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

**PREFACE** ........................................................................................................................................... I  
**SCOPE** ............................................................................................................................................... I  
**WHO SHOULD USE THIS DOCUMENT** ............................................................................................. I  
**SUMMARY OF CONTENTS** .................................................................................................................. I  

1** INTRODUCTION........................................................................................................................................ 1  

2 **GENERAL OVERVIEW OF VIRTUAL DATAPORT** ............................................................................. 2  
   2.1 **CREATING OR DEFINING DATA** .................................................................................................. 2  
      2.1.1 Defining base relations ....................................................................................................... 2  
      2.1.2 Defining datasources and wrappers .................................................................................... 4  
      2.1.3 Defining the views of the global schema ........................................................................... 5  
   2.2 **EXECUTING STATEMENTS** ..................................................................................................... 6  

3 **CHARACTERISTICS** .......................................................................................................................... 7  
   3.1 **DATA TYPES** ......................................................................................................................... 7  
   3.2 **INTERNATIONALIZATION** ..................................................................................................... 8  
   3.3 **QUERY CAPABILITIES** .......................................................................................................... 9  
   3.4 **REWITING RULES** ................................................................................................................. 9  

4 **LANGUAGE FOR DEFINING AND PROCESSING DATA: VQL** .......................................................... 10  
   4.1 **STATEMENTS** ....................................................................................................................... 10  
   4.2 **SELECT STATEMENT: CLAUSES** .......................................................................................... 11  
   4.3 **INSERT / UPDATE /DELETE: CLAUSES** ............................................................................... 11  
   4.4 **LOGICAL OPERATORS** .......................................................................................................... 12  
   4.5 **COMPARISON OPERATORS** ................................................................................................. 12  
   4.6 **DERIVED ATTRIBUTE AND CONDITION FUNCTIONS** ......................................................... 15  
      4.6.1 Arithmetic Functions ........................................................................................................... 16  
      4.6.2 Text Processing Functions .................................................................................................. 17  
      4.6.3 Date Processing functions .................................................................................................. 19  
      4.6.4 Type Conversion Functions ............................................................................................... 20  
      4.6.5 XML Processing Functions ............................................................................................... 21  
      4.6.6 Other functions .................................................................................................................. 22  
   4.7 **AGGREGATION FUNCTIONS** ................................................................................................... 23  
   4.8 **SYNTAX CONVENTIONS** ....................................................................................................... 23  
      4.8.1 Syntax of functions and condition values ............................................................................ 25  

5 **CREATING A BASE RELATION (OR BASE VIEW)** .......................................................................... 28  
   5.1 **MODIFYING A BASE RELATION** ............................................................................................ 28  
   5.2 **QUERY CAPABILITIES: SEARCH METHODS AND WRAPPERS** ........................................ 30  
      5.2.1 Query Constraints ............................................................................................................... 30  
      5.2.2 Assigning Wrappers to Search methods ............................................................................ 31  
      5.2.3 Example of How a Search Method is Created ................................................................... 31  
   5.3 **REWITING RULES** .................................................................................................................. 31  
      5.3.1 Input Rewriting Rules ......................................................................................................... 32  
      5.3.2 Output Rewriting Rules ..................................................................................................... 34
6 QUERIES: SELECT STATEMENT ................................................................. 37
   6.1 FROM CLAUSE .................................................................................. 39
       6.1.1 Join operations ........................................................................... 39
       6.1.2 FLATTEN VIEW (flattening data structures) ................................. 40
   6.2 SELECT CLAUSE ............................................................................... 41
       6.2.1 Derived Attributes ..................................................................... 41
   6.3 WHERE CLAUSE ............................................................................... 42
       6.3.1 Conditions with compound values .............................................. 43
   6.4 GROUP BY CLAUSE .......................................................................... 43
       6.4.1 Use of Aggregation Functions ..................................................... 44
   6.5 HAVING CLAUSE ............................................................................... 44
   6.6 UNION CLAUSE ................................................................................ 44
       6.6.1 Specifying projections in UNION queries ..................................... 44
   6.7 ORDER BY CLAUSE .......................................................................... 45
   6.8 CONTEXT CLAUSE .......................................................................... 45
   6.9 TRACE CLAUSE ................................................................................. 46

7 DEFINING A DERIVED VIEW ................................................................. 49
   7.1 MODIFYING A DERIVED VIEW ......................................................... 50
       7.1.1 Rewriting Rules on Derived Views ............................................. 51

8 INSERTIONS, UPDATES AND DELETION OF VIEWS .............................. 53
   8.1 INSERT STATEMENT ......................................................................... 53
   8.2 UPDATE STATEMENT ........................................................................ 54
   8.3 DELETE STATEMENT ......................................................................... 55
   8.4 USE OF THE WITH CHECK OPTION ............................................... 55

9 TRANSACTIONS IN DATAPORT .............................................................. 57

10 STORED PROCEDURES ......................................................................... 58
   10.1 IMPORTING A STORED PROCEDURE .............................................. 58
   10.2 USE OF STORED PROCEDURES ...................................................... 58
   10.3 CREATION OF NEW STORED PROCEDURES ..................................... 59
   10.4 PRE-DEFINED PROCEDURES ......................................................... 61

11 DEFINING OTHER ELEMENTS OF THE CATALOG ............................... 62
   11.1 DEFINING A DATA TYPE ............................................................... 62
   11.2 DEFINITION OF A MAP .................................................................. 63

12 CREATING DATABASES, USERS AND ACCESS LEVELS .......................... 65
   12.1 DATABASES IN VIRTUAL DATAPORT .............................................. 65
   12.2 USER AND ACCESS STRUCTURES IN VIRTUAL DATAPORT .............. 65
       12.2.1 Types of users .......................................................................... 65
       12.2.2 Types of access rights ............................................................... 65
   12.3 VQL STATEMENTS OF DATABASES, USERS AND ACCESS .............. 67
       12.3.1 Creating Databases ................................................................. 67
       12.3.2 Modifying and Deleting Databases .......................................... 67
       12.3.3 Creating users ......................................................................... 68
       12.3.4 Modifying and Deleting users .................................................. 69
FIGURES

Figure 1  Search form of a virtual bookshop on the Internet ................................................................. 3
Figure 2  Search method for a bookshop ................................................................................................ 4
Figure 3  Basic primitives for specifying VQL statements ................................................................. 25
Figure 4  Rules for forming functions .................................................................................................. 26
Figure 5  Syntax of the statement CREATE TABLE ........................................................................... 28
Figure 6  Example of creating a base view ........................................................................................... 28
Figure 7  Syntax of the statement ALTER TABLE ........................................................................... 30
Figure 8  Example of how a search method is created with ALTER TABLE ...................................... 31
Figure 9  Adding an input rewriting rule REGEXP() ........................................................................ 33
Figure 10 Adding a rewriting rule on MAPI conditions ...................................................................... 34
Figure 11 Adding an input rewriting CHANGEOPERATOR () .......................................................... 34
Figure 12 Adding an output rewriting rule with the function MAPI() ................................................ 35
Figure 13 Syntax of the SELECT statement ........................................................................................ 38
Figure 14 Syntax of a FLATTEN view ................................................................................................ 41
Figure 15 Syntax for a list of conditions .............................................................................................. 42
Figure 16 Syntax for a projection of the result of an union .................................................................. 45
Figure 17 Syntax of the CONTEXT clause .......................................................................................... 46
Figure 18 Execution trace ................................................................................................................... 48
Figure 19 Syntax of the statement CREATE VIEW ........................................................................... 49
Figure 20 Example of how a view is defined in accordance with others .......................................... 49
Figure 21 Syntax of the statement ALTER VIEW ............................................................................ 51
Figure 22 Adding an input rewriting rule for a derived view ............................................................ 52
Figure 23 Deleting an output rewriting rule from a non-base view .................................................. 52
Figure 24 Adding a results rewriting in a derived view ...................................................................... 52
Figure 25 Syntax of the INSERT statement ......................................................................................... 53
Figure 26 Syntax of the UPDATE statement ....................................................................................... 54
Figure 27 Syntax of the DELETE statement ......................................................................................... 55
Figure 28 CREATE PROCEDURE syntax ........................................................................................... 58
Figure 29 ALTER PROCEDURE syntax ............................................................................................ 58
Figure 30 Syntax of the CALL statement ........................................................................................... 58
Figure 31 Syntax of the statement CREATE TYPE ............................................................................. 62
Figure 32 Creating an enumerated data type ..................................................................................... 62
Figure 33 Creating a register data type ............................................................................................. 63
Figure 34 Creating a data type array and the register type it contains .............................................. 63
Figure 35 Syntax of the statement CREATE MAP.............................................................................. 63
Figure 36 Creation of an map of type inputrewriter ........................................................................... 64
Figure 37 Syntax of the CREATE DATABASE statement ............................................................... 67
Figure 38 Simplified syntax of the ALTER DATABASE statement .................................................... 68
Figure 39 Syntax of the CREATE USER statement ........................................................................... 68
Figure 40 Syntax of the ALTER USER statement ............................................................................ 69
Figure 41 Syntax of the CONNECT and CLOSE statements .............................................................. 69
Figure 42 Syntax of the GRANT/REVOKE clauses for Databases ..................................................... 70
Figure 43 Syntax of the clauses GRANT/REVOKE for Databases ..................................................... 71
Figure 44 Syntax of the clauses GRANT/REVOKE for views ............................................................. 72
Figure 45 Example of assigning privileges to users ........................................................................... 72
Figure 46 Syntax of the statement DESC ........................................................................................ 73
Figure 47 Syntax of the statement LIST ............................................................................................ 75
Figure 48 Syntax of the statement DROP ........................................................................................ 77
Figure 49 Syntax of the statement CLEAR CACHE ........................................................................ 79
Figure 50 Syntax of the statement HELP ......................................................................................... 80
Figure 51 Syntax to request help on the command ALTER TABLE .................................................. 80
TABLES

Table 1  Type conversions permitted with the CAST function .................................................. 20
Table 2  Automatic conversions between JAVA types and Virtual DataPort types ..................... 82
Table 3  Other valid conversions between JAVA types and Virtual DataPort types ..................... 82
Table 4  Type Conversion Tables for JDBC Wrappers ............................................................. 83
Table 5  Type Conversion Table for Web Services Wrappers .................................................. 84
Table 6  Type Conversion Table for XML Wrappers ............................................................... 85
Table 7  Reserved Characters for Date Format ........................................................................ 141
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 to which reference will be made as the need arises.

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 system, 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 (query and insertion) 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 describes how to use the graphical tools that allow 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 writing VQL sentences at hand. This is not the usual case: the Virtual DataPort Administration Guide [3] describes how to use the graphical tools that allow to perform the most usual administration tasks. Graphic mode is 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 datasource may have to be created prior to the wrappers. The datasources 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:

- \(\text{attribute}\) is an attribute of the relation.

- \(\text{operators}\) is the group of operators that can be used in the queries on this source and with this search method. ‘ANY’ represents any operator allowed by the attribute data type.

- \(\text{obligatoriness}\) can have three values: ‘OBL’ indicates that the attribute should obligatorily appear in any query on the source. ‘OPT’ indicates that the attribute may or may not appear in the query (it is optional) and ‘NOS’ indicates that the queries for this attribute are not allowed in the source.

- \(\text{multiplicity}\) 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 (‘NOS’ value in the obligatoriness field), the value is necessarily 0. ‘+’ indicates a number of values greater than 0 but without an upper limit.

- \(\text{possible\_values}\) is the list of values that can be used to query the attribute. If the value ‘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 ‘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 (‘OPT’ value in \(\text{obligatoriness}\)) and with any multiplicity (‘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](image)

Search form of a virtual bookshop on the Internet
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 \( R = \{ \text{TITLE}, \text{AUTHOR}, \text{FORMAT}, \text{PRICE} \} \) with a search method containing the 5-uples shown in Figure 2.

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

**Figure 2** Search method for a bookshop

Note that the first 5-tuple has the value \{\text{like, =}\} in the OPERATORS field and \text{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:

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

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 datasources and wrappers

A wrapper must be created and assigned to a base relation in order for it 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, Web Services, 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 [3]). It is normally necessary to previously create a datasource 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 in a fully graphic manner with Denodo ITPro (see [6]). It is also possible to create wrappers for specific applications using the CUSTOM wrapper type.

Should you wish to create datasources and wrappers directly using VQL, without the help of the administration tool, section 17 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 required search method or methods from the base relation that represents said source in the system (see section 5). 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 an average review according to C and the minimum value found for the 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 `bookview` like the union of A and B.

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

2. Creating the view R as the join of `bookview` and C by applying an operation `groupby` to select the minimum price for each book.

   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 *a priori* if a specific query can be answered and it also makes it possible for a Virtual DataPort server to be used as a source for another Virtual DataPort server, thus facilitating large-scale projects to be taken on using an incremental approach.

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, EII is ready to accept queries and/or updates on updatable views, according to the SQL 92 standard, and on JDBC/ODBC base relations and CUSTOM wrappers. Details on how external applications can send and execute statements on the Virtual Database are provided in the Developer Guide [5]. 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 checking, if a response can be given to the query in accordance with the capability of the views involved. If the query cannot be responded to, the user is informed of this. If it can, the process continues.

Subsequently, the Plan Generator creates all 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 group, 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 18.2 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.

Another fundamental aspect is that, optionally, as mentioned previously, all or part of the source data in the system caché can be pre-loaded or maintained. 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 8 for further details.
3 CHARACTERISTICS

Virtual DataPort facilitates quick and simple extraction and fusion of data from multiple heterogeneous, structured, semi-structured and unstructured sources.

Virtual DataPort provides a data definition and handling language called Denodo VQL (Virtual Query Language), the syntax of which is similar to SQL (Structured Query Language) with certain extensions required for a virtual integration environment for heterogeneous and distributed data sources. For example, VQL includes different builds to make the querying of non-structured data and its combining with structured data easier.

This section describes some of the important characteristics of Virtual DataPort which the reader must know in order to understand the rest of the document.

- It starts by describing the different available data types.
- Then it goes on to describe the mechanisms for internationalizing the different system data types.
- It introduces the mechanisms for representing the query capabilities of a relation.
- Finally, it introduces the rewriting rule system, which allows difficulties derived from semantic heterogeneities in sources to be resolved.

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 \(1.4\times10^{-45}\) to \(3.4028235\times10^{38}\).
- **double**: Represents a real number in the range \(4.9\times10^{-324}\) to \(1.7976931348623157\times10^{308}\).
- **boolean**: Represents a logical value, true or false.
- **text**: Represents a character string.
- **date**: Encapsulates a date.
Virtual DataPort 4.0 Advanced VQL Guide

• **time**: Represents a time interval or duration.

• **money**: Represents a currency value.

• **blob**: Represents a binary data element.

• **xml**: Represents an XML document (or a fragment of an XML document).

Compound data types that allow new data types to be created 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 arrangement of the elements matters.

As can be seen below (see sections 11.1 and 18.1), Virtual DataPort allows specifically formed compound data types to be defined, which means that hierarchic data elements, such as those normally used in data sources such as Web Services or XML documents, can be naturally modeled.

### 3.2 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 sums 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 English sources, the results can display the currencies, times and dates corresponding to Spain).

There is 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 11.2). Virtual DataPort includes maps already created for the most common configurations of many countries. The name of said configurations uses the standard prefix defined in the standard ISO-3166 [2] (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 18.3 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 18.3 for more information about the parameters of an internationalization configuration and section 13 to know how to obtain the parameters assigned to a certain configuration. Section 4.6.3 describes different functions to process date values that may be useful to express them in the required format.
3.3 QUERY CAPABILITIES

In the context in which Virtual DataPort operates it can occur that the data sources present limited query capabilities. For example, in most Web sources only those queries that comply with the constraints imposed by a certain HTML query form can be executed.

For this, the administrator has to directly specify the query capabilities of the relations that represent the sources of the system (these relations are called base relations). Query capabilities are described in Virtual DataPort using what are called search methods. For each base relation the administrator defines one or more search methods.

The search methods of the derived views are obtained automatically by the system from the base relation search methods that appear in the view and the expression used to construct it, whereby the administrator does not have to worry about its definition in this case.

3.4 REWRITING RULES

At times, different sources can use different formats to represent the same type of semantic content. Virtual DataPort incorporates a system for processing the heterogeneous arrangement of data from various sources. Specifically, a mechanism is provided that allows to rewrite both the queries executed against a source and the results retrieved from it, in order for them to be adapted to the required format.

For example, imagine that a specific source A has an attribute that stores the name of the author of a book using an attribute with the arrangement format ’Surname, Name’ [e.g. ‘Smith, John’ or ‘Shakespeare, William’], while another source B uses a format ’Name Surname’ [e.g. ‘John Smith’ or ‘William Shakespeare’]. Furthermore, when queries are made by the attribute author, both sources expect the name to appear in the query in its specific format. Now imagine that we want to use Virtual DataPort to define a union view of A and B, which we will call C. When the server receives the query `select * from C where author = ‘John Smith’`, it should issue subqueries to A and B to extract the books of the author in question from both sources and return the result.

However, in order for source A to correctly respond to the query, a query rewriting rule will have to be included that converts the character string ‘John Smith’ to ‘Smith, John’.

Just as it may be necessary to rewrite queries to deal with arrangement heterogeneities in the sources, the results received from these may also need to be converted for the arrangement to adapt to the view format of the global schema. The rewriting of results is executed by Virtual DataPort through the result or output rewriting rules.

Continuing with the above example, source A returns its results with the attribute author encoded in the format ’Surname, Name’, while in view C all the results should be in the format ’Name Surname’. Therefore, an output rewriting rule that converts data extracted from A is required.

Rewriting rules are applied to search methods. The types of rewriting rules that can be applied vary depending on whether the search method belongs to a base relation or a derived view. See sections 5.3 and 7.1.1 for more detail.
4 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 following the EII paradigm.

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.

4.1 STATEMENTS

Two types of statements exist:

- 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, maps, types, databases and users.
  - DROP: Eliminates elements such as tables (base relations), views, stored procedures, wrappers, data sources, maps, types, databases and users.
  - ALTER: Modifies specific properties of a table (or base relation) such as its internationalization configuration, cache, swapping, rewrite rules, and so on. It also allows database and user descriptions to be modified, user permission assignments such as access and creation, and datasource, stored procedure and wrapper redefinition.
  - DESC: Shows the description of data types, views, stored procedures, adapters, maps, operators, wrappers, data sources, databases and users defined in the server. It also allows to obtain a hierarchical approach about the dependences of a specific view (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.

4.2 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 are conditions imposed on a statement that assist in defining data that are to be defined, selected or manipulated. The clauses supported by the language are:

- **FROM**: Specifies the relation or relations from which the data is to be selected. It is possible to specify subqueries. It is also possible to specify invocation of a saved procedure.

- **WHERE**: Specifies the conditions the data to be selected should fulfill.

- **UNION**: Facilitates the union of two SELECT statements.

- **GROUP BY**: Used to group the results obtained as the response to a query according to specifically named 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 attribute.

- **CONTEXT**: Used to modify certain configuration options to execute a specific query.

- **TRACE**: Provides the execution plan of a statement.

4.3 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 the 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 8).

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**: Allows for the view to be updated to be indicated.

- **SET**: Allows for the attributes of the view to be altered by the operation to be indicated as well as the new value 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**: Allows for the view to be updated to be indicated.

- **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.

### 4.4 LOGICAL OPERATORS

Logical operators are used to combine 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.

### 4.5 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 XXXX).

The operators currently supported by the administrator are:
• `<': Receives two operands that can be of the types: int, long, float, double, date, time, 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, time, 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, time, money link. 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 regular expressions as operands. It checks whether the character string conforms with all the regular expressions received. Each regular expression must follow standard format in SQL for the expressions used with the SQL like operator. More specifically, the character '%' represents a segment of any length within a character string and the character '_' represents a segment of length 1. For example, the expression '%commerce_' matches any character string ending with the substring 'commerce' and followed by any character. Should the characters '%' or '_' be included as part of a constant substring, they must be escaped by prefixing them with the character '$'. If the escape character is to be included, it must be written '$$'.

Examples: The first query shown returns tuples from the internet_inc view with a summary attribute containing the text 'adsl'. The second requires that they also contain the text 'error':

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

• `contains': This accepts 2 text-type elements as operands. The first operand will be a text-type attribute from an index of external, non-structured data (e.g. Aracne and/or Google Mini data sources). The second operand will be a Boolean search expression written in the search language on non-structured DataPort data.

The syntax of the search language on non-structured data is described in section 19. 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 19.2 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 17.3.10.1).
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 17.3.6 and 17.3.10.1).

**Examples:** The following query returns tuples from the aracneview view, the searchablecontent attribute of which contains the words ‘acme’ and ‘incorporated’:

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

The following query returns tuples from the aracneview view, the searchablecontent attribute of which contains the exact words ‘acme incorporated’ and some other word starting with ‘product’:

```
SELECT * FROM aracneview WHERE searchablecontent contains '"acme incorporated "AND product"'
```

- **containsor**: This 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**: This 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 sole operand, which can belong to the following data types: int, long, float, double, boolean, text, enumerated, date, time, 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, time, money and link. Evaluates if the value is null, i.e. if it does not have any value.

- **is TRUE**: Applied to one sole 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, time, money and link. 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, time 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_d 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 assessment of this operator returns a value of between 0 and 1 that estimates the similarity between the two text-type operands using a variety of similarity algorithms. As well as 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 right-hand operand is a list of text-type elements. The first element in this list is the second character string 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 4.6.2).

**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'

**NOTE:** blob data type values cannot take part in query conditions.

### 4.6 DERIVED ATTRIBUTE AND CONDITION FUNCTIONS

Derived attribute functions are used to generate new attributes in the schema of a view, applying a process to the values of the other attributes of the view, the constants and/or the result of assessing other functions. 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 new functions. In some cases (a subset of the arithmetic functions and those of data handling) the parameters received by a function as well as the value returned by them should all belong to the same data type. For example, the function SUM can add two or more integer values, two or more floating values or two or more values of the type double, but it will not add an integer value to a floating point value. In addition, some functions only operate with elements belonging to a specific data type.

Virtual DataPort provides a series of predefined functions that can be grouped into different types based on the data type to which they are applied:

- Arithmetic functions
- Functions for text processing
- Functions for date processing
- Type conversion functions.
The functions supported by the system are described in the following paragraphs.

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).

### 4.6.1 Arithmetic Functions

Arithmetic functions are applied to numeric-type attributes and literals, int, long, float, double, time and money, with the constraint that all the parameters should have the same type. These allow mathematical calculations to be made on attributes and literals.

The supported arithmetic functions are:

- **SUM**: The sum function receives a variable number of arguments (greater than or equal to two) and returns as a result a new element of the same type containing the sum of those preceding. The infix version of this function receives two arguments and is represented by the symbol ′+′.

- **SUBTRACT**: The substract function receives two arguments and returns a new element of the same type with the result of subtracting the value of the second argument from that of the first. The infix version of this function receives two arguments and is represented by the symbol ′-′.

- **MULT**: The mult function receives a variable number of arguments (greater than or equal to two) and returns a new element of the same type with the result of multiplying the different arguments. The infix version of this function receives two arguments and is represented by the symbol ′*′.

- **DIV**: The div function receives two numeric-type arguments and returns a new element of the same type with the result of dividing the first argument by the second. If the 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**: The min function receives a variable number of arguments (greater than or equal to two) and returns as a result the value of the smallest argument of the list.

- **MAX**: The max function receives a variable number of arguments (greater than or equal to two) and returns as a result the value of the biggest argument of the list.

- **ABS**: The abs function receives one sole numeric-type argument and returns as a result its absolute value.

- **MOD**: The mod function receives two arguments of numeric or money type 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 `time` or `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 `time` or `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 `time` or `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.

### 4.6.2 Text Processing Functions

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

- **TEXTCONSTANT**: This function enables the creation of a text-type element from the literal used as a parameter (it is only needed in the SELECT clause).

- **CONCAT**: The concatenation function receives a variable number of arguments and allows a text-type element to be obtained as a result of concatenating its parameters. The infix version of this function receives 2 arguments and is represented by the symbol ‘||’.

- **LEN**: The LEN function receives as a parameter a text-type argument and returns the number of characters that form it.

- **REPLACE**: This function receives 3 text-type arguments and returns the result of replacing the occurrences of the second in the first by those of the third.

- **REPLACEMAP**: This function 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:
REPLACEMAP (originalText: text, mapName: text). The keys and the replacement values are specified by a key/value map defined by the administrator (see section 11.2 for a description as to the way in which maps are created). 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.

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

• LOWER: This function receives a text-type argument and returns it to the output with all the characters it comprises changed to lower case.

• UPPER: This function receives a text-type argument and returns it to the output with all the characters it comprises changed to upper case.

• SUBSTRING: The substring function receives as parameters a text-type argument and two integer numbers. It returns as output the part of the substring of the first argument that corresponds to the positions indicated by the second (beginning) and third (end) arguments.

• REGEXP: This function allows for transformations on character strings based on regular expressions. It is given 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 [11]. The function behaves in the following manner: The input regular expression is assessed 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:

REGEXP('Shakespeare,William','(\w+),(\w+)','$2 $1')

will be the value of text type ‘William Shakespeare’.
• **TRIM**: This function receives a text-type argument and returns the same argument with all the spaces and beginning and end carriage returns removed.

• **REMOVEACCENTS**: This function receives a text-type argument and returns that same argument value but with no accents.

• **SIMILARITY(value1: text, value2: text, algorithm:text)**: This function receives two character strings and returns a value of 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):

  - Based on the editing distance between the text strings: ScaledLevenshtein, JaroWinkler, Jaro, Level2Jaro, MongeElkan, Level2MongeElkan.

  - 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:

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

4.6.3 **Date Processing functions**

Date functions allow to manipulate date values:

• **NOW**: This function creates a new data value containing the actual date.

• **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.

• TO_DATE: This allows text strings representing dates to be converted into date-type elements. Three text-type arguments are given. The first represents a pattern to express dates (following the standard syntax in JAVA language specified in [12]), whereas the second will be a date expressed according to said 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.

### 4.6.4 Type Conversion Functions

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

• CAST: This function is given 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, time, money</td>
</tr>
<tr>
<td>enumerated</td>
<td>text, enumerated</td>
</tr>
<tr>
<td>float</td>
<td>text, int, long, float, double, time, money</td>
</tr>
<tr>
<td>int</td>
<td>text, int, long, float, double, time, money</td>
</tr>
<tr>
<td>link</td>
<td>text, link</td>
</tr>
<tr>
<td>long</td>
<td>text, int, long, float, double, time, 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, time, xml, money, link, blob, enumerated, register, array</td>
</tr>
<tr>
<td>Time</td>
<td>text, long, int, time</td>
</tr>
<tr>
<td>Xml</td>
<td>text, xml</td>
</tr>
</tbody>
</table>

Table 1 Type conversions permitted with the CAST function

• TO_DATE: This allows text strings representing dates to be converted into date-type elements. See section 4.6.3.
• **CREATETYPEFROMXML**: This function creates a *register* compound type (see section 18.1) based on an XML-type element. It is given 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. See section 4.6.5 for more details.

### 4.6.5 XML Processing Functions

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

- **XPATH**: This function applies an Xpath expression [13] to an XML-type element. It is given 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 be given 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.

- **CREATETYPEFROMXML**: This function creates a *register* compound type (see section 18.1) based on an XML-type element. It is given two arguments: the first is text type and must contain the name of the new type, while the second is a character string which contains an example of the XML element (of text type). The compound-type schema will be inferred by DataPort by analyzing the XML element structure. It returns the name of the new type created. See next sub-section for an example.

#### 4.6.5.1 Converting XML data into Virtual DataPort compound types

By combining the `CAST` and `CREATETYPEFROMXML` functions it is possible to create new *register*-type compound attributes in a view (see section 18.1) based on XML data. Observe the following example: suppose there is a view `V` with an XML-type attribute called `PERSONAL_DATA_XML`. Also suppose that each value of the attribute `PERSONAL_DATA_XML` contains XML code as follows:

```xml
<person>
  <name> John Smith </name>
  <age> 25 </age>
</person>
```

Now consider the following expression:

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

The derived attribute `PERSONALDATA` from the `PERSON` view would be of the `personaldatatype` type, which would be a *register* type made up of the fields `name` (text type) and `age` (long type). Please notice that the second parameter of the `CREATETYPEFROMXML` function must be an example of the values contained in the `PERSONAL_DATA_XML` field from the view `V`.

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.
4.6.6 Other functions

This section describes miscellaneous functions:

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

- **MAP**: This function allows for the value associated with a certain key in a value conversion map to be obtained. There are two possible signatures:

  \[
  \text{MAP} \ (\text{key}: \text{text}, \ \text{viewName}: \text{text}, \ \text{keyField}:\text{text}, \ \text{valueField}:\text{text})
  \]

  This allows for the value associated with a certain key based on values contained in a DataPort view to be obtained. It is given four parameters: the required key, the name of the view containing the conversion map, the name of the view attribute containing the keys and the name of the view attribute containing the values.

  Both signatures return a text-type element with the associated value for the specified key (where no key exists, it is returned as null). The key is upper/lower case-insensitive.

- **COALESCE**: This function receives a variable number of arguments (greater than or equal to two), all of the same data type, and returns as a result the first non-null argument.

- **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:

  \[
  \text{CONTEXTUALSUMMARY}(\text{content}:\text{text}, \ \text{keyword}:\text{text}, \ \text{beginDelim}:\text{text}, \ \text{endDelim}:\text{text}, \ \text{fragmentSeparator}:\text{text}, \ \text{fragmentLength}:\text{int} \ [,\text{maxFragmentNumber}:\text{int}] )
  \]

  , where:

  - **content**: text to analyze and the one from which the most relevant fragments are to be 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 words 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.
4.7 AGGREGATION FUNCTIONS

The aggregation functions are used inside of a SELECT statement to return one single value for every group of tuples which are the result of a grouping operation.

The aggregation functions currently supported by the management system 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.

The different aggregation functions are listed below:

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

- **COUNT**: Used to return the total 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 not null value for a specific attribute. Applicable to any type of attribute. This function can be used without the need to have applied a GROUP BY clause, but in that case it may only be used on the special attribute '*' to count the total of tuples returned.

- **SUM**: Used to return the sum of all the values of a specific attribute that are not null. Applicable to attributes of the type int, long, float, double, time and money.

- **MAX**: Used to return the highest value of a specified attribute. Applicable to attributes of the type int, long, float, double, time and money. The implementation of this function ignores the ALL/DISTINCT modifier.

- **MIN**: Used to return the lowest value of a specified attribute. Applicable to attributes of the type int, long, float, double, time and money. The implementation of this function ignores the ALL/DISTINCT modifier.

- **FIRST**: Used to return the value of an attribute in the first tuple of each group of values. Applicable to any type of attribute. The implementation of this function ignores the ALL/DISTINCT modifier.

- **LAST**: Used to return the value of an attribute in the last tuple of each group of values. Applicable to any type of attribute. The implementation of this function ignores the ALL/DISTINCT modifier.

- **LIST**: Aggregation function that returns a list with all the values of a specified attribute for each group. Applicable to any type of attribute.

4.8 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>`'>` `<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. They are also used to access elements of an array or to define a multivalued parameter of a function. In these cases it is 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. Has a special significance when used in the SELECT statement or with the aggregation function COUNT.

• 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 `-` sign 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>).** These allow any string that is not an identifier nor a number nor a logical value to be defined. This may be any string that is found in inverted commas (single or double commas). Allows text strings to be created containing any character, with the sole exception that single or double comma characters (depending on the case) should be escaped (escape character `\` and `\` respectively).

**Operators (<operator>).** Allow system operators to be defined that can be represented by identifiers, by a combination of the symbols defined for <opsymbol> or the expressions “is null”, “is not null”, “is true” or “is false”.

```plaintext
<identifier> ::= [A-Za-z\200-\377_][A-Za-z\200-\377_0-9$]*
<integer> ::= [-][0-9]+
<number> ::= <integer> | (([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))
<boolean> ::= true | false
<literal> ::= '[^']*' | "[^"]*"
<operator> ::= <unary operator> | <binary operator>
<opsymbol> ::= [~!@#$%^&\|`?<>\=]+
<brainary operator> ::= =
| <identifier>
| <opsymbol>
<reserved VQL word> ::= ADD, ALL, ALTER, AND, ANY, ARN, AS, ASC, BASE, CALL, CLEAR, CONNECT, CONTEXT, CREATE, CROSS, CUSTOM, DATABASE, DEFAULT, DESC, DF, DISTINCT, DROP, EXCEL, EXISTS, FALSE, FIELD, FILTER, 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, RAW, RAWPATTERNS, READ, REVERSEORDER, REVOKE, RIGHT, ROW, SELECT, SWAP, TABLE, TO, TRACE, TRUE, UNION, USER, USING, VDB, VIEW, WHERE, WITH, WRITE, WS, ZERO
```

**Figure 3** Basic primitives for specifying VQL statements

### 4.8.1 Syntax of functions and condition values

As mentioned throughout this manual, different types of functions exist in Virtual DataPort that can be used for various purposes.

A first type of functions are those that calculate a value from a tuple. The functions of this type are listed in query conditions or are used to obtain a derived attribute.

Another type of functions is the aggregation functions that calculate a value from a series of tuples. Other transformation functions also exist that act in rewriting rules.

Virtual DataPort uses a common generic function syntax to represent all the functions existing in the system. Said syntax is shown in Figure 4.
To define the syntax of a function the following elements must be predefined:

- The element `<field name>` defines the syntax for specifying an attribute of a relation, in which the name of the relation and the reference to fields of compound-type elements can appear (see section 18.1 for a detailed description of support for compound types).

- The `<value>` element defines the syntax for any parameter of a function that can be a 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 6.3.1). As can be observed, the parameter of a function can in turn be a new function. In addition, a `<value>` allows infix notations to be specified for a function (see the `<value> <funcsymbol> <value>` rule).

Once the basic elements of a function have been defined, 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 18.2 for more information on the different
internationalization configuration parameters and section 13 to find out how to consult the parameters assigned to a specific internationalization configuration.
5 CREATING A BASE RELATION (OR BASE VIEW)

The statement `CREATE TABLE` allows a base relation to be created in Virtual DataPort. A base relation represents an external source (Web, relational, etc.) that supplies data for the mediator system.

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. The type of a certain attribute determines what operators can be applied to it. The schema of the view corresponds with the attributes that comprise each tuple of the base relation and the data types to which each attribute belongs.

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 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 derived views wherever possible.

Figure 6 shows an example of the creation of a base view through the statement `CREATE TABLE`. A base view of the 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

5.1 MODIFYING A BASE RELATION

By using the sentence `ALTER TABLE` it is possible to configure the following specific properties of a base relation: its internationalization configuration, its cache configuration, its swapping configuration, add, delete or modify a search method and add, delete or modify a rewriting rule. Figure 7 shows the syntax of `ALTER TABLE`.

Once the structure of a base relation has been created and defined, the search methods must be specified in order for the server to know what queries can be made on the source data it represents. Search methods are composed of rules that represent the restrictions with which a specific query should comply in order for it to be run using this
search method. Furthermore, each search method has an associated wrapper which contains the data necessary to translate the user query for the source and interpret its response. Section 5.2 provides more details on this matter.

It is also possible to apply query rewriting rules to base relation search methods (which rewrite queries before sending them to the source to adapt them to the formats required by it) or result rewriting rules (which rewrite the results obtained from the source to adapt them to another format as required) in order to deal with heterogeneous representations of data from various sources. Section 5.3 deals with the rewriting rules in detail.

The \texttt{ALTER\ TABLE} command also indicates whether the base relation should be cached (\texttt{CACHE\ ON}), i.e. if the tuples being extracted from the source as a result of executing the queries should appear in the local cache. Section 18.2.2 provides more details on this matter.

Finally, it is possible to configure the swapping to disk policy used by DataPort on running queries on the base relation involving a large number of tuples. See section 18.2.3 for a detailed description.

\begin{verbatim}
ALTER TABLE <name:identifier>
 [ I18N <name:identifier> ]
 [ CACHE { ON | POST | OFF | INVALIDATE } ]
 [ TIMETOLIVEINCACHE <seconds:integer> ]
 [ SWAP { ON | OFF } ]
 [ SWAPSIZE <megabytes:integer> ]
 [ <table search method clause>* ]

<table search method clause> ::= 
ADD SEARCHMETHOD <name:identifier> ( 
 [ I18N <name:identifier> ]
 [ CONSTRAINTS { [ <constraint clause> ]+ } ]
 [ OUTPUTLIST ( <output clause> ) ]
 [ <wrapper clause> ]
 [ PATTERNS { [ <pattern clause> ]+ } ]
 [ RAWPATTERNS ( [ <pattern clause> ]+ ) ]
 )
 | ALTER SEARCHMETHOD <name:identifier> ( 
 [ I18N { <name:identifier> | DEFAULT } ]
 [ CONSTRAINTS ( [ <constraint clause> ]+ ) ]
 [ OUTPUTLIST ( <output clause> ) ]
 [ <wrapper clause> ]
 [ PATTERNS ( [ <pattern clause> ]+ ) ]
 [ RAWPATTERNS ( [ <pattern clause> ]+ ) ]
 )
 | DROP SEARCHMETHOD <name:identifier>

<constraint clause> ::= 
ADD <field> ( [ <operator> [, <operator> ]* ] )
 { 
 <obligatoriness> <multiplicity> 
 [ ( { <value_1:value> [ , <value_i:value> ]* } ) ]
 | NOS { ZERO | 0 } ()
 } 
 | DROP <integer>

<output clause> ::= <field> [,<field> ]*
<wrapper clause> =
\end{verbatim}
5.2 QUERY CAPABILITIES: SEARCH METHODS AND WRAPPERS

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.

5.2.1 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 taken: ‘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 with 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 taken. 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 the search range is not stated (within the range associated
with the data type of the attribute) and the attribute can be queried with any value. If the obrigatoriness
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.

5.2.2 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 17.

5.2.3 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.

```
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 ITPilot-type wrapper called booktest is associated with the
search method as that in charge of extracting the results, when a query is carried out using this search method.

It is important to highlight that although the source cannot make queries itself for specific attributes (in the previous
example this occurs with FORMAT, PRICE, etc.), Virtual DataPort is capable of executing some of the queries on
said attributes through post-processing of the results obtained from the sources. For example, if the administrator
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.

5.3 REWRITING RULES

As already mentioned, the rewriting rules system is used to deal with both semantic and source format
representation heterogeneities.
There are two types of rewriting rules:

- **Input Rewriting Rules.** These adapt the queries sent to a relation to the representation formats it is familiar with.

- **Output Rewriting Rules.** These adapt the results obtained from a relation to the format expected by the higher views.

Rewriting rules are applied to search methods. The type of rewriting rules that can be applied vary depending on whether the search method belongs to a base relation or a derived view.

More specifically, a search method for a base relation has four different associated rewriting rule lists: two lists of the type raw (one for results and another for queries) and another two of the type not raw (an input - queries - and another output - results -). However, the search method of a derived view has only two associated lists of the type not raw.

A raw-type rule differs in that it can process the attribute values of a base relation without being subjected to the restrictions associated with the data types of said attributes. This is possible because these rules operate either on queries to be sent to a wrapper (input rules) or on data returned by a wrapper (output rules), and the data sent to or returned by a wrapper are not typed by the server. The use of raw-type rules is particularly important when dealing with enumerated-type attributes. See sections 5.3.2.3 and 5.3.1.3 to see some examples of raw-type rules.

On the other hand, a derived view has two associated rewriting rule lists for each subview immediately below it in the view hierarchy: one rewriting rule list for queries and another list of rules that rewrites the results, whereby raw-type rules cannot be used.

The following sections describe the use of input and output rewriting rules, respectively.

### 5.3.1 Input Rewriting Rules

Input rewriting rules of a base relation, i.e. those that act on query conditions (independent of whether they are raw or not), are applied to a search method and are comprised of the following elements:

- The expressions-pattern or patterns associated with the rewriting rule indicate the structure to be followed by the query conditions that are rewritten using this rewriting rule. An expression-pattern is composed of various conditions-pattern. Each condition-pattern consists of two elements: an attribute of the relation and a group of operators. A query condition matches or verifies a condition pattern, if its left part is the condition-pattern attribute and one of the operators contained in the condition-pattern is its operator. The expressions-pattern should be constructed so that no condition matches various conditions-pattern. A pattern-expression is verified for a query condition if: (1) for each pattern-expression there exists a simple condition satisfying it; and (2) if the simple conditions that accomplish each one of the conditions-patterns are not contained in any OR nor NOT condition.

- The transformation function. This is the function that rewrites the conditions that have matched the expressions-pattern, producing as output a new group of query conditions that replace the original. A transformation function is a predefined function parameterized by individual values or lists of individual values.

- The position the rewriting rule takes in the rewriting rules list associated with the search method on which it is acting.
The result of condition rewriting is the subgroup of conditions that have not matched any expression-pattern plus the result of applying the transformation function to the subgroup of conditions that have matched.

As we can see in Section 4.8, a transformation function of an input rewriting rule is comprised of a name and a series of parameters. The parameters are optional, separated by commas and can be simple elements (literal, attribute name or number, with the general syntax defined in section 4.8) or lists. Lists are delimited by the characters '[' and ']', their elements are separated with the character ',', and can be any simple element.

The already implemented input transformation functions incorporated into Virtual DataPort are: REMOVEACCENTS(), REGEXP(), MAP() and CHANGEOPERATOR(). The following subsections deal with each of these, respectively.

5.3.1.1 REMOVEACCENTS
The transformation function REMOVEACCENTS() removes the accents from search literals. Parameters are not given to this function.

5.3.1.2 REGEXP
The function REGEXP() transforms the values that result from the evaluation of the right parts of the conditions from an input format to an output format, parameterizing these formats with variables. This function is given two parameters: an input regular expression and an output one.

Regular expressions must be written by using the JAVA syntax for regular expressions [11]. Figure 9 shows an example of an input rewriting rule that uses the transformation function REGEXP(). The transformation function receives overwrites the conditions in the form (AUTHOR like '<value>'), where '<value>' is a string that represents the name and the surname of the author separated by a comma, interchanging the order of the name and surname and eliminating the comma. For example, said rewriting rule replaces the condition (AUTHOR like 'Smith, John') with (AUTHOR like 'John Smith').

```
ALTER TABLE bookview
ALTER SEARCHMETHOD bookview_sm1 (PATTERNS (ADD INPUT FIELD AUTHOR like REGEXP('$\w+'),(\w+),',','$2 $1') 1));
```

Figure 9  Adding an input rewriting rule REGEXP()

5.3.1.3 MAP
The MAP() input transformation function modifies the operands of a list of query conditions based on the application of maps directly corresponding with specific conditions. That is, it transforms some conditions into others based on a map (see map creation in Section 11.2).

This function is attributed two parameters: the name of the internationalization configuration for the map conditions and the name of the map that contains the condition transformation correspondences.

Figure 10 shows a code fragment that adds a raw-type input rewriting rule to a specific search method, which uses the function MAP() in the second position in the search method rewriting rules list.

```
ALTER TABLE otherview
```

Creating a Base Relation (or Base View) 33
ALTER SEARCHMETHOD otherview_sm1 {
    RAWPATTERNS {
        ADD INPUT
        FIELD TIMETABLE =
            Map('es_euro', 'daysOfWeek') 2
    }
};

Figure 10 Adding a rewriting rule on MAP() conditions

The function MAP() uses the map "daysOfWeek" (created in Figure 36), which contains conditions expressed in accordance with the internationalization configuration called "es_euro". Note that, in this case, a rewriting of the type raw must be used, as the field TIMETABLE has an enumerated data type that only allows certain values (more specifically, those shown in the column on the left of Figure 36), and the result of the rewriting is a value that is not compatible with this data type (more specifically, the result is one of the values shown in the column on the right of Figure 36 that are not values permitted by the data type of the attribute TIMETABLE).

5.3.1.4 CHANGEOPERATOR

The input transformation function CHANGEOPERATOR() changes the operator of the list of conditions that match the rewriting rule. Receives as a parameter the conditions operator name after the transformation function has been applied to them. For example, in Figure 11 a rewriting rule is defined that, when applied to a condition of the form (TITLE = 'value'), is replaced by the condition (TITLE like 'value').

NOTE: This function is currently deprecated. It should not be used in new applications.

ALTER TABLE bookview
ALTER SEARCHMETHOD bookview_sm1 {
    PATTERNS {
        ADD INPUT
        FIELD TITLE =
            ChangeOperator('like') 1
    }
};

Figure 11 Adding an input rewriting CHANGEOPERATOR()

5.3.2 Output Rewriting Rules

The output rewriting rules of a base relation, i.e. those that act on result tuples, are applied to a search method and are comprised of the following elements:

- The condition associated with the rewriting rule. Indicates which tuples this rewriting rule should be applied to. If no condition is specified, this means that the rewriting rule rewrites all the tuples of the view. These conditions are constructed in the same manner as those used with the WHERE clause in the SELECT statement (see section 6.3).

- Internationalization of the condition. Indicates the internationalization configuration of the literals contained in the condition.

- Output transformation function. The function that executes rewriting of a result tuple. Said transformation function receives the tuples that verify an expression-pattern and rewrites them obtaining a new series of tuples that replaces those initially matched.
Position of the rule in the list of rewriting rules associated with the search method with which it is associated.

An output transformation function has the same syntax as an input transformation function (it follows the generic syntax of the Virtual DataPort predefined functions described in section 4.8).

The transformation functions for results included are: REMOVEACCENTS(), REGEXP() and MAP(). The following subsections deal with each of these, respectively.

5.3.2.1 REMOVEACCENTS

The output transformation function REMOVEACCENTS() removes the accents from the values of a group of attributes. The parameter given is the list of text-type attributes on which it should act, removing the accents from the strings they have as values in the tuple. The list of attributes is specified in brackets, separated by commas (e.g. [field1, field2]).

5.3.2.2 REGEXP

The output transformation function REGEXP() rewrites the values for a group of tuple attributes that match an input regular expression, to an output format, parameterizing both formats with variables. The input and output regular expressions follow the same syntax as those used by the input transformation function of the same name (described in section 5.3.1.2). This function is attributed three parameters: the list of attributes of the tuple being rewritten (specified in brackets and comma-separated), the input patterns and the output patterns. The last two indicate the format of the input and output values for said attributes, respectively.

5.3.2.3 MAP

The MAP() function rewrites a tuple applying direct correspondence maps to specific fields of same. The internationalization option to be applied is given as a parameter, in addition to the list of the names of the attributes to which the maps are applied (specified in brackets and comma-separated) and the list of maps to be applied to each of the attributes specified in the previous attribute.

For example, imagine that we have an electronic bookshop which can return one of the following values for format data: 'eBook', 'Hardcover' or 'Paperback', while the base relation that represents said source has an attribute FORMAT that belongs to an enumerated data type that allows the following values: 'EB','HC' or 'PB'. We could define a map to convert values and, subsequently, use a rewriting function MAP to rewrite the results obtained from the source correctly.

Figure 12 shows how to add a raw-type rewriting rule to results that rewrites the value of the field FORMAT by applying the map "testBookFormat" (expressed in accordance with the internationalization configuration "es_euro") to all the tuples resulting from the execution of the search method bookview_sm1.

```
ALTER TABLE bookview
ALTER SEARCHMETHOD bookview_sm1 (
  RAWPATTERNS (ADD OUTPUT () es Map('es_euro', [FORMAT], ['testBookFormat']) 1)
);
```

Figure 12 Adding an output rewriting rule with the function MAP()

Note that, in this example, the rewriting rule is of the raw-type, because the values used by the source for the data of the FORMAT attribute are not valid for the enumerated data type defined for this attribute (in fact, it will be the
rewriting rule that converts the value received from the source to one of the values accepted by the attribute data type).
6 QUERIES: SELECT STATEMENT

Virtual DataPort allows queries to be made to previously created relations using the SELECT statement. The syntax is shown in Figure 13. The syntax of this and of all VQL statements can also be queried by using the HELP command (see section 16.2).

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

```sql
{ <select> | <union select> }
| FILTER <function> [; <function> ]*
| ORDER BY <order by field> [; <order by field>]* [ ASC | DESC ]
| 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 ( <join condition> )
| <inner view1:view> NATURAL [ <join type> ] JOIN <inner view2:view>
| <inner view1:view> NATURAL [ <join type> ] JOIN [ <nestedParallelNumber:integer> ] <inner view2:view>
| <inner view1:view> NATURAL [ <join type> ] JOIN [ <nestedParallelNumber:integer> ] <inner view2:view> USING ( <field> [, <field>]* )
| <inner view1:view> USING ( <field> [, <field>]* )

Queries: Select Statement
Figure 13 Syntax of the SELECT statement
6.1 FROM CLAUSE

Specification of the origin table 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. Aliases can be indicated for the relations in the FROM clause that can be used in the other clauses in the SELECT statement and these will facilitate the creation of Join conditions. If an alias is indicated for a relation in the FROM clause, the name of the relation may 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 Cartesian 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 10 for more details.

6.1.1 Join operations

The FROM clause also allows the Join conditions among the views involved in a query to be defined. The following construction must be used to do so:

```
FROM view1 JOIN view2 ON (joinCondition)
```

where view1 and view2 are involved views and joinCondition specifies the required join condition.

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 the tuples built from the tuples of both relations associated according to the join conditions in the result. 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 will have no associated tuple in the other relation (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 Cartesian 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. This case is equivalent to using a join condition that requires values of the attributes specified in both relations to be equal. **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 18.2.1 for more details on this matter.

### 6.1.2 **FLATTEN VIEW (Flattening data structures)**

Denodo Virtual DataPort supports the modeling of data types with tree structure through the use of the types register and array (see section 18.1).

In Virtual DataPort, an array-type element must be thought of as a subrelation. In truth, 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 is assigned no 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 sole 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. Said 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 18.1). The constructor FLATTEN allows tuples to be generated from the compound subfields of the type array of the original tuples of a specific view. Its syntax (see Figure 14) allows the following alternatives:

- Specifying the name of an attribute of the type array, a view is generated that has as its schema that of the register contained in the indicated array. The name of the specified array subelement can be found contained in one or various register structures (but no other array may be traversed).
- Specifying the name of a view and an alias it is possible to obtain the flattened representation of an array that could in turn be contained in 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 contained in another an alias
must be added to the last array indicated and the array element of interest must be specified on this (to traverse more 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.

```
<flattened 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 14** 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)
```

### 6.2 SELECT CLAUSE

Through the `SELECT` clause the attributes to be obtained from the relations specified in the `FROM` clause are indicated (separated by commas). The queries that a user can make may involve either base relations or previously defined derived complex views.

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.

As the result of a query to Virtual DataPort, a relation or view is obtained whose schema is given by the names and types of the attributes specified in the `SELECT` clause. An alias may also be defined for the columns obtained, thus allowing the name of any attribute to be modified.

In the case of derived attributes, it is not necessary to define an alias to indicate the name of the new field of the relation (see section 6.2.1). If not specified, then the system will associate as an alias the name of the function that generates the derived attribute, in the case of functions, and the name of the last subfield listed in the case of a subelement of a compound field.

In a query, if a field has no associated alias, and it is a reference to a field of a lower-level view, its name is the name of the view where it comes from, followed by the field name (separated by the "." character). However, in a view, the names of the fields do not keep the name of a lower-level view; therefore, the name of a field with no associated alias, which is itself a reference to a lower-level view, in a view, will be simply the name of the field of the lower-level view with no view name at all.

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 alias).

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

#### 6.2.1 Derived Attributes

In the `SELECT` clause, in addition to columns that come directly from attributes of relations included in the `FROM` clause, columns can be included the values of which are derived from a calculation based on values from other columns. This type of attribute is usually called a *derived attribute*. 
The values of a derived attribute are obtained through a predefined function (see section 4.6) that can receive as parameters the values of other columns, as well as constant values or even the result of applying other functions of the same type. The parameters received by a function, as well as the value that will be returned for it, should belong to the same data type. Furthermore, some functions only operate with elements belonging to a specific data type.

A description of the functions supported by Virtual DataPort can be found in section 4.6.

Some examples of how to use derived attribute functions are shown below. The following query obtains the salary of the employees with an increase of 1,000 euros in a column called `newSalary`.

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

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

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

### WHERE CLAUSE

The WHERE clause specifies the conditions the results of the query made should comply with. The syntax for specifying a list of conditions is shown in Figure 15.

```
<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** Syntax for a list of conditions

A condition is a sequence of condition elements separated by the logical operators AND, OR or NOT. Each condition represents a comparison operation which, when evaluated, obtains a positive (true) or negative (false) value as a result. The conditions can be grouped between the symbols ‘(’ and ‘)’ to vary their priority.

A condition is composed of an argument, which is the element to which the condition is applied and which corresponds to the first operand, followed by a comparison operator. This operator, in turn and wherever it is not unary (IS NULL, IS NOT NULL, IS TRUE or IS FALSE), is followed by one or various arguments depending on the nature of same. The comparison operators supported by Virtual DataPort are specified in section 4.5, but they include operators of equality, greater/lesser comparison, string contention, etc.

It can be said that a condition is composed of three elements: a left part (which is the operand to which the comparison operation is applied), an operator and a right part (which is optional and contains the rest of the operands that are to be compared with the left part).

A condition operand can be the name of an attribute, a literal, a number, a logical value, an expression to be evaluated or a compound value (see section 6.3.1). An expression as a condition operand is a predefined and parameterized function like that used to specify a derived attribute in the SELECT clause, but with one added peculiarity: in most of the cases, some operand to which the condition is applied (i.e. that specified before the operator or some operand of the right) must be parameterized by at least the name of an attribute (only in some
cases: for example, it does not apply to the TEXTCONSTANT(“text”) function because it always return a text-type element).

As mentioned earlier, some operators can allow operands of various data types, but in a same condition all the operands should belong to the same type. For example, the operator ‘=’ compares elements of any data type (text, integer, float, etc). But within a specific condition all the operands must be of the same type (for example, an integer-type value cannot be compared with a text-type value, although it is possible to use the CAST function for precise type conversions before the comparison).

6.3.1 Conditions with compound values

The ROW constructor allows register-type compound values to be created (see section 18.1). 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 ‘{ ’y’}’. 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.

6.4 GROUP BY CLAUSE

Another clause belonging to the SELECT command is GROUP BY. This clause enables the results obtained from a query to be grouped 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 or with empty 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 group by attributes can be specified in it (i.e. those specified in the GROUP BY). The remainder of the attributes should appear as parameters of aggregation functions that calculate the "aggregated" value of each attribute in the result tuples. When an aggregation function is specified in the SELECT clause, an alias must be indicated for the new attribute in the result view. 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.
6.4.1 Use of Aggregation Functions

An aggregation or aggregate function is applied to the tuples belonging to a group resulting from a GROUP BY operation and calculates a value from same. The aggregation functions that exist in Virtual DataPort are enumerated in section 4.6.4.

The aggregation functions follow the general syntax of the predefined functions (see section 4.8), 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 counts 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 these, the following query would be executed:

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

6.5 HAVING CLAUSE

The HAVING clause specifies filter 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:

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

6.6 UNION CLAUSE

The SELECT statement also allows a new view to be obtained that contains the tuples contained in another two views or queries. This is possible through the UNION clause. This corresponds with the relational algebra union operation with the exception of some slight differences.

Each relation involved in the union operation corresponds to a specific query. To specify a union of various queries these are separated by the word UNION.

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 (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.

6.6.1 Specifying projections in UNION queries

As an additional characteristic, the fields to be projected from a union view can be indicated in the SELECT statements of Virtual DataPort; that is, a <union view> result projection (SELECT <field list>) can be indicated. Its syntax is shown in Figure 16.
<union select> ::= <select> [ UNION <select> ]+
<projected union select> ::= SELECT <select fields> FROM ( <union select> )

Figure 16 Syntax for a projection of the result of an union

6.7 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. 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).

6.8 CONTEXT CLAUSE

A query has an execution context, i.e. a series of specific execution characteristics that have default values. For example, the results of a query, unless otherwise stated, are displayed in accordance with a specific internationalization configuration, i.e. in a specific currency, language, etc. The SELECT statement allows the execution context to be altered for the current query through the CONTEXT clause.

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. 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 17 for a formal description of the syntax). The execution characteristics that can be configured through CONTEXT are:

- **i18n**: Internationalization for the results of a query. Said property takes the name of an internationalization configuration as a value (for example, es_euro). An internationalization is valid, if a map of the type i18n exists in the catalog (in section 11.2 the map types belonging to the catalog are explained in detail).

- **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".

- **Recalculate**: Forces the search methods of the views involved in the query to be recalculated. This property is useful, when some source search methods have been altered and the system must disseminate said changes through the view tree. The values this property can take are "ON" (that obliges the capabilities to
be calculated) and “OFF” (to not recalculate the capabilities of the views). By default, Virtual DataPort does not recalculate the capabilities.

**NOTE:** The modification of search methods on views on which other derived views already depend is an action that is likely to fail due to the fact that the propagation of these changes may affect operations undertaken by the derived views (e.g. access to fields that no longer exist). A very careful study of the critical implications this entails is therefore recommended.

- **QueryPlan:** This allows different characteristics of the query running plan to be specified in run time. See section 18.2.1 for more details.

- **DelegateUnnamedViews:** It can take the values ‘YES’ or ‘NO’. If it is not specified, ‘YES’ is assumed. Why this option is useful is explained in the following lines. When the administrator creates views using the DataPort graphical administration tool, the system may create some additional intermediate views. DataPort assigns an internal name for these views instead of allowing the user to specify one. That is why we term these views as “unnamed”. Besides, when a query is executed against an existing view, DataPort can also create a temporal unnamed projection or aggregation view which is destroyed when the query ends. In addition, DataPort will allow tries to push as much processing as possible to the data sources. In some cases, pushing the processing of the unnamed views to the source may cause the cache of a view to be ignored, even if the cache has been activated. In those cases, the user can decide to set the value of this property to ‘NO’. See section 18.2.1.1 for detail.

- **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 18.2.3 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 18.2.3 for more details.

```plaintext
<context information> ::= 
    <literal> = <context value> 
    | QUERYPLAN = <query plan>

<query plan> ::= (see section 18.2.1.1)
<context value> ::= <number> | <boolean> | <literal>
```

**Figure 17** Syntax of the CONTEXT clause

### 6.9 TRACE CLAUSE

Using the TRACE clause of the SELECT command, the server is requested a detailed trace level in the execution 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 source via a wrapper.
The `TRACE` clause displays its most relevant parameters for each node on the query execution tree. The DataPort administration tool (see *Administrator Guide* [3]) allows for the execution trace to be displayed in graphic form, providing a simpler analysis of this information.

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 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 datasource and the connection data used to access each one are indicated.
  - For each view-type node, whether the cache has been used, whether swapping has been necessary and whether there are rewrite rules associated to it, etc. are indicated.

- **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:

```sql
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.

```sql
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,
```

Queries: Select Statement
Figure 18: Execution trace

To analyze the query execution trace, the use of the DataPort administration tool is recommended (see Administrator Guide [3]), which displays the execution trace in graphic form.
7 DEFINING A DERIVED VIEW

The administrator can use the base relations of the system to define new ones, extending the functionalities of the server. These new relations are called derived views or views of the global schema.

Virtual DataPort allows mediation schemas to be defined for each specific aggregation application by defining views of the base relations. The system is responsible for automatically calculating the search methods of the new relations from the search methods of the base relations involved in the view and the expression used to create these. The statement CREATE VIEW allows derived views to be created in the administrator mediator schema. The syntax is shown in Figure 19.

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

<select> ::= (see Figure 13)
<context information> ::= (see Figure 13)
```

**Figure 19** Syntax of the statement CREATE VIEW

As can be seen, a name and the query that defines it are specified, when defining a view. The query specified is defined using the syntax of the SELECT statement, which has been explained in detail in section 6.

Therefore, the administrator can create new derived views using different types of relations between elements already existing in the system. For example, a new relation can be generated that is the union of various, the join, the Cartesian product, the selection, the projection or the group-by.

Furthermore, when creating a new relation, several of these types of operators can be combined with the level of complexity required. Views can also be defined that take other previously defined views as a base, whereby view trees with as many levels as required can be created.

For example, considering the views A, B and R as base relations (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 20.

```
∪
∪
```

**Figure 20** Example of how a view is defined in accordance with others

Some properties of the view are indicated through the context of the query that defines it (CONTEXT clause of the SELECT statement; see section 6.8). For example, the internationalization configuration for the values of the tuples
contained in a view is indicated in the context through the property ǐ18n (this manner, if the base relations that form part of a view use different internationalization formats, Virtual DataPort automatically homogenizes them).

The creation of a view also accepts the SQL standard clause WITH CHECK OPTION, the use of 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 8.4.

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 capacities of the upper level derived views wherever possible.

7.1 MODIFYING A DERIVED VIEW

Once a derived view has been created using other existing views, it is possible to alter some of the properties:

- Its internationalization options through option ǐ18n (see section 3.2),

- Cache configuration through the CACHE and TIMETOLIVEINCACHE options (see section 18.2.2),

- DataPort swapping policy configuration through the SWAP and SWAPSIZE options (see section 18.2.3),

- Execution strategy configuration for the joins involved in defining the view through the QUERYPLAN option (see section 18.2.1).

- Condition or result rewriting rule configuration for processing the heterogeneous representations of data from several sources (see section 7.1.1).

The statement ALTER VIEW allows the Virtual DataPort administrator to execute all these operations. The syntax is shown in Figure 21.

```sql
ALTER VIEW <name:identifier> 
[ CACHE { ON | POST | OFF | INVALIDATE } ]
[ TIMETOLIVEINCACHE <seconds:integer> ]
[ SWAP { ON | OFF } ]
[ SWAPSIZE <megabytes:integer> ]
[ QUERYPLAN = <query plan> ]
[ <view search method clause> ]*

(view search method clause) ::= 
ALTERN <name:identifier> {
   [ PATTERNS ( [ <pattern clause> ]+ ) ]
}

<pattern clause> =
ADD INPUT [FIELD <field> <operator> [, <operator> ]+ ]* ]
<function:integer>
| ADD OUTPUT { () | <condition_clause> }
```
### 7.1.1 Rewriting Rules on Derived Views

The statement `ALTER VIEW` allows rewriting rules to be added or removed. Input rewriting rules indicate how the queries made to the higher view should be “translated” so that they can be delegated to the views immediately below. And output rewriting rules indicate how the tuples obtained from the lower views should be converted to include them as a result of the view immediately above in a homogeneous manner.

In section 5.3 the structure of the input and output rewriting rules for a base relation search method is described. The rewriting rules associated with a derived view are the same as those applied to the base relations, with the difference mentioned earlier that rewriting rules for base views are indicated at the search methods from a source, while rewriting rules for derived views are specified at the level of the views found below in the view hierarchy. A derived view has two associated lists of rewriting rules of the `non-raw` type for each view immediately below it: a list of result rewriting rules and another for queries. Note that rules of the type `raw` (that can be used on base relations, as shown in section 5.3) cannot be used at this level, as consistency must be ensured in relation to the data types of the attributes of a derived view and of the relations that comprise it.

As already mentioned in section 5.3.1, a rewriting rule on conditions or input comprises three different parts:

- the expressions-pattern or patterns that indicate the structure to be followed by the query conditions, which should be rewritten using this rewriting rule, and

- the transformation function which rewrites the conditions that have matched the expressions-pattern and their position in the list of rewriting rules associated with the specified container.

- the position that it occupies in the list of input rewriting rules associated with the specified container.

Likewise, a rewriting rule on results or output is structured in the following elements:

- the condition which should verify the tuple in order that the rewriting rule be applied to it,

- the internationalization of said condition,

- the output transformation function that rewrites the results and

- the position that it occupies in the list of output rewriting rules.

The example of Figure 22 an input rewriting rule is added to the view `bookview` in the first position. The rewriting rule changes the format of the values of the query conditions that use the `like` operator for the attribute `AUTHOR`, delegating them to the base subview `bookshop`.

```
ALTER VIEW bookview
```
ALTER bookshop (  
PATTERNS (  
    ADD INPUT  
    FIELD AUTHOR like  
    REGEXP('(\w+),\w+)', '"$2 $1') 1  
  )  
);  

Figure 22 Adding an input rewriting rule for a derived view

The statement in Figure 23 should be executed to remove the first input rewriting rule applied to the view bookshop from bookview.

ALTER VIEW bookview  
ALTER bookshop (  
PATTERNS (  
    DROP INPUT 1  
  )  
);  

Figure 23 Deleting an output rewriting rule from a non-base view

And finally, Figure 24 shows how to add a result rewriting rule to the view bookview in the first position. The output rewriting rule modifies the format from the values of the attribute AUTHOR in all the tuples (as there is no condition) that come from the subview bookshop.

ALTER VIEW bookview  
ALTER bookshop (  
PATTERNS (  
    ADD OUTPUT () es REGEXP([AUTHOR], '(\w+),\w+', '"$2 $1') 1  
  )  
);  

Figure 24 Adding a results rewriting in a derived view
8 INSERTIONS, UPDATES AND DELETION OF VIEWS

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 database-type sources (JDBC/ODBC sources. See sections 17.3.1 and 17.3.2) or CUSTOM-type sources (see section 17.4.9). 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 DataSources. See sections 17.3.1 and 17.3.2) or use a MY-type wrapper providing support for updates (see section 17.4.9).

- The derived attributes cannot be updated.

- A view using an aggregation function (despite there being no GROUP BY clause) cannot be updated.

8.1 INSERT STATEMENT

The INSERT statement allows for a new tuple to be inserted in a view, updating the data source directly. Figure 25 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>[.<identifier>]
```  

For example, the following statement adds a new tuple to the internet_inc view:

```
INSERT INTO internet_inc
VALUES ("a", "b", "c")
```
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 represented by 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"

8.2 UPDATE STATEMENT

The UPDATE statement allows for the value of certain attributes to be altered for all tuples of a view that verify a certain condition, directly updating the data source. Figure 26 shows its syntax:

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

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

For example, the following statement alters the tuples of the internet_inc view, the value of which for the iinc_id attribute is 6, making its value for specific_field1 and specific_field2 attributes 10:

```
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, represented in the table by 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
```

### 8.3 DELETE STATEMENT

The DELETE statement allows for the tuples of a view to be deleted, verifying a certain condition and updating the data source directly. Figure 27 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 27 Syntax of the DELETE statement

For example, the following statement deletes the tuples of the `internet_inc` view, the value of which 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, represented in the table by the `internet_inc` view.

### 8.4 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 `taxid` attribute is inconsistent with the selection condition used to define `incidences_acme`.

```
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")

The CASCADED modifier is used so that this check is also applied to the conditions of lower level views (see Figure 19). Where not indicated, the check will only be made using the conditions defined in this view.
9 TRANSACTIONS IN DATAPORT

DataPort allows for transactions to be defined, using the following clauses:

- **BEGIN**. Allows for a transaction to begin.

- **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. Therefore, the main type of DataPort sources, the views of which can take part in transactions, are JDBC/ODBC-type data sources (see sections 17.3.1 and 17.3.2).

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 17.4.11 and 10.3).

It is possible to specify whether a data source supports distributed transactions by using the `supportsDistributedTransactions` property of the source configuration (see sections 17.3.10 and 17.4.12).
10 STORED PROCEDURES

DataPort supports the creation of stored procedures written in JAVA language. This section describes how to create them and import them in DataPort using VQL language. 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.

10.1 IMPORTING A STORED PROCEDURE

The statement `CREATE PROCEDURE` allows for a new stored procedure to be added to the DataPort server. Figure 28 shows its syntax.

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

Figure 28 CREATE PROCEDURE syntax

The CLASSNAME clause indicates the name of the JAVA class implementing the stored procedure. This class must be present in the CLASSPATH of the DataPort server (see the Post-Installation Tasks section of the Administrator Guide[3]). 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 recalculating of schemas and capabilities of the derived views using said procedure.

Once created, a stored procedure can be modified using the `ALTER PROCEDURE` statement, the syntax of which is shown in Figure 30.

```
ALTER PROCEDURE <name:identifier>
  [CLASSNAME <className:literal>]
  [CLASSPATH <classPath:literal>]
```

Figure 29 ALTER PROCEDURE syntax

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

10.2 USE OF STORED PROCEDURES

A stored procedure is called up using the `CALL` statement. The syntax is shown in Figure 30.

```
CALL <procedureName:identifier> (  
  [ <paramValue:literal> 
    [ , <paramValue:literal> ]* ]  
)  
[ CONTEXT ( "il8n" = <literal> ) ] [ TRACE ]
```

Figure 30 Syntax of the CALL statement
For example, the following statement calls up the stored procedure `DropIncidence`, passing it a single numeric-type parameter:

```
CALL DropIncidence(5)
```

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:

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

In this case, we shall use the stored procedure `CalculateAvgRevenue` as an example, which receives a parameter of the type array of registers 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 turnover of the indicated clients.

### 10.3 CREATION OF NEW STORED PROCEDURES

The necessary classes and interfaces for creating new stored procedures are in the `com.denodo.vdb.engine.storedprocedure` package. This section briefly describes the use of its main classes. See the Javadoc documentation [4] for further details on classes and operations.

Any stored procedure may extend the `AbstractStoredProcedure` class, which implements the `StoredProcedure` and `StoredProcedureExecutor` interfaces. The following methods can be rewritten:

- `public void initialize(DatabaseEnvironment environment)`. Method called up when initializing the stored procedure (which implementation is mandatory). It receives a `DatabaseEnvironment`-type object as a parameter, which provides methods of use for communications with the DataPort server. These methods allow for the following actions to be taken (see Javadoc documentation [4] for further details):
  - Execute statements on the DataPort server (`executeQuery`, `executeUpdate` methods),
  - Obtain references to other stored procedures in the server (`lookupProcedure` method) in order to execute them,
  - Obtain references to a server function (`lookupFunction` method) in order to execute them,
  - Create transactions (`createTransaction` method),
  - Join a stored procedure to the current transaction (`joinTransaction` method),
  - Write in the server log (`log` method),
o 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(). The description of the stored procedure must be returned (which implementation is mandatory).

- public String getName(). The name of the stored procedure must be returned (which implementation is mandatory).

- void prepare(). The stored procedure can optionally rewrite this method to prepare the current transaction.

- void commit(). The stored procedure can optionally rewrite this method to confirm the current transaction.

- void rollback(). The stored procedure can optionally rewrite this method to undo the current transaction.

- public StoredProcedureParameter[] getParameters(). Method that must specify the input and output parameters of the stored procedure (which implementation is mandatory). These parameters are returned as an array of StoredProcedureParameter objects. Each StoredProcedureParameter object specifies the name, type and address (input or output) of a parameter. In the event of the parameter being of the compound type, an array of StoredProcedureParameter objects must also be specified to describe its fields. See Javadoc documentation [4] for more details.

- public void doCall(Object[] inputValues) throws StoredProcedureException. Method called up to execute the stored procedure (which implementation is mandatory).

- public int getNumOfAffectedRows(). This returns the number of data entities affected by the execution of the procedure (which implementation is mandatory).

The AbstractStoredProcedure class also provides the following methods:

- public StoredProcedureResultSet getProcedureResultSet(). Method used to obtain the StoredProcedureResultSet object associated with the stored procedure. This object contains the results to 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 to create 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 to create a struct SQL-type object. This is required when the stored procedure returns compound-type values.

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.

10.4 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 allows for the log level to be altered for a certain log category. Its change is non-persistent between different server executions.
11 DEFINING OTHER ELEMENTS OF THE CATALOG

11.1 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 18.1 for an explanation of how to handle the compound types array and register.

Figure 31 shows the syntax of the statement CREATE TYPE.

```
CREATE 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 31** 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 32 an enumerated data type is created to represent the days of the week in English.

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

**Figure 32** Creating an enumerated data type

To create a new register type, the type of elements it contains must be indicated and a name assigned to them. In Figure 33 a register data type is created that contains the personal data referring to a person: the name (attribute NAME of text-type), the 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
};
```

Defining Other elements of the Catalog
When defining an array data type the name of the register type of the elements it contains must be indicated. In Figure 34 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 34 Creating a data type array and the register type it contains

### 11.2 DEFINITION OF 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 4.6).

- **i18n.** These represent internationalization configuration referring to a specific location. Some examples of parameters configured through these files are: currency, symbols used as decimal and thousands separators for currency, date format, etc. See section 3.2 for more details.

- **inputrewrite.** These are used to translate condition values to the format of the source to delegate a query to it. These are used by the condition transformation function (that are part of an input rewriting rule) called MAP().

- **outputrewrite.** These maps are used to convert enumerations, to adapt them to the integration format using the format of a specific source. This type of map is used in the output transformation function — that comprises a result rewriting rule — called MAP().

Maps can be created with Virtual DataPort through the statement CREATE MAP. The syntax is shown in Figure 35. This statement allows different types of maps to be created. For this the map type should be indicated: i18n, inputrewritemap or outputrewritemap or simple, the map name that identifies it and the list of key-value pairs that form part of same.

```
CREATE MAP { I18N|SIMPLE|INPUTREWRITEMAP|OUTPUTREWRITEMAP } <name:identifier> ( [ <name:literal> = <value:literal> ]+ )
```

Figure 35 Syntax of the statement CREATE MAP

Figure 36 shows an example of how an map of type inputrewrite is created.

```
CREATE MAP INPUTREWRITEMAP daysOfWeek ( 'lunes' = "TIMETABLE = 'Monday'" ,
    'martes' = "TIMETABLE = 'Tuesday'"
    'miercoles' = "TIMETABLE = 'Wednesday'"
    'jueves' = "TIMETABLE = 'Thursday'"
    'viernes' = "TIMETABLE = 'Friday'"
```

Figure 36 Creating a map daysOfWeek of type inputrewrite
'sábado' = "TIMETABLE = 'Saturday'"
'domingo' = "TIMETABLE = 'Sunday'"
};

**Figure 36** Creation of an map of type *inputrewrite*
12 CREATING DATABASES, USERS AND ACCESS LEVELS

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

Section 12.1 describes the concept of databases as understood in the context of a Virtual DataPort server. Section 12.2 describes the general concepts of user and access right management in DataPort. Finally, section 12.3 describes the VQL commands for managing this structure.

12.1 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 system sources). A Virtual DataPort database represents a virtual schema comprised of a series of DataSources, Wrappers, views and base relations.

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 can not be deleted.

12.2 USER AND ACCESS STRUCTURES IN VIRTUAL DATAPORT

12.2.1 Types of users

Denodo Virtual DataPort distinguishes two types of users:

- **Administrators**: These 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 installed 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.

12.2.2 Types of access rights

Virtual DataPort access rights are applied to a specific user to delimit the actions he/she 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 access to a database on a global level, 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 DESCR 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 access to a database on a global level, he/she can carry out the following tasks on same:
  - Creating DataSources, views, stored procedures and base relations on the database (corresponds to the VQL CREATE command).

- **Write access:** Having write access implies that you automatically also have read access. If a user has this access to 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 DataSource of the database he/she has created, but cannot delete DataSources 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 DataSource of the database he/she has created, but cannot modify DataSources created by other users (corresponds to the VQL ALTER command).

- **Connection access:** If a user has this access to a database, then he/she can connect to same, otherwise no. This type of access is useful if, for example, you wish to temporarily revoke user access to a database without having to modify his/her 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 access to 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 DESCR command).
  - Execute queries against the view of 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 access to a database, but does have it 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 access implies that you automatically also have read access. If a user has this access to 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 for tuples to be inserted in the view through INSERT statements. Not applicable to stored procedures.

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

### 12.3 VQL STATEMENTS OF DATABASES, USERS AND ACCESS

To manage the databases, users and access rights of a Virtual DataPort server, access must be through an administrator-type user and no database need to be specified in the server 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.

#### 12.3.1 Creating Databases

The VQL `CREATE DATABASE` statement allows an administrator user to create a new database in the server, indicating a name for the new database and, optionally, a description of same. Figure 37 shows the syntax of the `CREATE DATABASE` statement. Use of the user privilege assignment options is described in section 12.3.6.

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

*Figure 37 Syntax of the `CREATE DATABASE` statement*

#### 12.3.2 Modifying and Deleting Databases

To view the list of current databases in the server the `LIST` command should be used (see section 14). 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 38).

```sql
ALTER DATABASE <name:identifier> [ <description:literal> ] [ I18N {DEFAULT | <name:identifier>}] [ CACHE {
   [ DEFAULT | [ON | OFF ] {
      [MAINTAINERPERIOD <seconds:integer>] [TIMETOLIVE <seconds:integer>] [DATASOURCE {DEFAULT | CUSTOM}]
   } ]
} [ SWAP {
   [ DEFAULT | [ON | OFF ] {
      [SWAPSIZE <megabytes:integer>] [BLOCKSIZE <megabytes:integer>]
   }
} [ <grant> ]]*
```
Through this statement, it is possible to modify user access privileges for the database (see section 12.3.6.1) and the default preferences in the database for cache configuration (see section 18.2.2) and the swapping to disk of large queries (see section 18.2.3).

The DESC command (see section 13) allows data to be obtained in relation to 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 to the indicated database.

An administrator user can also delete a database from the management system using the DROP command (see section 15). Note than when a database is deleted all its components are eliminated: DataSources, views, base relations, etc.

### 12.3.3 Creating users

The CREATE USER statement (see Figure 39) allows a new user to be created 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, an optional description may also be included. The create statement also specifies, if it is a new administrator user (ADMIN modifier) or a normal user. The ENCRYPTED modifier specifies that the password is already coded and, as such, does not need to be coded again.

Users can be authenticated against DataPort or against an LDAP-type data source registered in DataPort (see section 17.3.8). The second case is specified using the LDAP modifier. In this case, two additional pieces of data must be provided:

- LDAP server to use [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 against that with which it was authenticated 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 12.3.6.
NOTE: If a LDAP datasource is deleted on cascade (see section 15), then the users depending on it will be also deleted. This operation can only be executed by an administrator user.

12.3.4 Modifying and Deleting users

The LIST statement (see section 14) allows a list of users of the management system to be obtained. Data relative to a user can be obtained as well as the access rights to the different databases and views of same through the command DESC (see section 13). An administrator user can access all the data of any user. The remaining users can only obtain their own data.

An administrator user can eliminate users of the management system using the DROP statement (see section 15). The predefined "admin" administrator cannot be deleted.

12.3.4.1 Modifying user data

Any user can change their access code and description using the ALTER USER statement (see Figure 40). In the case of a user being authenticated against an LDAP server, the server data can also be modified (see section 12.3.3). It is also possible to modify the privileges of a user (see section 12.3.6).

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

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

```sql
<grant> ::= (see section 12.3.6.2)
```

Figure 40 Syntax of the ALTER USER statement

12.3.5 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 41).

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

Figure 41 Syntax of the CONNECT and CLOSE statements

The CONNECT command allows a user name and password to be indicated 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.

12.3.6 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.

12.3.6.1 Specifying privileges by databases

Using the `CREATE DATABASE` (Figure 37) or `ALTER DATABASE` (Figure 38) statements, it is possible to specify the access rights of the different server users by databases, making use of the `GRANT` and `REVOKE` clauses.

Figure 42 shows the syntax of these clauses for assigning user access rights on a database global level. On a database level it is possible to grant or revoke all access rights (`ALL PRIVILEGES`) or a list of the following access rights:

- **CONNECT**: Allows the indicated user to connect to the database. If a user does not have this access right to a database, the other privileges are not considered.
- **CREATE**: Allows a user to create new elements in the server catalog.
- **READ**: Allows a user to access all the views and stored procedures of the indicated database.
- **WRITE**: Allows the specified user to modify or delete any view and/or stored procedures of the indicated database. Write access implies read access.

```
<grant> ::= 
  GRANT <database privileges> TO <user:identifier>
  REVOKE <database privileges> TO <user:identifier>

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

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

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

**Figure 42** Syntax of the GRANT/REVOKE clauses for Databases

12.3.6.2 Specifying privileges by user

User privileges can also be assigned, when the user is created or once created, with the statements `CREATE USER` (Figure 39) or `ALTER USER` (Figure 40), respectively.

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 43 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**: Allows the user to connect to the indicated database. If the user does not have this access to a database, the other privileges are not considered.
- **CREATE**: Allows a user to create new elements in the server catalog.
- **READ**: Allows a user to access all the views and stored procedures of the indicated database.
- **WRITE**: Allows the user to modify or delete any view and stored procedure of the indicated database. Write access implies read access.
Figure 43 Syntax of the clauses GRANT/REVOKE for Databases

Figure 44 shows the syntax of these clauses for assigning user access rights to individual views and/or stored procedures. These assignments are effective, 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 44  Syntax of the clauses GRANT/REVOKE for views

Figure 45 shows an example in which two databases are created, “database1” and “database2”, and a user “user1” to whom the following privileges are assigned in relation to “database1” and “database2”:
- has full privileges to “database1”
- has connection access to “database2” and create access, but only 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 45  Example of assigning privileges to users
13 DESCRIBING CATALOG ELEMENTS

The previous sections explain the statements that allow some of the elements of the Virtual DataPort data catalog or dictionary to be created and altered: base relations, derived views, wrappers, data sources, data types, maps, databases and users.

Virtual DataPort allows the current status of the main elements that belong to the catalog to be visualized through the statement DESC.

| DESC DATABASE | USER | TYPE | PROCEDURE | VIEW [TREE] | <name:identifier> |
| DESC DATASOURCE { ARN | CUSTOM | DF | GS | JDBC | LDAP | ODBC | WS | XML } | <name:identifier> |
| DESC MAP { I18N | INPUTREWRITE | OUTPUTREWRITE | SIMPLE } | <name:identifier> |
| DESC OPERATOR | <name:operator> | <type:identifier> |
| DESC PROCEDURE AS VIEW | <name:identifier> | { [<procedureParameter>] } | <procedureParameter> ::= <value> |
| DESC WRAPPER { ARN | CUSTOM | DF | GS | ITP | JDBC | ODBC | WS | XML } | <name:identifier> |
| DESC SESSION |
| DESC VQL | VIEW | PROCEDURE | TYPE | <name:identifier> |
| DESC VQL WRAPPER { ARN | CUSTOM | GS | DF | ITP | JDBC | ODBC | WS | XML } | <name:identifier> |
| DESC VQL MAP { I18N | INPUTREWRITE | OUTPUTREWRITE | SIMPLE } | <name:identifier> |
| DESC VQL DATABASE | <name:identifier> |

**Figure 46** Syntax of the statement DESC

The available options for describing different catalog elements are given in Figure 46.

The first group of statements allows different catalog elements to be described:

- The first statement allows the description of a database, a user, a data type, a stored procedure or a view to be requested from its name. Optionally, when a view is described, the modifier TREE can be indicated, which allows the group of views on which the current view is defined to be obtained together with the relational algebra operators that combine them.
- The DESC DATASOURCE statement allows the different data source types defined in the catalog to be viewed.
- The DESC MAP statement allows the content of a map of the type i18n, inputrewrite or outputrewrite to be viewed.
- The DESC OPERATOR statement requests 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 10.2). In this case, the procedure call-up parameters must be specified.
- The **DESC WRAPPER** statement allows the different wrapper types defined in the catalog to be viewed.

The **DESC SESSION** sentence allows a user to obtain the name of the database he or she is connected to, along with the login name used in the connection.

The group of statements **DESC VQL** allows the group of **VQL** statements necessary to create a view, stored procedure, data type, datasource or wrapper of one of the types specified to be viewed. Both the statements required by the current view and those required to create all the views it depends on are displayed. Likewise, the statements to create types, wrappers and data sources required to completely define the element of the selected catalog are displayed. The statement for deleting the element is shown before the creation statement of an element, whereby execution of the group of statements shown will result in the complete reconstruction of this element in the catalog (deleting any previous occurrences that may have existed).

### 13.1 EXPORTING METADATA

The **DESC VQL DATABASE** sentence allows exporting all the metadata from a given Virtual DataPort database, or even all the metadata from the server. This is especially useful for backup and migration purposes.

If the **DESC VQL DATABASE** sentence specifies the name of a database, all the metadata from the given database will be exported. The exported file will not include neither user definitions nor the active permissions assignments for the database. The **CREATE DATABASE** VQL sentence required for creating the database is also not included.

If the **DESC VQL DATABASE** sentence is used without specifying the name of a database, then all the Server metadata will be exported, including every database, user definition and active permission assignments.
14  LISTING ELEMENTS IN THE CATALOG

The LIST statement allows the elements in the catalog to be listed. Figure 47 shows the different options for using this statement:

- The first allows the list of all the databases, users or internationalization configurations to be requested.
- LIST TYPES shows all data types from the catalog or those of a certain characteristic (enumerated, array or register).
- LIST VIEWS allows the base relations or all the views defined in the server to be listed.
- LIST PROCEDURES allows the stored procedures defined in the server to be listed.
- LIST PATTERNS allows the rewriting rules - whether of the type raw or not - existing for a search method of a relation or view to be listed.
- LIST OPERATORS allows the operators that act on a specific data type to be listed, that is, those that allow values of a specific data type as operands.
- LIST WRAPPERS shows the enumeration of wrappers of the specified type (see section 17).
- LIST DATASOURCES shows the list of data origins of the type specified (see section 17.2).
- LIST MAPS allows all the maps of the type simple, i18n, inputrewrite or outputrewrite to be listed.

```
LIST { DATABASES | USERS | I18NS }
LIST TYPES [ ENUMERATED | ARRAY | REGISTER ]
LIST VIEWS [ BASE | ALL ]
LIST { INPUT | OUTPUT } PATTERNS [RAW] <view:identifier> <container:identifier>
LIST OPERATORS [ <type:identifier> ]
LIST WRAPPERS { ARN | CUSTOM | DF | GS | ITP | JDBC | MY | ODBC | WS | XML }
LIST DATASOURCES { ARN | CUSTOM | DF | GS | JDBC | LDAP [ALL] | ODBC | WS | XML }
LIST MAPS { I18N | INPUTREWRITE | OUTPUTREWRITE | SIMPLE }
LIST PROCEDURES
```

Figure 47  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;
```

To list the input rewriting rules of the type not raw for the search method shopview_sm1 of the base view shopview execute:

```
LIST MAPS INPUTREWRITE shopview_sm1 shopview;
```
LIST INPUT PATTERNS shopview shopview_sm1;

And to list the operators that operate on the data type int use the statement:

LIST OPERATORS int;
15 REMOVING ELEMENTS FROM THE CATALOG

The Virtual DataPort server also allows a specific element to be removed from the data dictionary using the statement DROP.

```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 { ARN | CUSTOM | DF | GS | ITP | JDBC | ODBC | WS | XML } [ IF EXISTS ] <name:identifier> [ CASCADE ]
DROP DATASOURCE { ARN | CUSTOM | DF | GS | JDBC | LDAP | ODBC | WS | XML } [ IF EXISTS ] <name:identifier> [ CASCADE ]
DROP MAP { INPUTREWRITE | OUTPUTREWRITE | I18N | SIMPLE } [ IF EXISTS ] <name:identifier>
DROP PROCEDURE [ IF EXISTS ] <name:identifier>
```

Figure 48 shows the different uses of the DROP statement:
- Removal of a database or a user from the system.
- Removal of a data type.
- Removal of a view (base or derived) from its name.
- Removal of a specific wrapper (see section 17) or data source (see section 17.3), indicating its type and name.
- Removal of a specific data dictionary map using its type (i18n, inputrewrite, outputrewrite) and its name.
- Removal of a stored procedure.

The IF EXISTS 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 may 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:

```sql
DROP VIEW shopview;
```

To remove the ITPilot wrapper called shopview simply execute:

```sql
DROP WRAPPER ITP shopview;
```
And to remove the map type \textit{i18n es_euro} the following statement is used:

\texttt{DROP MAP I18N es_euro;}
16 OTHER COMMANDS

16.1 DELETING INPUTS FROM THE CACHE OF THE DATA DICTIONARY

The content of the Virtual DataPort catalog is stored on disk according to a specific metadata storage policy. With the aim of reducing the access time to catalog elements the server stores memory caches of some of the elements such as: views, data types, wrappers, data sources, users, databases and maps.

The system provides a statement that allows the content of the different cache systems it uses to be deleted through the statement CLEAR CACHE (see Figure 49). In this manner, the metadata are forced to reload from their permanent storage (may have changed).

```
CLEAR CACHE OF { ALL | VIEWS | TYPES | WRAPPERS | DATASOURCES | USERS | DATABASES | PROCEDURES |
  MAPS [I18N | INPUTREWRITE | OUTPUTREWRITE | SIMPLE ] }
```

Figure 49 Syntax of the statement CLEAR CACHE

To delete the content of the cache referring to the views, data types, wrappers, data sources, users or databases the statement CLEAR CACHE should be parameterized with the options VIEWS, TYPES, WRAPPERS, DATASOURCES, USERS, DATABASES and PROCEDURES, respectively.

Likewise, to eliminate the map and internationalization caches and those used by the MAP() input and output transformation functions (that is, maps of the type inputrewrite and outputrewrite) the statement should be parameterized with the option MAPS SIMPLE, I18N INPUTREWRITE, OUTPUTREWRITE, respectively.

Finally, the statement CLEAR CACHE ALL deletes all the metadata cache existing in Virtual DataPort.

16.2 HELP COMMAND

Virtual DataPort includes a help statement, HELP, which gives the user a detailed overview of the syntax of all the existing commands. The syntax of the HELP command is shown in Figure 50.

If it is not assigned any parameters, the HELP statement presents its own syntax. Optionally, it allows as a parameter the command name for which help is required. For example, the statement of Figure 51 allows the user to know in detail the syntax of the command ALTER TABLE.

```
HELP <topic>

<topic> ::= ...
```

Figure 50 Syntax of the statement HELP
XML }
| BEGIN
| CALL
| CLEAR CACHE
| CLOSE
| COMMIT
| CONNECT
| CREATE DATABASE
| CREATE MAP
| CREATE PROCEDURE
| CREATE TABLE
| CREATE TYPE
| CREATE USER
| CREATE VIEW
| CREATE WRAPPER { ARN | CUSTOM | DF | GS | ITP | JDBC | ODBC | WS | XML }
| CREATE DATASOURCE { ARN | CUSTOM | DF | GS | JDBC | LDAP | ODBC | WS | XML }
| DELETE
| DESC
| DROP
| HELP HELP
| INSERT
| LIST
| QUERY WRAPPER { ARN | CUSTOM | DF | GS | ITP | JDBC | MY | ODBC | WS | XML }
| ROLLBACK
| SELECT
| UPDATE
| XML2BIN

**Figure 50** Syntax of the statement HELP

```
HELP ALTER TABLE
```

**Figure 51** Syntax to request help on the command ALTER TABLE

Detailed information on the general VQL syntax can be obtained using the HELP HELP statement.
17 GENERATING WRAPPERS AND DATASOURCES

Wrappers are components responsible for offering the server an overall view of the sources according to a common model. Each search method in a base relation has an associated wrapper, which is in charge of receiving the queries issued in relation to same, transforming them into queries to the source and obtaining the results, returning them to the logical layer in accordance with a format that is compatible with 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:

- **ITPilot**: This is used to incorporate wrappers for semi-structured sources created using Denodo ITPilot into the system [6]. 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 (see Denodo ITPilot documentation [6]).

- **JDBC**: Extracts data from a remote database via JDBC.

- **ODBC**: This extracts data from a remote database via ODBC.

- **Web Services**: Extracts data invoking operations defined by Web services.

- **XML**: These allow data encapsulated in XML files, optionally following a specific DTD or schema, to be extracted.

- **DF**: Extract data from flat text files that use specific characters as tuple and field delimiters. Amongst the files supported are files in CSV format that tend to be obtained as a result of downloading data from databases or Excel documents.

- **ARACNE**: This provides access to indexes on non-structured data created using Denodo Aracne [16].

- **GOOGLE MINI**: This provides access to indexes on non-structured data created using the search tool Google Mini [17].

- **CUSTOM (also called MY)**: Extracts 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 datasource elements for all wrappers, except ITPilot-type ones, to encapsulate certain data on data source access and configuration. There is also a type of data source to represent the LDAP servers that can be used to authenticate DataPort users (see section 12.3.3). LDAP data sources do not have associated wrappers.

This section describes how to create and modify wrappers (and their *data sources*) of any type in Virtual DataPort.

The remainder of this section is structured as follows. Sections 17.1 and 17.2 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 in Virtual DataPort. Section 17.3 specifies how to add to the system data sources of the various available types. Finally, section 17.4 shows how to create wrappers for each of these source types.
17.1 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 Administrator Guide [3]).

<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,</td>
<td>date</td>
</tr>
<tr>
<td>java.sql.Date, java.sql.Timestamp,</td>
<td></td>
</tr>
<tr>
<td>java.sql.Time</td>
<td></td>
</tr>
<tr>
<td>byte[], java.sql.Blob</td>
<td>blob</td>
</tr>
</tbody>
</table>

Table 2 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>long, java.lang.Long</td>
<td>time</td>
</tr>
<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>

Table 3 Other valid conversions between JAVA types and Virtual DataPort types

Likewise, wrappers can provide compound elements such as arrays and registers that are directly associated with VDP arrays and registers.

17.1.1 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 through DataPort registers and arrays respectively.
### 17.1.1.1 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>JDBCArrayTypeVO 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.String</td>
</tr>
<tr>
<td>OTHER</td>
<td>JDBCRegisterTypeVO</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>JDBCRegisterTypeVO</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>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

**Table 4** Type Conversion Tables for JDBC Wrappers

Other types are converted to `java.lang.String`. There may be different conversions depending on the database version accessed.

### 17.1.1.2 Type conversion table for ODBC wrappers

For ODBC wrappers the same conversions are applied as for the JDBC wrappers on which they are based.

### 17.1.1.3 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>SOAP Types</td>
<td>Java Types</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------</td>
</tr>
<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>

Table 1: ITPilot-type conversions

The Web source wrappers generated using versions of ITPilot prior to 4.0 do not provide data related to the type of elements obtained, whereby they are encapsulated using the java class java.lang.String.

17.1.1.4 Type conversion table for Web Services wrappers

The other elements are treated as Java objects using the reflection API to access the properties and in this way to rebuild the structure.
17.1.1.5 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></td>
</tr>
<tr>
<td>gYear</td>
<td></td>
</tr>
<tr>
<td>gMonth</td>
<td></td>
</tr>
<tr>
<td>gDay</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>unsignedLong</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>unsignedByte</td>
<td></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></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></td>
</tr>
<tr>
<td>token</td>
<td></td>
</tr>
<tr>
<td>base64Binary</td>
<td></td>
</tr>
<tr>
<td>hexBinary</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td></td>
</tr>
<tr>
<td>dateTime</td>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td></td>
</tr>
<tr>
<td>gYearMonth</td>
<td></td>
</tr>
<tr>
<td>gMonthDay</td>
<td></td>
</tr>
</tbody>
</table>

Table 6  Type Conversion Table for XML Wrappers

17.1.1.6 Type conversion table for delimited file wrappers

A DF wrapper does not have metadata that allow data values stored in a file to be identified with its types, whereby a DF wrapper always processes elements obtained encapsulated in java class java.lang.String.

17.1.1.7 Type conversion table for CUSTOM wrappers

A CUSTOM wrapper indicates the types of its fields with Java classes and, therefore, requires no conversion.

17.1.1.8 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 17.4.5.2.
17.1.1.9 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 17.3.7.

17.2 SPECIFYING PATHS IN VIRTUAL DATAPORT

Virtual DataPort uses path specification in various points of data source and wrapper creation. These paths allow a specific resource to be located (file, Web page etc.) as required during the process.

Three types of paths exist 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:
  
  o The class name used to implement the connection used by the path. For this type of path one sole connection class is provided: LocalConnection.
  
  o 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:
  
  o The class name used to implement the connection used by the path. For this type of path the server provides two different classes:
    
    ▪ http.HTTPClientConnection: 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 to obtain the response headings of the http request made. 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.IEBrowserConnection: Makes a connection to a Web server using a Denodo ITPIlot browser pool [6]. These browsers are capable of executing advanced navigation sequences on the Microsoft Internet Explorer browser [10] written in the language described by ITPIlot. Not assigned any parameter.

  o Access pattern (urlpattern). 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 an http request to be specified (expressed in the normal format used for GET requests). ITPIlot [6] 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 18.4).
- 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.

- FTP: 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 leading to the resource (host:port/path/file).
  - User identifier that should be used for access and
  - Password for this user.

17.3 CREATING DATA SOURCES

Before describing in detail each of the existing wrapper types in Virtual DataPort it is important to introduce the concept of DataSource. DataSources are used by the wrappers to locate data origins. This allows the source localizer to be reused based on a name and also connection pools to be maintained and configured on same (if the concept is applicable to this source type). For instance, the use of connection pools is necessary for efficiency when processing relational data sources.

The following sections describe the manual creation process for each DataSource type.

**NOTE**: It is strongly recommended that the wrapper and datasource creation process be undertaken graphically using the DataPort administration tool (see [3]).

17.3.1 JDBC DataSources

To facilitate access to JDBC sources different parameters can be specified in JDBC-type DataSources for the connections pool and pool of PreparedStatements to avoid the costly process of creating a new connection for each new query. Where all the possible parameters are not specified implicitly in the create statement, the default values will be used.

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 in question.
- 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 (zero implies no limit).

Data source configuration parameters [SOURCECONFIGURATION]. Virtual DataPort allows specific characteristics of underlying sources to be indicated, so that they are taken into account when executing statements on them. See section 17.3.10 for further details.

The **OR_REPLACE** modifier can also be specified in the datasource creation statement. In this case, if a datasource 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 with the options of the different parameter groups.

```
CREATE [ OR REPLACE ] DATASOURCE JDBC <name:identifier> 
  DRIVERCLASSNAME=<literal> 
  DATABASEURI=<literal> 
  USERNAME=<literal> 
  USERPASSWORD=<literal> 
  [ CLASSPATH=<literal> ] 
  [ 
    DATABASENAME=<literal> 
    DATABASEVERSION=<literal> 
  ] 
  [ 
    VALIDATIONQUERY=<literal> 
    INITIALSIZE=<integer> 
    MAXACTIVE=<integer> 
  ] 
  [ 
    VALIDATIONQUERY=<literal> 
    INITIALSIZE=<integer> 
    MAXIDLE=<integer> 
    MINIDLE=<integer> 
    MAXACTIVE=<integer> 
    EXHAUSTEDACTION=<integer> 
    TESTONBorrow=<boolean> 
    TESTONRETURN=<boolean> 
    TESTWHILEIDLE=<boolean> 
    [ 
      TIMEBETWEENEVICITION=<integer> 
      NUMTESTPEREVICITION=<integer> 
      MINEVIDECTABLETIME=<integer> 
    ] 
    POOLPREPAREDSTATEMENTS=<boolean> 
    MAXSLEEPINGPS=<integer> 
    INITIALCAPACITYPS=<integer> 
  ] 
) 
```
Figure 52 Syntax of the CREATE DATASOURCE JDBC statement

A JDBC data source modification statement exists (ALTER DATASOURCE JDBC). The syntax allows the same parameters as the creation statement to be indicated.

```sql
ALTER DATASOURCE JDBC <name:identifier>
  DRIVERCLASSNAME=<literal>
  DATABASEURI=<literal>
  USERNAME=<literal>
  USERPASSWORD=<literal>
  [ CLASSPATH=<literal> ]
```
```
DATABASENAME=<literal>
DATABASEVERSION=<literal>
]
[
VALIDATIONQUERY=<literal>
INITIALSIZE=<integer>
MAXACTIVE=<integer>
]
[
VALIDATIONQUERY=<literal>
INITIALSIZE=<integer>
MAXACTIVE=<integer>
MAXIDLE=<integer>
MINIDLE=<integer>
MAXACTIVE=<integer>
EXHAUSTEDACTION=<integer>
TESTONBORROW=<boolean>
TESTONRETURN=<boolean>
TESTWHILEIDLE=<boolean>
[
  TIMEBETWEENEVICTION=<integer>
  NUMTESTPEREVICTION=<integer>
  MINEVIDECTABLETIME=<integer>
  [
    POOLPREPAREDSTATEMENTS=<boolean>
    MAXSLEEPINGPS=<integer>
    INITIALCAPACITYPS=<integer>
  ]
]
[
  SOURCECONFIGURATION ( [ <source configuration property>
   [, <source configuration property> ]* ] )
  ]
```

Figure 53 Syntax of the ALTER DATASOURCE JDBC statement

### 17.3.2 ODBC DataSources

Virtual DataPort allows databases accessible via ODBC to be defined as system sources.

Figure 54 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 JDBC datasources.

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.

Configuration of different parameters belonging to the data source can also be specified [SOURCECONFIGURATION], which will be taken into account by Virtual DataPort when executing statements on them. See section 17.3.10 for further details.
In the case of ODBC data sources, the driver class to be used for the connection to the management system may not be specified. For this, the DSN attribute should be specified instead of the DATABASEURI together with the DRIVERCLASSNAME. 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 management system 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. In any case, the connection between VDP and the ODBC management system or ODBC Database will be local.

```
CREATE [OR REPLACE] DATASOURCE ODBC <name:identifier>
{  DSN=<literal>
   | DATABASEURI=<literal>
   DRIVERCLASSNAME=<literal>
}
USERNAME=<literal>
USERPASSWORD=<literal>
[ PROPERTIES=<literal> ]
[ CLASSPATH=<literal> ]
[
   DATABASENAME=<literal>
   DATABASEVERSION=<literal>
]
[
   INITIALIZE=<integer>
   MAXACTIVE=<integer>
   VALIDATIONQUERY=<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 }
| 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 } |
| SUPPORTSFULLNOTEQOUTERJOIN = { true | false | DEFAULT } |
| SUPPORTSJOINONCONDITION = { true | false | DEFAULT } |
| SUPPORTSNATURALJOIN = { true | false | DEFAULT } |
| SUPPORTSUSINGJOIN = { true | false | DEFAULT } |
| DELEGATEAGGREGATEFUNCTIONS = { DEFAULT | ( <function:identifier>, <function:identifier>, ... ) } |
| DELEGATESCALARFUNCTIONS = { DEFAULT | ( <function:identifier>, ... ) } |
| DELEGATEOPERATORSLIST = { DEFAULT | ( <operator:identifier>, ... ) } |

Figure 54 Syntax of the CREATE DATASOURCE ODBC statement

As with JDBC data sources, an ODBC data source modification statement exists (ALTER DATASOURCE ODBC) with the same syntax as the creation statement.

ALTER DATASOURCE ODBC <name:identifier>
{  DSN=<literal>
   | DATABASEURI=<literal>
   DRIVERCLASSNAME=<literal>
 }  
USERNAME=<literal>  
USERPASSWORD=<literal>  
[ PROPERTIES=<literal> ]  
[ CLASSPATH=<literal> ]  
[ DATABASENAME=<literal>  
DATABASEVERSION=<literal> ]  
[ INITIALSIZE=<integer>
 MAXACTIVE=<integer>  
VALIDATIONQUERY=<literal> ]  
[ SOURCECONFIGURATION ( [ <source configuration property>, ... ] ) ]

Figure 55 Syntax of the ALTER DATASOURCE ODBC statement

17.3.3 DataSources for Web Services

To configure a Web service as data source the URI to the WSDL file that defines the Web Service must be specified. Figure 56 shows the create syntax of a datasource for a Web service.

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.
CREATE [OR REPLACE] DATASOURCE WS <name:identifier>
  WSDLURI <literal>

Figure 56 Syntax of the CREATE DATASOURCE WS statement

The modification statement of a data source of this type is similar.

ALTER DATASOURCE WS <name:identifier>
  WSDLURI = <literal>

Figure 57 Syntax of the ALTER DATASOURCE WS statement

A WSDL file allows one or several Web services to be defined, and each Web service can have various ports with one or various operations. A data source for Web services allows wrappers to be created modeling any of the operations defined.

17.3.4 XML DataSources

Virtual DataPort allows XML files to be defined 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 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 seek 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 datasource (see section 18.4).

The path specification in DataPort was described in section 17.2.

The creation syntax can be seen in Figure 58:

CREATE [OR REPLACE] DATASOURCE XML <name:identifier>
  [ { SCHEMA | DTD } <route>]
  ROUTE <route>

<route> ::= 
  LOCAL <connection class name:literal> <uri:literal>
  | HTTP <connection class name:literal> { GET | POST } <uri:literal>
  | FTP <connection class name:literal> <uri:literal>
  <login:literal> <password:literal>

Figure 58 Syntax of the CREATE DATASOURCE XML statement

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.

The syntax of the modification statement of an XML data source is shown below.

ALTER DATASOURCE XML <name:identifier>
  [ { SCHEMA | DTD } <route>]
  ROUTE <route>
17.3.5 DF DataSources

This data origin type allows Denodo Virtual DataPort to access the data contained in flat files in CSV format (Comma Separated Values) or similar.

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. This may include interpolation variables to parameterize the access path depending on the conditions of the query executed on the datasource (see section 18.4).
- COLUMNDELIMITER: Character string used as an element separator in the delimited file.
- 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 no 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.
- ENDDELIMITER: A JAVA regular expression identifying the position in the file where the system must stop searching for tuples. If no 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.
- HEADER: Allows specifying whether the first data tuple of the file should be used as header, that is, as metadata that provide the name of the different fields that make up a tuple of the delimited file.

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.

```
CREATE [OR REPLACE] DATASOURCE DF <name:identifier>
  ROUTE <route>
  COLUMNDELIMITER = <literal>
  [ ENDOFLINEDELIMITER = <literal> ]
  [ BEGINDELIMITER = <literal> [ISDATA] ]
  [ ENDDELIMITER = <literal> [ISDATA] ]
  [ HEADER = <boolean> ]
```

Figure 60 Syntax of the CREATE DATASOURCE DF statement

Figure 61 shows the syntax of the modification statement of a delimited file data source.

```
ALTER DATASOURCE DF <name:identifier>
```

Figure 59 Syntax of the ALTER DATASOURCE XML statement
**Figure 61** Syntax of the CREATE DATASOURCE DF statement

Path specification in DataPort was described in section 17.2.

### 17.3.6 Denodo Aracne DataSources

Virtual DataPort allows for a Denodo Aracne search server [16] to be imported 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 to be imported. The URI format is `host:port`, where `host` is the name of the machine from where the search engine is accessible and `port` the port through which it is run. This port is 4000 in the Aracne default installation.

The creation syntax can be seen in Figure 62:

```
CREATE [ OR REPLACE ] DATASOURCE ARN <name:identifier>
ARNURI = <literal>
```

**Figure 62** Syntax of the create statement of an Aracne 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 an Aracne data source.

```
ALTER DATASOURCE ARN <name:identifier>
ARNURI = <literal>
```

**Figure 63** Syntax of the modification statement of an Aracne data source

### 17.3.7 Google Mini DataSources

Virtual DataPort allows to import a Google Mini [17] 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 Mini search server to be imported. The URI format is `host:port`, where `host` is the name of the machine from where the search engine is accessible and `port` the port through which it is run.
The creation syntax can be seen in Figure 64:

```
CREATE [ OR REPLACE ] DATASOURCE GS <name:identifier> 
  GSURI = <literal>
```

**Figure 64** 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>
```

**Figure 65** Syntax of the modification statement of a Google Mini data source

### 17.3.8 LDAP DataSources

Virtual DataPort allows for an LDAP server to be imported as a data source. Imported LDAP servers can be used for DataPort users to authenticate against them (see section 12.3.3). The following parameters must be specified to add a new LDAP data source:

- **name**: Name to be given to the data source in Virtual DataPort.
- **URI**: Access URI to the LDAP server to be imported. The URI format is `ldap://host:port`, where `host` is the name of the machine from where the server is accessible and `port` the port through which it is run.

The creation syntax can be seen in Figure 66:

```
CREATE [ OR REPLACE ] DATASOURCE LDAP <name:identifier> 
  URI=<serverURI:literal>
```

**Figure 66** Syntax of the create statement of an LDAP 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 an Aracne data source.

```
ALTER DATASOURCE LDAP <name:identifier> 
  URI=<serverURI:literal>
```

**Figure 67** Syntax of the modification statement of an LDAP data source

### 17.3.9 Custom DataSources

Virtual DataPort allows for wrappers to be created ad-hoc for data sources for which no specific connector is provided by DataPort. To do so, two JAVA classes must be created to implement the required behavior (see section 17.4.11.3). Once this class has 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 17.4.5.2.

- **CLASSPATH**. (optional) Additional classpath required for running the wrapper.

The creation syntax can be seen in Figure 68:

```
CREATE [ OR REPLACE ] DATASOURCE CUSTOM <name:identifier>
CLASSNAME=<className:literal>
[ CLASSPATH=<classPath:literal> ]
```

**Figure 68** Syntax of the create statement of a Custom 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 Custom data source.

```
ALTER DATASOURCE CUSTOM <name:identifier>
CLASSNAME=<className:literal>
[ CLASSPATH=<classPath:literal> ]
```

**Figure 69** Syntax of the modification statement of a Custom data source

### 17.3.10 Data Source Configuration Properties

Virtual DataPort keeps properties for all of the data sources and wrappers to configure specific characteristics of underlying sources such as their distributed transaction support capacity or whether insert operations are allowed. This function enables the system administrator to optimally configure the characteristics of each data source and wrapper and, therefore, their combination and execution possibilities.

The properties of each data source can be configured by adding parameter/value pairs to the DataSource creation statement or graphically using the administration tool (see Virtual DataPort Administrator Guide [3]). 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 (DELEGATELEFTLITERAL, 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 (DELEGATENATURALOUTERJOIN, DS: JDBC, ODBC). This indicates whether the source allows for the Natural Outer Join operator to be delegated. The value is “false” by default for JDBC and ODBC sources.

• Delegate NOT Condition (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 (DELEGATEORCONDITION, 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, the list is made up of the `ABS`, `CEIL`, `CONCAT`, `DIV`, `FLOOR`, `GETDAY`, `GETHOUR`, `GETMINUTE`, `GETMONTH`, `GETYEAR`, `LEN`, `LOG`, `LOWER`, `MOD`, `MULT`, `NOW`, `POWER`, `REPLACE`, `ROUND`, `SQRT`, `SUBTRACT`, `SUM`, `TEXTCONSTANT`, `TRIM` and `UPPER` functions.

• **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 for properties to be specified on 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. At present, these properties only exist for the `contains` operator (see section 17.3.10.1).

**Example**: Suppose that the creation of a DataSource 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 70** Example of altering a DataSource configuration

Virtual DataPort has default values for some specific relational databases (MySQL, Oracle, Postgres, etc.) that may vary in relation to those described above.

### 17.3.10.1 CONTAINS Operator Configuration Properties

The CONTAINS operator allows for complex Boolean keyword searches to be made on text-type attributes from an index of external, non-structured data (e.g. Aracne and/or Google Mini data sources).

The syntax of the search language on non-structured data is described in section 19. However, the search options available depend on the capacities natively provided by the data source. Section 19.2 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 Operator Configuration Properties. 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.

- **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.

- **Supports Multi Wildcards First Position.** This takes the value `true`, if wildcards match with multiple characters (i.e. the wildcard '*') in the first position of a term.

- **Supports Multi Wildcards Rest Position.** This takes the value `true`, if wildcards match with multiple characters (i.e. the wildcard '*') 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 are supported without specifying a minimum similarity among the search terms and the agreements found.

- **Supports Fuzzy Terms With Minimum Relevance.** This takes the value `true`, if fuzzy searches are supported specifying a minimum similarity among the search terms and the agreements found.

- **Supports Proximity Terms Without Maximum Distance.** This takes the value `true`, if searches by proximity are supported without specifying a maximum distance among the terms of the search phrase.

- **Supports Proximity Terms With Maximum Distance.** This takes the value `true`, if searches by proximity are supported specifying a maximum distance among the terms of the search phrase.

- **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:

  \[
  \text{title contains } '(\text{term1 AND term2}) \text{ OR (term3)}' \]

• **Supports Grouping.** This takes the value true, if the combination of logic operators AND and OR in different query conditions is supported. For example:

  \[
  \text{title contains 'term1' AND (content contains 'term2' OR summary contains 'term3')} \]

### 17.4 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 datasource creation process be undertaken graphically using the DataPort administration tool (see [3]).

Previously, the concepts of **execution context** and **interpolation strings**, which will be used in creating some kinds of wrappers, are introduced in section 17.4.1 while general information about the schema metadata of the results returned by wrappers will be provided in section 17.4.2.

#### 17.4.1 Execution Context and Interpolation Strings

As already mentioned in previous sections, the mission of a source wrapper is to execute queries and/or updates on same in a manner that is transparent for the higher levels of the DataPort server.

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 ITPilot-type wrappers using versions of Denodo ITPilot prior to 4.0 (see section 77) and by JDBC wrappers using a pattern SQL query (see section 17.4.5.2). Details on the use of interpolation strings can be found in section 18.4.
17.4.2 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 are searchable fields (obligatory or optional). A wrapper will expect to receive query conditions on fields specified as obligatory in order to execute. The Java data types associated with the elements coming from the data source are also specified based on the conversion tables specified in section 17.1.1.
- **REGISTER**: formed by one or various fields, both simple and compound.
- **ARRAY**: lists formed of register-type fields.

These data allow base relations to be generated automatically from wrappers.

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.

17.4.3 JDBC Wrapper

A JDBC wrapper extracts data from a remote Database via JDBC. The syntax for creating a wrapper of this type is shown in Figure 71.

```
CREATE [ OR REPLACE ] WRAPPER JDBC <name:identifier>
  DATASOURCENAME=<name:identifier>
  [ RELATIONNAME=<name:literal> | SQLSENTENCE=<literal> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
  [ 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>

  <register field> ::=<name:identifier> [ = <mapping:literal> ] :
    REGISTER OF ( <field> [, <field>]* ) [ <inline constraints> ]*

  <inline constraint> ::=[ NOT ] NULL
  | [ NOT ] UPDATEABLE
  | ( { SOURTABE [ ASC | DESC ] | NOT SORTABLE } )
  | EXTERN

  <source configuration property> ::=ALLOWDELETE = { true | false | DEFAULT }
  | ALLOWINSERT = { true | false | DEFAULT }
```
### ALTER WRAPPER JDBC Syntax

```plaintext
ALTER WRAPPER JDBC <name:identifier>
[ DATASOURCENAME=<name:identifier> ]
[ RELATIONNAME=<name:identifier> | SQLSENTENCE=<literal> ]
[ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
[ SOURCECONFIGURATION ( [ <source configuration property> [, <source configuration property> ]* ] ) ]

<field> ::= (see CREATE WRAPPER JDBC for details)
<source configuration property> ::= (see CREATE WRAPPER JDBC for details)
```

**Figure 72** Syntax of the ALTER WRAPPER JDBC statement

To specify a JDBC-type wrapper it is needed to indicate the name of the JDBC datasource to be used (DATASOURCENAME) and data relative to the database specified by the datasource and which will be accessed by the wrapper.

Two mechanisms can be used to indicate the data from the database the wrapper will access to:

- Indicate the name of the table in the remote database (RELATIONNAME).
- Specify a SQL statement (SQLSENTENCE). The SQL statement can be an interpolation string (see section 18.4).

Another important consideration is that the results returned for the query should be compatible with the schema of the base relation to which said wrapper is linked in the server. More specifically, the attribute names obtained as a result of the query should coincide with those of the base relation and, furthermore, their values should be compatible with the data types in the base relation. To allow greater flexibility and independence between base relations and data sources optionally the JDBC wrapper allows the output schema of the data it will provide to be defined (see section 17.4.2). 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).

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) that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 72), and are explained in section 17.4.12.

#### 17.4.3.1 Specification of a table in the remote database

The first alternative in 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 are to be extracted.

On execution, the wrapper will receive a list of conditions and a list with the output field names requested from the wrapper. Based on these elements the wrapper will automatically construct a valid SQL statement to execute in the remote database management system.
17.4.3.2 Using a pattern SQL statement

The other mechanism for modeling the process for selecting data in a JDBC wrapper is based on defining a SQL statement with the query to be made to the database. Use of this mechanism can be useful, for example, to use the result of executing an *ad hoc* SQL query as a base relation, including a stored procedure on the remote database.

The specified SQL statement is an interpolation string susceptible to being parameterized with variables received from the execution context and with *interpolation functions* (see section 18.4 for details on same).

17.4.4 ODBC Wrapper

An ODBC wrapper allows data sources to be added that comply with the ODBC data interoperability standard in the Virtual DataPort system.

Figure 73 shows the syntax of creation of an ODBC wrapper that follows the same structure as that defined for a JDBC wrapper. To create a wrapper of this type it is necessary to specify the connection string to the data source (predefined ODBC datasource), the relation from which data are extracted or the pattern SQL query and the output schema that provides the wrapper along with its type structure. For more information, see section 17.4.3.

CREATE [ OR REPLACE ] WRAPPER ODBC <name:identifier>
DATASOURCENAME=<name:identifier>
{ RELATIONNAME=<name:literal> | SQLSENTENCE=<literal> }
{ OUTPUTSCHEMA ( <field> [, <field>] )* } 
{ 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>

<register field> ::= 
<name:identifier> [ = <mapping:literal> ] :
REGISTER OF ( <field> [, <field>] )* [ <inline constraints> ]*

<inline constraint> ::= 
[ NOT ] NULL
| [ NOT ] UPDATEABLE
| { SORTABLE [ ASC | DESC ] | NOT SORTABLE }
| EXTERN

<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 }

| SUPPORTSDISTRIBUTEDTRANSACTIONS = { true | false | DEFAULT }

**Figure 73** Syntax of the CREATE WRAPPER ODBC statement

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] that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 73), and are explained in section 17.4.12.
ITPilot Wrapper

ITPilot wrappers are used to import semi-structured data sources (typically semi-structured web sources) to the system. These sources may be accessible in the web, through the local filesystem or through a FTP service. This kind of wrappers require Denodo ITPilot [6] to execute (ITPilot also allows graphical creation and automatic maintenance 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 ITPilot wrappers is shown in Figure 75.

```
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> ]* ) ] ) ]

<field> ::=<name:identifier> [ <regexp> ] [ { OBL | OPT } ]
[ ( <alias:literal> [, <alias:literal>]* ) ]
[ <inline constraints> ]*  
| <name:identifier>:ARRAY OF ( <register field> ) [ <inline constraints> ]*  
| <name:register field>

<register field> ::=  
<name:identifier>:REGISTER OF ( <field> [, <field>]* )  
[ <inline constraints> ]*  

<sequence clause> ::=  
REGISTER ( <field> [, <field>]* )  
[ <inline constraints> ]*  

<route> ::=  
LOCAL <connection class name:literal> <specification:literal>
| HTTP <connection class name:literal> <specification:literal>
| FTP <connection class name:literal> <specification:literal>
```

Figure 74 Syntax of the ALTER WRAPPER ODBC statement

**17.4.5 ITPilot Wrapper**

ITPilot wrappers are used to import semi-structured data sources (typically semi-structured web sources) to the system. These sources may be accessible in the web, through the local filesystem or through a FTP service. This kind of wrappers require Denodo ITPilot [6] to execute (ITPilot also allows graphical creation and automatic maintenance 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 ITPilot wrappers is shown in Figure 75.
The syntax for modifying ITPilot wrappers is similar (Figure 76).

```sql
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)
<source configuration property> ::= (see CREATE WRAPPER ITP for details)
```

**Figure 76** Syntax of the ALTER WRAPPER ITP statement

There are two alternative ways of creating an ITPilot wrapper, depending on whether the version of ITPilot used is before or after version 4.0. Section 17.4.5.1 deals with wrappers created using ITPilot 4.0 or after and section Figure 77 deals with wrappers created with previous versions. The options common to both cases are described below.

The MAINTENANCE clause allows for the ITPilot automatic maintenance system for the wrapper to be enabled or disabled. See the ITPilot documentation [6] 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.

Certain wrapper properties can also be specified (SOURCECONFIGURATION) that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 75) and are explained in section 17.4.12.

**17.4.5.1 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 77 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 77 Example of ITPilot 4.0 wrapper

17.4.5.2 ITPilot wrappers with versions of ITPilot prior to 4.0

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 18.4).

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 17.2).

- **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 17.2, adding a data extraction specification that contains the resource noted by the defined path (specification should be written using the ITPilot DEXTL data extraction language [6]). In addition, access patterns can be parameterized using context variables and interpolation functions (see section 18.4).**

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 general metadata, indicated relative to the output schema provided by the wrapper (see section 17.4.2), can be extended by adding 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 ITPilot 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 [6] for more information). Additionally, both in the creation statement and the ITPilot
17.4.5.3 Substitutions

An ITPilot 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 asks for resolution.

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 18.4). The preconditions are specified as `<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 `TITLE` attribute and with the operator `containsor` (that is, "TITLE containsor 'values'"); to achieve it, a substitution would be created with a precondition "TITLE#containsor".

Figure 78 shows an ITPilot wrapper with a default sequence which uses an HTTP route, with a pattern access called ACCESSPAT1 (compliant with any of the formats supported by ITPilot [6]) and a data extraction specification DATAEXTRACTSPEC1 (written in the ITPilot data extraction language, known as DEXTL [6]).

Besides, a substitution is included, which is used in case the source is queried with the operator `containsor` over the `TITLE` attribute. Another sequence would be used, which consists in an HTTP route with ACCESSPAT2 access pattern and an extraction specification called 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 78 Creation of an ITPilot wrapper

17.4.6 Web Services Wrapper

Virtual DataPort supports the creation of wrappers for 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 79 shows the syntax of the VQL statement for creating a Web services wrapper.

```
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>

<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 79 Syntax of the CREATE WRAPPER WS statement

The modification syntax of a Web service wrapper is similar and is shown in Figure 80.

```
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>

<field> ::= (see CREATE WRAPPER WS for details)

<source configuration property> ::= (see CREATE WRAPPER WS for details)
```

Figure 80 Syntax of the ALTER WRAPPER WS statement
In addition to the Web service *datasource* name that identifies the WSDL definition file, it is necessary to indicate other parameters that unequivocally 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 from which the specific operation to be modeled is to be selected.
- **OPERATIONNAME**: Name of the operation to be modeled. There can be many 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).

Attributes of the selected operation messages and their type structure define the Web services wrapper output 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 as of Virtual DataPort version 3.1. These parameters will convert to DataPort compound types (see section 18.1) 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 6.3.1).

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>** ➔ references the parameter of the indicated position of the Web service operation.
- **$$** ➔ references the output parameter returned through invocation of the Web service operation.

This is the notation used for the elements of the first level (parameters and output of the Web service). For the other elements (fields of a result object or parameter of the Web service) the mapping is obtained from the real 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**) that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 79), and are explained in section 17.4.12.

### 17.4.7 XML Wrapper

Virtual DataPort supports the creation of wrappers that have XML data files as a source. Figure 81 shows the syntax for creating an XML wrapper.

```
CREATE [ OR REPLACE ] WRAPPER XML <name:identifier>
```
Generating Wrappers and Datasources

**Figure 81** Syntax of the CREATE WRAPPER XML statement

The modification syntax of an XML wrapper is similar and can be seen in Figure 82.

**Figure 82** Syntax of the ALTER WRAPPER XML statement

An XML wrapper is defined through an XML datasource 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. “Flattening” operations can be used (see section 6.1.2) to transform the resulting base relation in the required manner.

As with the other wrappers, the output schema of the data provided by the wrapper can be specified, whereby only those elements of interest from the XML document are selected and the name too can be changed (in mapping the name is specified in the wrapper; name stores the external name).

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 \texttt{SOURCECONFIGURATION} that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 81), and are explained in section 17.4.12.

### 17.4.8 DF wrapper

Virtual DataPort supports the creation of wrappers of CSV delimited files and similar. To create a wrapper of this type the name of the data source must be indicated (how to access the file that contains the data), \texttt{DATASOURCENAME}. Optionally, as with the other wrappers, the schema of data returned by the wrapper may be specified \texttt{OUTPUTSCHEMA}.

The \texttt{WRAPPER} creation statement accepts the \texttt{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 \texttt{SOURCECONFIGURATION} that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 83), and are explained in section 17.4.12.

\textbf{NOTE}: In this type of wrappers, "mapping" or specification of registers or arrays as elements of the output schema are not supported.

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 ] ]* ) }
```

\textbf{Figure 83} Syntax of the \texttt{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 84** Syntax of the `ALTER WRAPPER DF` statement

### 17.4.9 Denodo Aracne Wrapper

Virtual DataPort supports the creation of wrappers on indexes of non-structured data created using Denodo Aracne [16].

To create a wrapper of this type, the name of the data source – `DATASOURCENAME` – must be indicated along with the name of the Aracne handler – `HANDLERNAME` – used to create the wrapper.

As with the other wrappers, the schema of data returned by the wrapper may be specified (OUTPUTSCHEMA). In this case, the schema must contain a series of fixed attributes that must be present whenever the selected Denodo Aracne handler exports them. Only the name of these fixed attributes may be modified. Furthermore, the schema may also include specific attributes corresponding to other specific fields exported by the Aracne handler.

Below is a description of the fixed attributes (see [16] for further details):

- **TASK.** Name of the Aracne task that obtained and indexed this document. This is of the character string type.
- **PUBDATE.** Document publication date. This only appears in the event of the index containing RSS-type documents. This is of the character string-type.
- **TITLE.** Title generated by Aracne for the document. This is of the character string type.
- **ANCHORTEXT.** Where a document obtained by Aracne using a Web crawling process contains the link text used to access the document. This is of the character string-type.
- **SUMMARY.** Summary generated by Aracne for the document. This is of the character string type.
- **URL.** In the case of documents obtained over the Web page, this contains the original document URL. This corresponds to the `link` field value of the RSS item in RSS documents. 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 the character string type.
- **IDENTIFIER.** Standardized URL. This is of the character string-type.
- **CONTENT.** “Useful” contents of the document generated by Aracne. See the Aracne Administrator Guide [16] for further details. This is of the character string type.
- **DESCRIPTION.** This only appears in the event of the index containing RSS-type documents. In this case, it takes the value of the DESCRIPTION element from the RSS document. This is of the character string type.
- **MODIFIED.** Date on which the document in the index was last modified.
- **SEARCHABLECONTENT.** Field added by DataPort that saves the concatenation of the contents of the main fixed textual fields of the index (title, summary, contents, anchortext, etc.) and the specific 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 the character 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 the character string type.
• SUMMARYXML. Summary of the document in XML with information on the view structure of the contents (paragraphs). This field is used to visually represent the summary and not for searches. This is of the character string type.
• PATH. Where the Aracne server saves a local copy to the document, this contains the path to it. This is of the character 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 the floating type.
• MAXDOCS. Attribute added by DataPort to restrict the maximum number of results returned by a search. This is of the integer type.
• CATEGORIES. This only appears in the event of the index containing 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 the character string type.

Denodo Aracne is also capable of automatically generating the most relevant words of a document or a field thereof, 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 also related to this function. See section 17.4.9.1.

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 85.

```
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> ::= 
    MAINTERMS ( <name:identifier>,<num_of_mainterms_integer> [, { ( <literal> [, <literal>]* ) } ] )
```

Figure 85 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, wherever the selected Denodo Aracne handler also includes them. 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 17.4.9.1).
CREATE WRAPPER ARN aracneview3
   DATASOURCENAME=aracnesearch
   HANDLERNANE='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 86 Example of creating a Denodo Aracne wrapper

The syntax of the wrapper modification statement is similar and is shown in Figure 87.
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 87 Modification syntax of a Denodo Aracne wrapper

17.4.9.1 Adding fields with the most relevant terms

Denodo Aracne is capable of automatically generating the most relevant words of a document or a field thereof, 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 86 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 an array-type compound record attribute (see section 18.1). 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 the 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 85). 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.

17.4.10  Google Mini Wrapper

Virtual DataPort supports the creation of wrappers on search engines created using the Google Mini tool [17].
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 Mini server, the
collections on which to make the search. The collections are created by the Google Mini server administrator. Its
name is upper/lower 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 Mini 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 Mini documentation [18].
- **NUMKEYMATCH**: This parameter is optional. Google Mini allows for the administrator to manually determine
the priority of the pages, when the results of a search are displayed. 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 with the priority specified or over.

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 generated by Google Mini for the document. This is of the character string type.
- **SUMMARY**: Summary generated by Google Mini for the document. This is of the character string type.
- **URL**: Document URL. This is of the character string type.
- **MIMETYPE**: MIME type of the document. This is of the character string type.
- **RATING**: Priority assigned manually by the Google Mini administrator for the document. This may take values of
between 0 and 5, where 5 is the maximum priority. This is of the integer type.
- **MAXDOCS**: Field added by DataPort to restrict the maximum number of results returned by a search. This is of
the integer type.
- **METAS**: Array-type compound record attribute (see section 18.1) that contains the metatags for the document.
Each record has two character 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 the character string
type.
- **SITE**: This allows for the documents returned to be restricted to those belonging to a certain domain (e.g.
‘acme.com’). This is of the character string type.
- **FILETYPE**: Extension of the document file. This is of the character 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 88.
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:register field>  
<register field> ::=  
<name:identifier> = <mapping:literal> :  
REGISTER OF ( [ <field> [, <field>]* ] )

Figure 88  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 89  Example of creating a Google Mini wrapper

The syntax of the wrapper modification statement is similar and is shown in Figure 89.
17.4.11 CUSTOM Wrapper

MY-type 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 source (see section 17.3.9), 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.

To create a new CUSTOM-type wrapper two Java classes should be extended:

- The abstract class com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl. By extending this class the output schema of the new wrapper is defined together with certain additional metadata.

- com.denodo.vdb.engine.wrapper.raw.my.MyAccessImpl. This class is extended to implement the specific behavior of the wrapper.

The following subsections deal with each of these respectively. Then indications are given on how to register the new CUSTOM wrapper in the DataPort server, once it has been implemented.

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.

17.4.11.1 Defining the metadata of the CUSTOM wrapper

To define the metadata of the new CUSTOM wrapper the abstract class com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl must be extended. Specifically, the following methods must be redefined (see the Javadoc documentation [4] and the examples for more details):

```sql
ALTER WRAPPER GS <name:identifier>
  DATASOURCENAME=<name:identifier>
  SITECOLLECTIONS ( <literal> [, <literal>]* )
  [ CLIENT=<literal> ]
  [ LANGUAGES ( <literal> [, <literal>]* ) ]
  [ NUMKEYMATCH=<integer> ]
  [ OUTPUTSCHEMA ( <field> [, <field>]* ) ]
ALTER WRAPPER GS <name:identifier>
  <field> ::=<name:identifier> = <mapping:literal> [ VALUE <literal> ] :
  <type:literal>
  [ ( { OBL | OPT } ) ]
  | <name:identifier> = <mapping:literal> : ARRAY OF ( <register field> )
  | <name:register field>
  <register field> ::=<name:identifier> = <mapping:literal> :
  REGISTER OF ( [ <field> [, <field>]* ] )
```

Figure 90 Modification syntax of a Google Mini wrapper
• public abstract MyAccessImpl doCreate() throws CreateWrapperException. This method is responsible for creating the wrapper class that will implement execution of the query. Said class will be dealt with in the next section.

• public com.denodo.vdb.catalog.wrapper.metadata.MetaRegisterRaw getOutputSchema() throws LoadWrapperException. This method should return the schema of the data which will be obtained through the queries made by the wrapper. For each of the attributes contained in the response tuples the following should be indicated:
  
  o The data type.

  o 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. On creating a configuration parameter, its name and whether the parameter is mandatory (true) or optional (false) must be specified in the constructor. Where this method is not implemented, the wrapper will have no configuration parameters.

• public com.denodo.vdb.catalog.wrapper.SourceConfiguration getSourceConfiguration(). (OPTIONAL) This method allows for the configuration properties of the CUSTOM data source to be specified (see section 17.3.10). The implementation of this method on the wrapper may call up the implementation of this method in the superclass to obtain the default configuration properties. Where this method is not implemented, the wrapper will use the default configuration properties.

To simplify the process 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 [4]).

17.4.11.2 Creating the wrapper

Once the class that encapsulates the wrapper metadata has been defined, the class that will define the specific behavior of same must be created.

This class will extend com.denodo.vdb.engine.wrapper.raw.my.MyAccessImpl and, as was mentioned in the preceding section, it will be returned by the method doCreate of com.denodo.vdb.catalog.wrapper.my.MetaMyWrapperImpl (see the Javadoc documentation [4] for more details).

The following methods can be redefined:

• doRun. (mandatory) This method will be invoked by DataPort to execute a query on the wrapper. The query condition list that DataPort delegates to the wrapper for execution on the source is assigned as a
parameter. This list is comprised of objects of the type com.denodo.vdb.engine.wrapper.condition.WrapperCondition (see Javadoc documentation [4]).

- **doInsert.** In the event of the wrapper supporting inserts, this method will be called up by DataPort to execute an **INSERT** statement. It receives a list of attribute names and a list of values to be inserted as parameter.

- **doUpdate.** In the event of the wrapper supporting updates, this method will be called up by DataPort to execute an **UPDATE** statement. It receives a list of the attributes to alter, a list with the new values and a list of query conditions made up of com.denodo.vdb.engine.wrapper.condition.WrapperCondition objects as parameters (see Javadoc documentation [4]).

- **doDelete.** In the event of the wrapper supporting deletions, this method will be called up by DataPort to execute a **DELETE** statement on the wrapper. It receives a list of query conditions made up of com.denodo.vdb.engine.wrapper.condition.WrapperCondition objects as a parameter (see Javadoc documentation [4]).

- **prepare.** In the event of the wrapper supporting transactions, this method will be called up to prepare a transaction.

- **commit.** In the event of the wrapper supporting transactions, this method will be called up to confirm a transaction.

- **rollback.** In the event of the wrapper supporting transactions, this method will be called up to undo the changes to a transaction.

- **stop.** (mandatory) This method will be called up by DataPort to stop the execution of a wrapper.

The implementation of these methods may access the value of the wrapper configuration parameters through the getParameters() Map method. For each parameter, there is a key in the map in the form of MetaPayRollWrapper.PNAME, where PNAME is the name of the parameter.

Execution of the wrapper should provide the results in accordance with the interface com.denodo.vdb.engine.IRawResult (see Javadoc documentation [4]).

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 in the MyAccessImpl object.
It is important to bear in mind that the results returned by the wrapper should be compatible with the schema of the base relation with which said wrapper is associated in the Virtual DataPort server.

17.4.11.3 Creating or modifying the CUSTOM wrapper in the DataPort server

Figure 91 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 17.3.9).

Where the data source wrappers accept configuration parameters, the PARAMETERS clause allows for them to be specified.

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] that DataPort will take into account to determine the operations that can be made on it. The applicable properties are indicated in the corresponding statement declaration (Figure 91) and are explained in section 17.4.12.

```
CREATE [ OR REPLACE ] WRAPPER CUSTOM <name:identifier>
  DATASOURCENAME=<name:identifier>
  [ PARAMETERS (  <paramName:identifier>=<paramValue:literal>
                   [,<paramName:identifier>=<paramValue:literal>]* ) ]
```

**Figure 91** Syntax of the CUSTOM wrapper creation statement

Figure 92 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.

```
CREATE WRAPPER CUSTOM testcustom
  DATASOURCENAME=testcustomds
  PARAMETERS (  ENTERPRISE='enterprise1', YEAR='2006'
                  ) ;
```

**Figure 92** Example of creating a CUSTOM

The modification statement syntax of a CUSTOM wrapper is that shown in Figure 93. 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 93** Syntax of the CUSTOM wrapper update

17.4.12 Wrapper Configuration Properties

As indicated in section 17.3.10, Virtual DataPort keeps properties for each data source and wrapper, which allow for specific characteristics of the underlying sources to be configured such as their distributed transaction support capacity or whether inserting operations are permitted. Section 17.3.10 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.

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, if the operation is to be carried out graphically (see VDP Administrator Guide [3] 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 row 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 row 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 "true" by default.

- **Delegate AND Condition (DELEGATEANDCONDITION)**: This indicates whether the source allows for the AND condition to be delegated. The value is "true" by default for base views from 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 an XA [14] 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, whether sorting is ascending [ASC] or descending [DESC] must be specified for each field. 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 created using the query made by the user, delegating part of the processing to the native source. While VDP carries out this action automatically on relational databases (leaving selection, projection, union or join operations to be executed by the database that the specific base view comes from), other source types do not provide this information in their metadata, despite this sometimes being possible. VDP allows for the list of operators that can be delegated to be indicated in the Web Service (all by default) and CUSTOM ("=" by default) data types.

**Example:** If it is to be specified that a CUSTOM wrapper does not accept transactions (i.e. it does not allow for insert, update and delete operations or support distributed transactions), the wrapper creation statement should be as follows:

```sql
CREATE WRAPPER MY_role
    CLASSNAME = 'com.denodo.vdp.demo.wrapper.my.MetaPayRollWrapper'
    SOURCECONFIGURATION {
        ALLOWINSERT = false, 
        ALLOWDELETE = false, 
        ALLOWUPDATE = false, 
        SUPPORTSDISTRIBUTEDTRANSACTIONS=false
    };
```

**Figure 94** CUSTOM Wrapper configuration example

### 17.5 WRAPPER QUERY STATEMENTS

Virtual DataPort allows queries to be made directly to wrappers (without having to define base relations on them or search methods).

The general syntax of the statement to execute queries on wrappers is shown in Figure 95. The type and name of the wrapper must be indicated and an optional list of conditions in the format `<value>, binary operator, <value>` (see general syntax of condition values in section 4.8.1). Unary operators and multivalued binary operators are not allowed. This is a simplified version of the query aimed at carrying out tests on the wrappers.
The query statement syntax of an ITPilot wrapper is slightly different and is shown in Figure 96. Only a list of key=value pairs can be indicated separated by commas, which will be directly received by the wrapper execution context.

```
QUERY WRAPPER ITP <name:identifier>
[
  {<name:identifier> = <value:literal>
      [, <name:identifier> = <value:literal>]"
  }
]
```

**Figure 96** Syntax of the ITPilot QUERY WRAPPER statements
18 ADVANCED CHARACTERISTICS

This section describes some advanced characteristics of Virtual DataPort which, although not always necessary in the most common administration tasks, can be of interest in certain cases.

18.1 MANAGEMENT OF COMPOUND-TYPE VALUES

Virtual DataPort has two classes of types: simple types and compound types. Compound types (array and register) allow to seamlessly represent hierarchical data in the DataPort base relations and views. Therefore, a mechanism must be defined that enables navigation through the internal structure of a compound value to access its subelements (simple or compound) in any level of the tree.

**NOTE:** In Virtual DataPort, an array-type element must be viewed as a subrelation. In truth, 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.

In Virtual DataPort values are always contained in views (also called relations). A relation has a schema, i.e. a list of attributes, each one belonging to a data type.

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 can belong to a different data type). Each arch is tagged with the name of the subelement.

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 only the index value, between brackets, will be indicated) the names of the different arches which have to be moved through before 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 none of the output arches are specified (through the corresponding index), then the URI indicates the list of values obtained passing through all the arches of the array that was not indexed.

Therefore, from the point of view of evaluating URIs on tuples two types can be distinguished:

- Those that indicate a simple type or register-type value.
Those that indicate a list of values, the consequence of not having indexed an array-type element found in the structure of the type tree. A URI of this type indicates a series of nodes that are found in the same level of the tree. 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 6.3.1) to build compound values and use them in the right side of a condition. In this case, the operators ‘=’ y ‘<>’ 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 6.1.2.
- Aggregation functions (see section 6.4.1) support this type of URIs. For example, the LIST aggregation function allows this type of URIs as a parameter and returns as a result an array of registers that have each of the values referenced by this URI as a sole subelement.

In addition to the trees that represent values, other trees exist to represent the internal structure of the compound data types to which compound values belong. In this case, the array-type nodes have just one subelement (the register node which represents its associated register), as what is to be represented is the internal structure of the type and not an instance with values.

18.1.1 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 97. The data type to represent elements of the type AUTHOR can be created with the following statements:
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 97  Trees of compound elements

Figure 98 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>NAME</td>
</tr>
<tr>
<td></td>
<td>SURNAME</td>
</tr>
<tr>
<td></td>
<td>CONTACTADDRESS</td>
</tr>
</tbody>
</table>
The structure of the value tree is shown in Figure 99.

**Figure 98** Tuple with compound elements
Now a base relation that models this relation can be created:

```sql
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.

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:
Once the wrapper has been created, a search method can be defined for the BOOK relation. 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):

```
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 102  Adding a search method with compound types

NOTE: In the specification 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 (optionally <table name>.<attribute name>) will be specified between parenthesis, indicating the fields of the compound fields afterwards (separated by dots, or indexed with an integer value between brackets).

Finally, some examples of queries that could be made on the relation are shown:

1. For all the books that contain in their title the word ‘java’ the title is obtained and the name of each of the authors.
SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
FROM BOOK
WHERE TITLE like '%java%'
GROUP BY TITLE;

2. For all the books that contain in their title the word 'java' the title and the list of contacts for each of the authors is obtained.

SELECT TITLE, LIST((AUTHOR).CONTACTADDRESS) AS AUTHORLIST
FROM BOOK
WHERE TITLE like '%java%'
GROUP BY TITLE;

3. For all the books that contain in their title the word 'java' the title and the first mail address of each of the authors is obtained.

SELECT TITLE, LIST((AUTHOR).CONTACTADDRESS[0].MAIL) AS AUTHORLIST
FROM BOOK
WHERE TITLE like '%java%'
GROUP BY TITLE;

4. For all the books that contain the word 'java' in their title and that have at least one author with a mail address that contains the word '.es', the title and the name of each of the authors is obtained.

SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
FROM BOOK
WHERE (TITLE like '%java%')
AND ((AUTHOR).CONTACTADDRESS.MAIL like '%.es%')
GROUP BY TITLE;

5. For 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', the title and the name of each of the authors is obtained.

SELECT TITLE, LIST((AUTHOR).NAME) AS AUTHORLIST
FROM BOOK
WHERE (TITLE like '%java%')
AND ((AUTHOR).CONTACTADDRESS.ADDRESS.STREET like '%Real%')
GROUP BY TITLE;

6. 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) will be obtained.

SELECT TITLE, AUTHOR
FROM BOOK
WHERE (AUTHOR).CONTACTADDRESS =
{ROW('john@mail.com',{ROW('Real','Madrid','Spain'))}

18.2 OPTIMIZING QUERIES

This section describes different aspects of interest in relation to the optimizing of queries made through 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, how to configure the DataPort swapping to disk policy, while queries involving large intermediate results are being executed, is described.
18.2.1 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 is the one with the smallest expected size.

- **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 parallel to each other, whereas they are issued sequentially in the case of NESTED. It accepts an additional parameter that specifies the maximum number of subqueries issued parallel to each other.

- **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 and 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)
```
18.2.1.1 Dynamic choice of join strategy

When a query is executed that uses derived views in its FROM clause 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). This is only possible when the join operation was created without choosing a specific join strategy.

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 7.1) 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 103.

```
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 103 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)
```

and

```
CREATE VIEW V5 AS
   SELECT A,B,C,D,E
   FROM V4 NESTED ORDERED JOIN V3 USING (C)
   WHERE A>a
```

Figure 104 shows the definition tree for view V5 (this tree can be easily obtained with the help of the Virtual DataPort graphic administration tool. See [3]). 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 set 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
CONTEXT (QUERYPLAN = V5:NESTED REVERSEORDER V4:HASH ANY)
```

As indicated above, the QUERYPLAN option is also available in the 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 V5, write:

```
ALTER VIEW V5 QUERYPLAN = (V5:NESTED REVERSEORDER V4:HASH ANY);
```
18.2.2 Using the Cache

The commands for modifying a base relation (ALTER TABLE. See section 5.1) and modifying a view (ALTER VIEW. See section 7.1) allow the cache system to be enabled (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 ALTER DATABASE command (see section 12.3.2) allows for the default configuration for base relations and the views of a certain database to be established.

Note that if this option is activated in a view, it can also be used to run periodic preloads 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 points out a correct hit in the cache only if an identical query to the current one has already been executed. This is the mode used, when the ON parameter is selected for the cache.

- More general query cache: (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 select * from view where (field1 = a) are in the cache and the system receives the query select * from view where (field1 = a and field2 = b), 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 field2 = b condition is not fulfilled. If this option is disabled, the system will only use the cache if the query received is the same as a certain previous query.

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 select * from view where (field1 = a) query, then the result of applying the post-processing condition (field2 = b) to the results of the query can be different to the result obtained executing directly on the source select * from view where (field1 = a and field2 = b).

In case caching is not desired in a base relation, the CACHE OFF option must be used. The cache data expiration timeout can also be modified by using the TIMETOLIVEINCACHE property (in seconds).

18.2.2.1 Effect of using DELEGATEUNNAMEDVIEWS with the Cache System

The CONTEXT clause allows modifying the execution context for a specific query (see section 6.8). Its options include the DELEGATEUNNAMEDVIEWS parameter, which has some implications in the cache system behaviour. These implications are described below.

The DELEGATEUNNAMEDVIEWS parameter can take the values “YES” or “NO”. If not value is specified, “YES” is assumed.

When the administrator creates views using the DataPort graphical administration tool, the system may create some additional intermediate views. DataPort assigns an internal name for these views instead of allowing the user to specify one. That is why we term these views as “unnamed”. For instance, let us assume a new view called U is created using the graphic administration tool. U is created as a union view of two views A and B. If the user removes an attribute coming from A during the process of graphically creating the new view, DataPort will transparently add an unnamed projection view over A in the tree of U.

Besides, when a query is executed against an existing view, DataPort can also create a temporal unnamed projection or aggregation view which is destroyed when the query ends. For instance, let us consider the following query:
SELECT ATTR1, ATTR2 FROM VIEW

In order to execute this query, DataPort will create a temporal unnamed projection over VIEW. This projection will be in charge of preserving ATTR1 and ATTR2 and discarding the remaining attributes (it is interesting to note that the execution of the query \texttt{SELECT * FROM VIEW} does not require of any additional view).

In addition, DataPort will always try to push as much processing as possible to the data sources to optimize query execution. In some cases, pushing the processing of the operations of the unnamed views to the source may cause the cache of a view to be ignored, even if the cache has been activated. In those cases, the user can decide to set the value of this property to ‘NO’.

Let us consider again the example of an unión \( U \) composed from the base views \( A \) and \( B \). Let us also assume that \( A \) corresponds with a table in a relational database and \( B \) corresponds with an operation provided by a Web Service. When \( U \) was graphically created, an attribute coming from \( A \) was removed from the output of \( U \). Therefore, DataPort introduced a unnamed projection view above \( A \) in the tree of \( U \). We will also suppose that the cache is activated for the view \( A \).

If we now execute queries on \( U \), the cache of \( A \) will never be used. The reason is the following: When the execution process reaches to the node corresponding with the unnamed projection in the tree of \( U \), DataPort directly pushes to the source database both the projection and the sub-query the execution plan specifies on \( A \). Therefore, the execution process will never reach the node corresponding with \( A \) and its cache will not be used.

On the contrary, if the DELEGATEUNNAMEDVIEWS parameter is set to ‘NO’, then DataPort will not push to the database the operations corresponding with unnamed projections. Therefore, in our example, the cache of \( A \) will be considered by the execution process.

18.2.3 Configuring Swapping Policies

DataPort may require the automatic execution of swapping to discard operations to avoid possible memory overflow errors, while queries involving the processing and combination of large volumes of data are being executed.

The commands for modifying a base relation (\texttt{ALTER TABLE}, see section 5.1), modifying a view (\texttt{ALTER VIEW}, see section 7.1) and for executing a query (\texttt{CONTEXT} clause from the \texttt{SELECT} command, see section 6.8) specify whether DataPort is allowed to swap intermediate results to disk using the \texttt{SWAP ON} or \texttt{SWAP OFF} option. The \texttt{ALTER DATABASE} command (see section 12.3.2) allows for the default configuration for base relations and the views of a certain database to be established.

DataPort will swap, when \texttt{SWAP ON} is chosen and where an intermediate result produced while the query or view was being executed exceeds a certain maximum size. This size may be indicated (in megabytes) using the \texttt{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 may be wise to disable swapping for a specific view or for a specific query in which no memory overflow is foreseen. It may also be wise to increase the \texttt{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 \texttt{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 \texttt{swapping} in a view:

\begin{verbatim}
ALTER VIEW V SWAP OFF;
\end{verbatim}
2) Enabling `swapping` in a view, establishing a `SWAPSIZE` of 100 Mb:

```sql
ALTER VIEW V SWAP ON SWAPSIZE 100;
```

3) Running a query and disabling `swapping`:

```sql
SELECT ... CONTEXT ('SWAP' = 'OFF')
```

4) Running a query with `swapping` enabled and a `SWAPSIZE` of 100 Mb:

```sql
SELECT ... CONTEXT ('SWAP' = 'ON', 'SWAPSIZE'='100' )
```

### 18.3 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 DataPort includes internationalization configurations already created 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 [1]. 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 [2]. 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. European/Madrid for Spain = GMT+01:00 =MET = CET).

- **Currency configuration**: Allows different properties to be configured for the `money`-type values.
  - `currencyDecimalPosition` – Number of decimals acknowledged by 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](http://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html) class in the API standard Java Developer Kit are used to analyze the currencies [see Javadoc](http://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html) documentation for more information).

• **Configuration of time-type data.**

• **timePattern** – Unit of time in which the values of this type are expressed in this location. The possible values are: SECOND, MINUTE, HOUR, DAY, WEEK, MONTH and YEAR.

• **Configuration of dates:** Configuration of data 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. Table 1 shows 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, please read [9], classes [java.text.DateFormat](http://docs.oracle.com/javase/7/docs/api/java/text/DateFormat.html) and/or [java.text.SimpleDateFormat](http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html).
In Table 7 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:

- **Text**: with 4 or more characters to use complete form; less than 4 characters to use the abbreviated form.

- **Number**: uses the minimum number of digits possible. The 0s are added to the left of the shortest numbers. 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.

In a date format the characters that are not found in the ranges ['a'..'z'] or ['A'..'Z'] are considered text in inverted commas, i.e. characters such as ':', ',', ' ', '#', and '@' appear in the resulting date, although they are not in inverted commas in the format text.

- **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:
CREATE MAP i18n i18n_es_euro {
  'language' = 'es'  
  'country' = 'ES_EURO'
  'timezone' = 'Europe/Madrid'
  'currencydecimalposition' = '2'
  'currencygroupseparator' = ''
  'currencysymbol' = ''
  'currency' = 'EURO'
  'timepattern' = 'DIA'
  'datepattern' = 'd-MMM-yyyy H\'h\' m\'m\'
  'moneypattern' = '###,###,###.## ¤'
  'doubledecimalposition' = '2'
  'doublegroupseparator' = ''
};

Figure 105 Internationalization configuration es_euro

18.3.1 Accessing and maintaining information about currency exchange rates

The exchange rate values used to convert currencies are obtained through a view predefined in the admin database of within Virtual DataPort called CURRENCY. The CURRENCY view has three fields for specifying the name of a country (COUNTRY), the exchange rate of its currency with respect to euro (CHANGE) and a description (DESCRIPTION). The three attributes are of text-type. The format for the country codes is defined by the ISO country codes [2].

DataPort will be able of make currency conversions between any two currencies present in the CURRENCY view.

To automatically feed the CURRENCY view with updated data, external sources can be imported into the system that provide these data. For example, the Web site of the European Central Bank [7] provides the exchange rates with respect to the euro of the main currencies of the world in HTML, XML and CSV formats. The OANDA on-line currency converter [8] also provides data in this respect. The default CURRENCY view contains the fixed exchange rates of various old European currencies in relation to the EURO.

18.4 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 datasource associated with a specific base relation depending on the queries made on this relation (see section 17).

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 character 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 what are known as interpolation strings.

An interpolation string is an expression related with the execution context variables, which generates a string of characters 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 'a' and 'A'). Variables with a name that includes any other character can be specified, despite this not being alphanumeric, including the name between the symbols "@ (" and ")".

**NOTE:** When any of the symbols '@', '\', '^', '{', '}' appear in the constant parts of the interpolation string, they must be escaped by the character '\'. Note that this means that, on specifying local file-type paths in Windows Operating Systems, the character '\' must be escaped as '\\'.

Take the following example into account in order to get an intuitive idea of how interpolation strings operate.

**Example:** Suppose you have a Web server that allows for access to certain reports from the departments of a particular company coded 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 for access to these reports. To do so you must create an XML-type datasource (see section 17.3.4) and an XML-type wrapper (see section 17.4.7). 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 18.1).

When creating the datasource for this base relation, the problem arises that the data file to be accessed depends on the department referred to by the query made. 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 ITPilot wrappers (see section 17.4.5).
19 APPENDICES

19.1 SYNTAX OF SEARCH EXPRESSIONS FOR THE CONTAINS OPERATOR

This section describes the syntax of search expressions for the DataPort contains operator.

19.1.1 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).

19.1.2 Term modifiers

The use of the following modifiers is accepted:

19.1.2.1 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 will be entered:

```
inform*
```

19.1.2.2 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
```

19.1.2.3 Proximity searches

Searches for terms among which there is a certain special 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:
19.1.2.4 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.

19.1.2.5 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).

19.1.3 Boolean operators

Boolean operators allow for terms to be combined using logic operators. The following Boolean operators are accepted: AND, OR, and NOT (Note: Boolean operators must be written in upper-case letters.).
19.1.4 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}

19.1.5 Escaping special characters
The list of special characters is:

( ) { } [ ] ^ " ~ * : \n
To escape these characters, use \ before the character.

19.2 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 19. 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.

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 17.3.10.1) that can be consulted using the DESC VIEW statement (see section 13).

At present, the data source types allowing for the use of the contains operator are the Aracne-type sources (see section 17.3.6), Google Mini (see section 17.3.7) and Custom (see section 17.3.9).

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 17.3.10.1 and section 13).

19.2.1 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.
- For this to work correctly, 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.
19.2.2 Google Mini

The following characteristics of the contains operator search language are not supported in Google Mini-type sources:

- Searches by exact phrase are not supported in the site attribute. They are, 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.
- For this to work correctly, 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).
REFERENCES

[9] Javadoc documentation of the Java Developer Kit 1.4.1 Standard API