimiter:text, register_begin_delimiter:text, register_end_delimiter:text,array_value:array):text

• separator. Character that separates the elements of the array.
• array_begin_delimiter. Character placed before the array and its inner arrays.
• array_end_delimiter. Character placed after the array and its inner arrays.
• register_begin_delimiter. Character placed before the inner register fields.
• register_end_delimiter. Character placed after the inner register fields.

Examples
Consider the view \( V \) with an array that has a register in it.

<table>
<thead>
<tr>
<th>name</th>
<th>info</th>
<th>registerSample</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>message</td>
<td>key</td>
</tr>
<tr>
<td>Virtual DataPort</td>
<td>Virtual Data Access Layer</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Data Federation</td>
<td>2</td>
</tr>
<tr>
<td>ITPilot</td>
<td>Web Integration</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Web Automation</td>
<td>4</td>
</tr>
<tr>
<td>Aracne</td>
<td>Crawling</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Quering non-structured data</td>
<td>6</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Job Scheduling</td>
<td>7</td>
</tr>
</tbody>
</table>

Example 1
SELECT name, ARRAY_TO_STRING(' - ', info)
FROM V

<table>
<thead>
<tr>
<th>name</th>
<th>array_to_string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual DataPort</td>
<td>{Virtual Data Access Layer,(1,one) - Data Federation,(2,\text{two})}</td>
</tr>
<tr>
<td>ITPilot</td>
<td>{Web Integration,(3,three) - Web Automation,(4,\text{four})}</td>
</tr>
<tr>
<td>Aracne</td>
<td>{Crawling,(5,five) - Quering non-structured data,(6,\text{six})}</td>
</tr>
<tr>
<td>Scheduler</td>
<td>{Job Scheduling,(7,seven)}</td>
</tr>
</tbody>
</table>
Example 2

```sql
SELECT name, ARRAY_TO_STRING(' ', ' [ ', ' ] ', ' |--- ', '---|', info)
FROM V
```

<table>
<thead>
<tr>
<th>name</th>
<th>array_to_string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual DataPort</td>
<td>[ Virtual Data Access Layer,</td>
</tr>
<tr>
<td>IT Pilot</td>
<td>[ Web Integration,</td>
</tr>
<tr>
<td>Aracne</td>
<td>[ Crawling,</td>
</tr>
<tr>
<td>Scheduler</td>
<td>[ Job Scheduling,</td>
</tr>
</tbody>
</table>

### 20.1.5.2 CAST

**Description**

Converts data from one data-type to another.

Section 3.7.4 contains a table with the possible casting types.

**Syntax**

```
CAST(data_type: text, value: expression)
```

- `data_type`. Required. Data type you want the value to be converted to.
- `value`. Required. The value to convert.

**Example 1**

```sql
SELECT
  CAST('blob', 'hello') AS text_to_blob_cast,
  CAST('boolean', 'true') AS text_to_boolean_cast ,
  CAST('boolean', 500000) AS long_to_boolean_cast ,
  CAST('boolean', 00000) AS long_to_boolean_cast_Zero,
  CAST('double', '5.32') AS text_to_double_cast ,
  CAST('double', 5) AS int_to_double_cast
FROM Dual();
```

<table>
<thead>
<tr>
<th>text_to_blob_cast</th>
<th>Text_to_boolean_cast</th>
<th>long_to_boolean_cast</th>
<th>long_to_boolean_cast_Zero</th>
<th>text_to_double_cast</th>
<th>int_to_double_cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BINARY DATA] - 5 bytes</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>5.32</td>
<td>5.0</td>
</tr>
</tbody>
</table>

**Example 2**

Consider the view `V` with a column `REGISTERSAMPLE` of type `register`. This register has a field `STR` of type `array`.

```sql
SELECT CAST('xml', registerSample)
FROM V
```

<table>
<thead>
<tr>
<th>REGISTERSAMPLE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>R1</td>
</tr>
<tr>
<td>hello</td>
<td>3</td>
</tr>
</tbody>
</table>
Example 3
Consider the view V with a column arraySample of type array. The array arraySample has another array into it.

<table>
<thead>
<tr>
<th>STR</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>denodo</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>platform</td>
<td></td>
<td>52.0</td>
</tr>
<tr>
<td>enterprise</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>data</td>
<td></td>
<td>72.0</td>
</tr>
<tr>
<td>virtualization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```xml
<?xml version="1.0" encoding="UTF-8"?><array>
  <item><R1>40</R1><R2>52.0</R2><STR>denodo</STR><STR>platform</STR></item>
  <item><R1>60</R1><R2>72.0</R2><STR>enterprise</STR><STR>data</STR><STR>virtualization</STR></item>
</array>
```

20.1.5.3 CREATETYPEFROMXML

**Description**
Creates a register or an array type from XML data.
If the type is created correctly, it returns the name of the new type.
For more information see section 3.7.5.1

**Syntax**
CREATETYPEFROMXML(newTypeName:text, xmlValue:{xml|text}):text
- newTypeName. Required. Name of the new type.
- xmlValue. Required. Sample XML used as a template to create the new type.

**Example 1**
Creating a new register type:
Example 2
Creating a new array type:

```sql
SELECT CREATETYPEFROMXML('title_type',
'<!--titles-->
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
<!--/titles-->') FROM Dual();
```
20.1.6 Aggregation Functions

20.1.6.1 AVG

Description
Returns the average of the non-null values of an attribute of the table. If all the values of the attribute are null, the function returns null.

Syntax
AVG(attribute)
- Attribute. Required. The type of the attribute has to be int, long, float, double or money.

Example 1
Consider the following view ITEMS:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3.45</td>
</tr>
<tr>
<td>B</td>
<td>9.99</td>
</tr>
<tr>
<td>C</td>
<td>4.99</td>
</tr>
</tbody>
</table>

```
SELECT AVG(PRICE) AS average_price
FROM ITEMS
```

```
AVERAGE_PRICE
6.1433333333333335
```

Example 2
Consider the following view ITEMS_2:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3.45</td>
</tr>
<tr>
<td>B</td>
<td>9.99</td>
</tr>
<tr>
<td>C</td>
<td>4.99</td>
</tr>
<tr>
<td>D</td>
<td>null</td>
</tr>
</tbody>
</table>

```
SELECT AVG(PRICE) AS average_price
FROM ITEMS_2
```

```
AVERAGE_PRICE
6.1433333333333335
```

20.1.6.2 COUNT

Description
Returns the number of tuples of the result of a selection operation. If the parameter is '*', it returns the number of tuples. If the parameter is an attribute, it returns the number of non-null values.

Syntax
COUNT(param)
- Param. Required. It can be either an attribute or '*'.

Examples
Consider the following view ITEMS:
ITEM | PRICE
---|---
A | 3.45
B | 9.99
C | 4.99
D | null

**Example 1**

```sql
SELECT COUNT(*)
FROM ITEMS
```

<table>
<thead>
<tr>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

**Example 2**

```sql
SELECT COUNT(price)
FROM ITEMS
```

<table>
<thead>
<tr>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

### 20.1.6.3  FIRST

**Description**

Returns the first value of an attribute for each group of values.

**Syntax**

```sql
FIRST (attribute)
```

- Attribute. Required. An attribute of the view.

**Examples**

Consider the following view `V`:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>one</td>
</tr>
<tr>
<td>group1</td>
<td>two</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>

**Example 1**

```sql
SELECT FIRST(B)
FROM V
```

<table>
<thead>
<tr>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
</tr>
</tbody>
</table>

**Example 2**

```sql
SELECT A, FIRST(B)
FROM V GROUP BY A
```

<table>
<thead>
<tr>
<th>A</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>one</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>
20.1.6.4 GROUP_CONCAT

**Description**
Returns, for each group, a string with the concatenation of all the field/fields values of each group.

**Syntax**

GROUP_CONCAT( [ rowSeparator:text [, fieldSeparator:text ] ]
, field [, field]*)
GROUP_CONCAT(ignoreNull, rowSeparator:text, fieldSeparator:text
, field [, field]*)

- IgnoreNulls. Optional. If true and any of the fields field of a row are NULL, GROUP_CONCAT ignores all the fields of that row. If false, no rows are ignored and NULL values are treated as empty characters. The default value is true.
- RowSeparator. Optional. Literal used to separate the values of each row. Default value: ','.
- FieldSeparator. Optional. Literal used to separate the values of the fields of the same row. The default value is a whitespace.
- Field. Required. Field which contains the values to concatenate.

**Examples**
Consider the following view V:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>group1</td>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
<td>three</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
<td>four</td>
</tr>
</tbody>
</table>

**Example 1**
SELECT A, group_concat(':', C)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>GROUP_CONCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>one:two:three</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>

**Example 2**
SELECT A, group_concat(':', ';', B, C)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>GROUP_CONCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1;one:2;two</td>
</tr>
<tr>
<td>group2</td>
<td>4;four</td>
</tr>
</tbody>
</table>

As the field B is NULL in the third row, GROUP_CONCAT ignores all the fields of that row. That is, it ignores the value “three”.

**Example 3**
SELECT A, group_concat(false, ':', ';', B, C)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>GROUP_CONCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1;one:2;two;three</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>
As the parameter `ignoreNulls` is false, `GROUP_CONCAT` does not ignore the value of the field `C` of the third row ("three"), even if the value of the field `B` of that row is `NULL`. In this case, `NULL` values are treated like empty characters.

20.1.6.5 LAST

**Description**
Returns the last value of an attribute for each group of values.

**Syntax**
LAST (attribute)
- Attribute. Required. Field name of the view.

**Examples**
Consider the following view `V`:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>group1</td>
<td>one</td>
</tr>
<tr>
<td>group1</td>
<td>two</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>

**Example 1**

SELECT LAST(B)
FROM V

LAST
four

**Example 2**

SELECT A, LAST(B)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th></th>
<th>LAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>

20.1.6.6 LIST

**Description**
Returns an array with all the values of a specified attribute.

**Syntax**
LIST (attribute)
- Attribute. Required. Attribute of the view.

**Examples**
Consider the following view `V`:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>group1</td>
<td>one</td>
</tr>
<tr>
<td>group1</td>
<td>two</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>four</td>
</tr>
</tbody>
</table>
Example 1
SELECT LIST(B)
FROM V

Example 2
SELECT A, LIST(B)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>Array { one, two, null }</td>
</tr>
<tr>
<td>group2</td>
<td>Array { four }</td>
</tr>
</tbody>
</table>

20.1.6.7 MAX

Description
Returns the highest value of an attribute for each group of values.

Syntax
MAX (attribute)
- Attribute. Required. Attribute of type date, int, long, float, double or money.

Examples
Consider the following view V:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1</td>
</tr>
<tr>
<td>group1</td>
<td>2</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
</tr>
</tbody>
</table>

Example 1
SELECT MAX(B)
FROM V

MAX
4

Example 2
SELECT A, MAX(B)
FROM V GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>2</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
</tr>
</tbody>
</table>

20.1.6.8 MIN

Description
Returns the lowest value of an attribute for each group of values.
Syntax
MIN (attribute)
- Attribute. Required. Attribute of type int, long, float, double, date or money.

Examples
Consider the following view V:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1</td>
</tr>
<tr>
<td>group1</td>
<td>2</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
</tr>
</tbody>
</table>

Example 1
SELECT MIN(B) FROM V

Example 2
SELECT A, MIN(B) FROM V GROUP BY A

20.1.6.9 NEST

Description
Returns an array with the values of the selected fields. Its result is inverse to the result of the FLATTEN views (see section 5.1.2 for more information about FLATTEN views)

Syntax
NEST(field_name:any_type [, field_name:any_type ]*:array
NEST(*)
- Field_name. The name of a field. Using (*) is equivalent to pass all the fields of the view to the function.

Example
Consider the view V:

<table>
<thead>
<tr>
<th>intsample</th>
<th>textsample</th>
<th>registersample</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Register { hello, how’re you }</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>Register { hello, good bye }</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>Register { another string, last string }</td>
</tr>
</tbody>
</table>

SELECT intsample, NEST(textsample, registersample) AS nestsample FROM V GROUP BY intsample;
## 20.1.6.10 SUM

### Description
Returns the sum of all non-null values of an attribute for each group of values.

### Syntax
```
SUM (attribute)
```
- Attribute. Required. Attribute of type `int`, `long`, `float`, `double` or `money`.

### Examples
Consider the following view `V`:

| A    | B
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1</td>
</tr>
<tr>
<td>group1</td>
<td>2</td>
</tr>
<tr>
<td>group1</td>
<td>null</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Example 1**
```
SELECT SUM(B)
FROM V
```
```
SUM
7
```

**Example 2**
```
SELECT A, MIN(B)
FROM V GROUP BY A
```

<table>
<thead>
<tr>
<th>A</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>3</td>
</tr>
<tr>
<td>group2</td>
<td>4</td>
</tr>
</tbody>
</table>

## 20.1.7 Other Functions

### 20.1.7.1 COALESCE

### Description
Returns the first non-null argument. COALESCE is equivalent to the expression:
```
CASE WHEN arg1 IS NOT NULL THEN arg1
WHEN arg2 IS NOT NULL THEN arg2
...
END
```

### Syntax
```
COALESCE(<param>, <param> [, <paramN>]*)
```
- param. Text or field name which can be null.
Examples
Consider the view called V:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.10</td>
<td>I am some text</td>
<td>21-ene-2005 0h 0m 0s</td>
</tr>
<tr>
<td>-80.10</td>
<td>Text is $% needed always</td>
<td>12-mar-2005 12h 30m 0s</td>
</tr>
<tr>
<td>20.50</td>
<td>Text for a living</td>
<td>01-feb-2006 16h 45m 0s</td>
</tr>
<tr>
<td>40.05</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Example 1
SELECT coalesce(B, 'hello')
FROM V

COALESCE
I am some text
Text is $% needed always
Text for a living
hello

Example 2
SELECT coalesce('hello', 'bye')
FROM V

COALESCE
hello
hello
hello
hello
hello

Example 3
SELECT coalesce(B, A)
FROM V

COALESCE
I am some text
Text is $% needed always
Text for a living
40.05

20.1.7.2 CONTEXTUALSUMMARY

Description
Returns relevant text fragments of a text, containing the word or sentence specified.

Syntax
CONTEXTUALSUMMARY(content:text, keyword:text, [beginDelim:text, endDelim:text, fragmentSeparator:text, fragmentLength:int [,maxFragmentsNumber:int [,analyzer:text]]])

- content: Required. Text that the most relevant fragments are to be extracted from.
- keyword: Required. Keyword used to extract the text fragments. The content of this argument can be a single word or a sentence.
• **beginDelim**: Optional. Text to add as prefix of the keyword whenever it appears in the text. Default value is "".
• **endDelim**: Optional. Text to add as suffix of the keyword whenever it appears in the text. Default value is "".
• **fragmentSeparator**: Optional. Text to separate each text fragment of the result. Default value is "...".
• **fragmentLength**: Optional. Approximate number of characters that will appear before and after the keyword occurrences inside of the text. Default value is 5.
• **maxFragmentNumber**: Optional. Maximum number of fragments to retrieve.
• **analyzer**: Optional. Analyzer used to search for keywords. By default, the Standard Analyzer (std) is used. Virtual DataPort also includes analyzers for English (en) and Spanish (es).

**Example 1**

```sql
SELECT contextualsummary(content, 'Denodo', '<b>', '</b>', ' ... ', 5, 1)
FROM demo_arn_view;
```

This query will return fragments of text `content` where the "Denodo" word appears.

**Example 2**

Consider the following view `TextSummarySample`:

<table>
<thead>
<tr>
<th>TEXTSAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A web service (also webservice) is defined by the W3C as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards. Web services are frequently just Internet Application Programming Interfaces (API) that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services. Other approaches with nearly the same functionality as web services are Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM) or Sun Microsystems's Java/Remote Method Invocation (RMI).</td>
</tr>
</tbody>
</table>

```sql
SELECT contextualsummary(textsample,'system') FROM textsummarysample;
```

**Example 3**

```sql
SELECT contextualsummary(textsample,'service') FROM textsummarysample;
```
Example 4

```
SELECT contextualsummary(textsample,'web', '\\', '/', '--', 25) FROM textsummarysample
```

Contextual summary

A \web/ service (also-- (specifically \Web/ Services-- with the \web/ service with other \web/-related as \web/ services

20.1.7.3 GETSESSION

**Description**
Provides information about the session established with a Virtual DataPort server.

**Syntax**

```
GETSESSION( <value:session_info> ):text
session_info::= 'user' | 'database' | 'i18n';
```

If 'value' is 'user', the function returns the user name of the current client.
If 'value' is 'database', it returns the Virtual DataPort database that the client is connected to.
If 'value' is 'i18n', it returns the i18N (internationalization) configuration of the database that the client is connected to.

**Example 1**

```
SELECT GETSESSION('user') AS user_name
FROM Dual()
```

<table>
<thead>
<tr>
<th>user_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin</td>
</tr>
</tbody>
</table>

**Example 2**

```
SELECT GETSESSION('database') AS db
FROM Dual()
```

<table>
<thead>
<tr>
<th>db</th>
</tr>
</thead>
<tbody>
<tr>
<td>denodo_samples_db</td>
</tr>
</tbody>
</table>

**Example 3**

```
SELECT GETSESSION('i18n') AS i18n
FROM Dual()
```

<table>
<thead>
<tr>
<th>i18n</th>
</tr>
</thead>
<tbody>
<tr>
<td>us_pst</td>
</tr>
</tbody>
</table>
20.1.7.4 HASH

**Description**
Returns the MD5 hash of a text.

**Syntax**
```
HASH(value: text): text
```
- `value`: Required. The name of a field or a literal.

**Examples**
Consider the view `V`:

<table>
<thead>
<tr>
<th>A</th>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am some text</td>
<td>sIzgqar3ATVqDiEhMwPneg==</td>
</tr>
<tr>
<td>Text is $% needed always</td>
<td>BBuXj7Xc9rsolLrPurEDKg==</td>
</tr>
<tr>
<td>Text for a living</td>
<td>oBDHVUsIz9pWH0UBev040g==</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

**Example 1**

```
SELECT A, hash(A)
FROM V
```

<table>
<thead>
<tr>
<th>A</th>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am some text</td>
<td>sIzgqar3ATVqDiEhMwPneg==</td>
</tr>
<tr>
<td>Text is $% needed always</td>
<td>BBuXj7Xc9rsolLrPurEDKg==</td>
</tr>
<tr>
<td>Text for a living</td>
<td>oBDHVUsIz9pWH0UBev040g==</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

**Example 2**

```
SELECT hash('hello')
FROM V
```

<table>
<thead>
<tr>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>XUFAKrxLKna5cZ2REBfFkg==</td>
</tr>
<tr>
<td>XUFAKrxLKna5cZ2REBfFkg==</td>
</tr>
<tr>
<td>XUFAKrxLKna5cZ2REBfFkg==</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

20.1.7.5 IS_PROJECTED_FIELD

**Description**
Returns `true` if the field passed as parameter is projected in the view. `false` otherwise.

**Syntax**
```
IS_PROJECTED_FIELD(fieldName: text): boolean
```
- `fieldName`: Required. The name of the field.

**Examples**
Consider the following view `ITEMS`:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3.45</td>
</tr>
<tr>
<td>B</td>
<td>9.99</td>
</tr>
<tr>
<td>C</td>
<td>4.99</td>
</tr>
</tbody>
</table>
Example 1

```sql
SELECT IS_PROJECTED_FIELD('ITEM') AS 
FROM ITEMS
```

<table>
<thead>
<tr>
<th>IS_PROJECTED_FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

Example 2

```sql
SELECT IS_PROJECTED_FIELD(ITEM) AS 
FROM ITEMS
```

<table>
<thead>
<tr>
<th>IS_PROJECTED_FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>false</td>
</tr>
</tbody>
</table>

In the second example, `IS_PROJECTED_FIELD` returns `false` because the parameter is the value of the cell 'ITEM' of each row. While in the first example it returns `true` because the parameter is a literal with the name of the field.

20.1.7.6 MAP

**Description**

Returns the value associated with a key or `null`. The pair key-value can be obtained from a view or from a Map (see section 10.2). When the key doesn’t exist, the function returns `null`.

There are two possible signatures:

**Syntax 1**

```sql
MAP (<key:text>, <view_name:text>, <key_field:text>, <value_field:text>)
```

It obtains the value associated with a key. MAP searches the value of a key in the columns of a view.

- `key`. Required. The value to search in the view.
- `view_name`. Required. The name of the view that contains the key and its value.
- `key_field`. Required. The column of the view that contains the keys.
- `value_field`. Required. The column of the view that contains the values.

**Syntax 2**

```sql
MAP (<key:text>, <map_name:text> [, <i18n:text> ] )
```

It obtains the value associated with a key from a Map.

- `key`. Required. The value to search.
- `map_name`. Required. The name of the map that contains the key and its value.
- `i18n`. Optional. Internationalization configuration of the contents.

**Note**: In both cases, `key` is a case-insensitive parameter.

**Example 1**

Consider the map `food`:

```sql
CREATE MAP SIMPLE food ( 
  'breakfast' = 'milk' 
  'dinner' = 'lettuce' 
  'lunch' = 'meat' 
);
```
```sql
SELECT map('breakfast', 'food', 'gr') AS breakfast,
      map('lunch', 'food', 'gr') AS lunch,
      map('dinner', 'food', 'gr') AS dinner,
      map('none', 'food', 'gr') AS none
FROM V
```

### Example 2
Consider the view `FOREIGN_SALES` that contains the revenue of a company in each country, in the country’s currency.

<table>
<thead>
<tr>
<th>Country</th>
<th>Month</th>
<th>Revenue</th>
<th>Currency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mexico</td>
<td>JAN</td>
<td>7536.00</td>
<td>MXN</td>
</tr>
<tr>
<td>Spain</td>
<td>JAN</td>
<td>20000.00</td>
<td>EUR</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>JAN</td>
<td>26816.00</td>
<td>GBP</td>
</tr>
<tr>
<td>Canada</td>
<td>FEB</td>
<td>-25616.00</td>
<td>CAD</td>
</tr>
<tr>
<td>Japan</td>
<td>FEB</td>
<td>100024.00</td>
<td>JPY</td>
</tr>
</tbody>
</table>

And the Map `CURRENCY_RATES_TO_USD` that contains the exchange rate of each currency to dollar.

```sql
CREATE MAP SIMPLE currency_rates_to_usd (  
    'CAD' = '0.957121'  
    'EUR' = '1.4971'  
    'GBP' = '1.67'  
    'JPY' = '0.011166'  
    'MXN' = '0.076989'  
    'USD' = '1.0'  
);
```

```sql
SELECT month, country, CAST('float', MAP(currency,  
    'currency_rates_to_usd')) * revenue AS REVENUE_USD FROM foreign_sales
```

### 20.1.7.7 NULLIF

**Description**

Compares two values or expressions and returns NULL if they are equal. Otherwise it returns the first value.

**Syntax**

`NULLIF(<expression1>, <expression2>)`
Examples
Consider the view internet_inc:

<table>
<thead>
<tr>
<th>ID</th>
<th>SUMMARY</th>
<th>TTIME</th>
<th>TAXID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596011</td>
</tr>
<tr>
<td>2</td>
<td>Incidence in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596012</td>
</tr>
<tr>
<td>3</td>
<td>Install additional line</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596013</td>
</tr>
<tr>
<td>4</td>
<td>Bandwidth increase</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596014</td>
</tr>
</tbody>
</table>

Example 1

```sql
SELECT NULLIF(ID, 1) AS Display FROM internet_inc
```

<table>
<thead>
<tr>
<th>DISPLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Example 2

```sql
SELECT * FROM internet_inc WHERE NULLIF(ID, 1) <> NULL
```

<table>
<thead>
<tr>
<th>ID</th>
<th>SUMMARY</th>
<th>TTIME</th>
<th>TAXID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Incidence in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596012</td>
</tr>
<tr>
<td>3</td>
<td>Install additional line</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596013</td>
</tr>
<tr>
<td>4</td>
<td>Bandwidth increase</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596014</td>
</tr>
</tbody>
</table>

Example 3

```sql
SELECT NULLIF ('      ','  ') AS Display FROM internet_inc;
```

<table>
<thead>
<tr>
<th>DISPLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

Note: NULLIF has removed the leading and trailing whitespaces of the parameters.

Example 4

```sql
SELECT COALESCE(NULLIF(ID,'1'), summary) AS Display FROM internet_inc
```

<table>
<thead>
<tr>
<th>DISPLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in ADSL router</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Note: NULLIF has automatically converted the second parameter to an integer to compare it with the values of the column ID which are also integers.

20.2 SYNTAX OF SEARCH EXPRESSIONS FOR THE CONTAINS OPERATOR

This section describes the syntax of search expressions for the DataPort contains operator.
20.2.1 Exact Terms and Phrases

A query is made up of terms and operators. There are two types of terms: Individual Terms and Exact Phrases.

An Individual Term is a single word. A phrase is a group of words between double inverted commas. Terms may be combined using Boolean operators to form complex queries (see below).

20.2.2 Term Modifiers

The use of the following modifiers is accepted:

20.2.2.1 Search Wildcards

The symbol "?" replaces ? for a single character in the word. The symbol "*" replaces * for 0 or more characters. For example, if you want to search for "information" or "informative", the following term could be entered:

```
inform*
```

20.2.2.2 Fuzzy Searches

Fuzzy searches are allowed (sources may implement this function using string editing distance techniques, for example). To make fuzzy searches, the symbol "~" must be used at the end of a simple term. For example, to search for terms written in a manner similar to "card", the following fuzzy search would be used:

```
card~
```

This would find terms such as "cad".

A parameter (optional) can be added to specify the minimum similarity required. For example:

```
card~0.8
```

20.2.2.3 Proximity Searches

Searches for terms among which there is a certain spatial proximity are allowed. To implement these, use the symbol "~" at the end of an exact phrase. The maximum number of words to separate the terms can also be specified. For example, to search for "denodo" and "technologies" with a distance of up to 8 words in the same document, the following search would be used:

```
"denodo technologies"~8
```

20.2.2.4 Range Searches

Range searches allow for documents with values within a certain range to be retrieved. The range specified may or may not include the upper and lower limits. Inclusive ranges are specified using square brackets and exclusive ranges using curly brackets. The classification follows the lexicographic order. For example:

```
[20020101 TO 20030101]
```
This query finds documents with a value of between 20020101 and 20030101, inclusively. The range search is not limited to the fields containing dates as the value:

{Aida TO Carmen}

This query retrieves all documents with titles found between Aida and Carmen, not inclusively.

20.2.2.5 Boosting the Relevance Level of a Term

It is possible to boost the weight of a term in the search when calculating the level of relevance using the symbol "^" with a boosting factor (a number) at the end of the search term. The higher the factor, the more relevant the term in the search.

This allows for the relevance of a document to be controlled by boosting the relevance level of its terms. For example, if you want to search for denodo technologies and the term "denodo" is to be the most relevant, you would use the symbol ^ with a relevance level boosting factor alongside the term:

denodo^4 technologies

This ensures that the documents containing the term "denodo" are most relevant for the search. This technique can also be used with phrases.

The default relevance factor is 1. This must be a positive number, although it may be less than 1 (for example, 0.2).

20.2.3 Boolean Operators

Boolean operators allow combining terms using logic operators. The following Boolean operators are accepted: AND, OR, and NOT (Note: Boolean operators must be written in upper-case letters.).

20.2.4 Groups

The use of brackets is allowed. For example, to search for "Corp" or "Inc" and "Denodo", the following query would be used:

(Corp OR Inc) AND denodo

20.2.5 Escaping Special Characters

The list of special characters is:
To escape these characters, use \ before the character.

## 20.3 Support for the contains Operator of Each Source Type

The syntax of the search language on non-structured data used with the contains operator is described in section 20.2. However, bear in mind that the search options available depend on the capacities natively provided by the data source. For example, Google Enterprise / Google Mini do not support different characteristics of the search language such as proximity searches. Therefore, when the contains operator is used with attributes from those sources, these capacities will not be available.

This section provides exact details as to the search capacities supported for each source type. These capacities are also specified in the Configuration Properties of each data source (see section 18.3.13.1) that can be consulted using the DESC VIEW statement (see section 12).

The data sources Aracne-type (see section 18.3.8), Google Mini (see 18.3.9) and Custom (see 18.3.12) can use the contains operator.

The following sections describe the capacities supported for Aracne and Google Mini wrappers, respectively. Custom-type wrappers can specify the capacities supported through the Configuration Properties (see section 18.3.13.1 and section 12).

### 20.3.1 Aracne

The following characteristics of the contains operator search language are not supported in Denodo Aracne-type sources:

- The wildcards ? and * cannot appear in the first position of a term.
- Searches using the proximity operator ~ must specify the maximum number of words that can separate the terms of the phrase.
- The logic operator NOT must appear at the same level as a logic operator AND. Example: The search (term1 AND NOT term2) would work correctly, but not the search (term1 OR NOT term2).

The remaining capacities of the search language are supported in Denodo Aracne-type sources.

### 20.3.2 Google Enterprise / Google Mini

The following characteristics of the contains operator search language are not supported in Google Enterprise/Mini sources:

- Searches by exact phrase are not supported in the site attribute. They are supported, however, in the remaining attributes.
- Wildcards, fuzzy searches, proximity searches, searches with relevance boost and range searches are not supported.
- Searches with the logic operators AND, OR, and NOT in the title, url, and site attributes are only valid, if the conditions are simple terms or exact phrases (i.e. logic conditions cannot be nested in searches on these attributes). This restriction does not exist for the remaining attributes.
- The logic operator NOT must appear at the same level as a logic operator AND. Example: The search (term1 AND NOT term2) would work correctly, but not the search (term1 OR NOT term2).
20.4 CASE CLAUSE EXAMPLES

Consider the following Virtual DataPort view named internet_inc:

<table>
<thead>
<tr>
<th>ID</th>
<th>SUMMARY</th>
<th>TIME</th>
<th>TAXID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596011</td>
</tr>
<tr>
<td>2</td>
<td>Incidence in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596012</td>
</tr>
<tr>
<td>3</td>
<td>Install additional line</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596013</td>
</tr>
<tr>
<td>4</td>
<td>Bandwidth increase</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596014</td>
</tr>
</tbody>
</table>

Example 1

```sql
SELECT id, summary,
    CASE
        WHEN LEN(summary) > 22 THEN summary
    ELSE id
    END
FROM internet_inc
```

Example 2

```sql
SELECT id,
    CASE
        WHEN id = 1 THEN true
    ELSE id
    END AS isFirst
FROM internet_inc
```

Error executing sentence: Incorrect select sentence: CASE argument IINC_ID is not compatible with the rest of values.

The type of the result of the WHEN clause is incompatible with the one of the ELSE clause. The first one has type `boolean` and the other, `long`.

Example 3

```sql
SELECT id,
    CASE
        WHEN id = 1 THEN "first"
    ELSE id
    END AS isFirst
FROM internet_inc
```

<table>
<thead>
<tr>
<th>ID</th>
<th>isFirst</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>first</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Note: If the type of the results of the WHEN or ELSE clauses are not the same, they are automatically converted to obtain a valid result. In this case the results are converted to `String`. 
Example 4
The CASE clause can also be used in the WHERE part of a query.

```sql
SELECT * FROM internet_inc
WHERE true = (CASE id
    WHEN 1 THEN true
    ELSE false
END)
```

<table>
<thead>
<tr>
<th>ID</th>
<th>SUMMARY</th>
<th>TTIME</th>
<th>TAXID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error in ADSL router</td>
<td>2005-06-29 19:19:41.0</td>
<td>B78596011</td>
</tr>
</tbody>
</table>

Example 5
These two queries are equivalent and obtain the same result, but use CASE in different ways:

```sql
SELECT id,
    CASE id
        WHEN CASE id WHEN 1 THEN 1
            ELSE 2
        END THEN "first"
        WHEN 2 THEN "second"
        ELSE "other"
    END
FROM internet_inc;
```

```sql
SELECT id,
    CASE id
        WHEN CASE WHEN id = 1 THEN 1
            ELSE 2
        END THEN "first"
        WHEN "second"
        WHEN "other"
    END
FROM internet_inc;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>first</td>
</tr>
<tr>
<td>2</td>
<td>first</td>
</tr>
<tr>
<td>3</td>
<td>other</td>
</tr>
<tr>
<td>4</td>
<td>other</td>
</tr>
</tbody>
</table>

**Note:** CASE returns the result of the first WHEN clause that evaluates to true. In this example, the first and second WHEN conditions are true, but it returns the result of the first one.

20.5 DATE AND TIME PATTERN STRINGS
VirtualDataPort uses the date and time Java patterns [JAVADATEFORMAT] to specify date and time formats. In these patterns, the letters of the first column represent parts of a date.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Date or Time Component</th>
<th>Presentation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Era designator</td>
<td>Text</td>
<td>AD</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>Year</td>
<td>1996; 96</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Month in year</td>
<td>July; Jul; 07</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>Week in year</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>Week in month</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Day in year</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Day in month</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Day of week in month</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Day in week</td>
<td>Tuesday; Tue</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>Am/pm marker</td>
<td>PM</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Hour in day (0-23)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Hour in day (1-24)</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Hour in am/pm (0-11)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>Hour in am/pm (1-12)</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>Minute in hour</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Second in minute</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Millisecond</td>
<td>978</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>Time zone</td>
<td>Pacific Standard Time; PST; GMT-08:00</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Time zone</td>
<td>-800</td>
<td></td>
</tr>
</tbody>
</table>

Java Date and time patterns used in VirtualDataPort
REFERENCES

[JAXRPC] JAX-RPC. https://jax-rpc.dev.java.net/
[JDKJAVADOC] Javadoc documentation of the Java Developer Kit 6 Standard API.
[JSON] JSON (JavaScript Object Notation) http://www.json.org/
[REEXP] Regular expressions in JAVA. http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

References 238
[SOAP_JMS] SOAP over Java Message Service (JMS). http://www.w3.org/TR/soapjms/


[XPATH] Xpath Language. http://www.w3.org/TR/xpath/


[XSLT] XSL Transformations (XSLT). http://www.w3.org/TR/xslt