SDMX Technical Working Group

VTL Task Force

**The VTL 2.0 Interpreter**

**and the Sandbox user interface**

Version 3.0

*July 2018*

Table of contents

[The VTL sandbox 4](#_Toc528148538)

[How to install and run the VTL sandbox 4](#_Toc528148539)

[Command-line parameters 4](#_Toc528148540)

[The VTL Interpreter 6](#_Toc528148541)

[Operators implemented 6](#_Toc528148542)

[Automatic propagation of the attribute values 7](#_Toc528148543)

[The data types time\_period, date and time 7](#_Toc528148544)

[Architecture 7](#_Toc528148545)

[How the data are stored 9](#_Toc528148546)

[How the metadata are stored 9](#_Toc528148547)

[How the programs are stored 9](#_Toc528148548)

[User management 9](#_Toc528148549)

[History and audit information 10](#_Toc528148550)

[Remote databases 10](#_Toc528148551)

[The metabase 10](#_Toc528148552)

[The user interface 11](#_Toc528148553)

[Run 12](#_Toc528148554)

[Find 12](#_Toc528148555)

[History 13](#_Toc528148556)

[Clipboard 13](#_Toc528148557)

[Info 13](#_Toc528148558)

[Numeric precision ( .00 ) 14](#_Toc528148559)

[Export all 14](#_Toc528148560)

[Label 14](#_Toc528148561)

[Wizard-ML 14](#_Toc528148562)

[Wizard DL 15](#_Toc528148563)

[Wizard Statements 15](#_Toc528148564)

[Assignments and control-flow statements 16](#_Toc528148565)

[case 16](#_Toc528148566)

[for-do 16](#_Toc528148567)

[load 17](#_Toc528148568)

[Non-persistent assignment 17](#_Toc528148569)

[Persistent assignment 17](#_Toc528148570)

[print 18](#_Toc528148571)

[range 18](#_Toc528148572)

[return 19](#_Toc528148573)

[sql 19](#_Toc528148574)

[throw 20](#_Toc528148575)

[try-catch 20](#_Toc528148576)

[Data definition statements 21](#_Toc528148577)

[alter 21](#_Toc528148578)

[copy 22](#_Toc528148579)

[define datapoint ruleset 22](#_Toc528148580)

[define dataset 22](#_Toc528148581)

[define dataset like 23](#_Toc528148582)

[define hierarchical ruleset 23](#_Toc528148583)

[define operator 24](#_Toc528148584)

[define function 24](#_Toc528148585)

[define synonym 25](#_Toc528148586)

[define valuedomain 25](#_Toc528148587)

[define valuedomain subset 26](#_Toc528148588)

[define view 27](#_Toc528148589)

[description 27](#_Toc528148590)

[drop 28](#_Toc528148591)

[grant 28](#_Toc528148592)

[purge recyclebin 29](#_Toc528148593)

[rename 29](#_Toc528148594)

[restore 29](#_Toc528148595)

[revoke 30](#_Toc528148596)

The VTL sandbox

The VTL Sandbox is made of two components, the Interpreter and the sandbox user interface.

The interpreter is a complete engine to execute VTL 2.0 programs.

The interpreter can be embedded in other applications and therefore can be used with different user interfaces. Due to its architectural choices, it is suitable for developing both interactive tools or validation and transformation services.

In addition to the VTL language, the interpreter is able to execute control-flow statements and additional Data Definition statements (see the next sections). This allows the user to implement a complete process or procedure, create and modify datasets and valuedomains, in summary to make all functionalities needed in statistical processes available in a single tool. There is a command-line option (see below) to disable all statements that are not defined in VTL 2.0: with this option, the interpreter accepts only the VTL 2.0 syntax.

The sandbox includes 3 wizards (for VTL expressions, Data Definition statements and control-flow statements) that help the user in writing the correct syntax. The wizard asks the user first to choose an operator of the language and then the operands (in turn, the operands can be expanded using other operators). This approach is really useful when the user does not remember the detailed syntax, in particular for the statements. On the other hand, this approach is less natural to write arithmetic expressions that combine several operators.

Please note that the term *sandbox* refers to the simple user interface that is offered by this tool.

How to install and run the VTL sandbox

What you need to run the sandbox:

1. a machine that is able to run a Java program (under Windows, Linux or Unix). The Java version required is version 7 or higher.
2. an access to an existing Oracle database: database URL, username and password. The username must be granted privileges to create and modify objects (SQL table, view, package, procedure, sequence). The database can be hosted on any server that is accessible from the machine on which you run the sandbox. The Oracle version required is v. 11.2 or higher.

Instructions to download and run the sandbox:

1. Download "Sandbox v. 1.0.zip" from <https://github.com/vtl-sdmx-task-force/sdmx-vtl/releases>
2. Copy VTL.cmd and VTL.jar to a local folder.
3. Change the parameters in the command line in VTL.cmd:

database URL of the Oracle database (e.g. dev.cc.cec.eu.int:1597/EBASESTA\_TAF.cc.cec.eu.int)

username Oracle username

password password of the Oracle username

1. Double click VTL.cmd. If the sandbox window opens then it means that the sandbox has successfully created the system tables. To test the sandbox, type an expression like 1+2 in the top window and press Run. Close the tool (press X at the top right corner).
2. Delete -createmetabase from the command line in VTL.cmd
3. Double click VTL.cmd to connect again.

Command-line parameters

The following command-line parameters are implemented:

databaseUrl username password { **-createmetabase** | **-readonly** | **-vtl2.0** | **-x** commandFile }

Command-line parameters:

**-createmetabase** create the VTL system tables (VTL dedicated Oracle tables, views, etc. where the sandbox stores the metadata) in the username schema. This option must be executed only once.

**-readonly** connect to the database in read-only mode: statements that change the data or the metadata are not allowed.

**-vtl2.0** the interpreter and in the wizards accept only the VTL 2.0 syntax. All the additional statements that are not part of VTL 2.0 are disabled.

**-x** commandFile execute the statements contained in the file commandFile

The VTL Interpreter

The interpreter implements all operators defined in the VTL 2.0 Reference Manual, plus additional statements as explained in the section “Operators implemented”.

The input of the interpreter is a list of statements and expressions separated by semicolon, e.g. the following is a valid input:

ds1 := ds2 + 1 ;

ds1 \* 100

the interpreter executes the statements (or expressions) in the list and, if the last element in the list is an expression, returns an SQL query to the user interface (or to any other caller program). The user interface then executes the SQL query and shows the result to the user (or print the data to a file).

Operators implemented

The interpreter implements all operators defined in the VTL 2.0 Reference Manual.

In addition to the VTL operators, the interpreter provides control-flow statements that can be used to implement decision points, loops, exception handling, data loading etc. The control-flow statements are not part of VTL 2.0, they are in a sense another language that is provided for convenience together with VTL but it has its own syntax and semantics, although harmonised when possible with the VTL principles.

In addition to the VTL Data Definition statements, the tool offers Data Definition statements that can be used for creating and modifying datasets, valuedomains, views, user-defined functions and synonyms. These statements are not part of VTL 2.0 (note that some of them were in version 1.1) and are provided to offer a complete set of operations for administrating the VTL database.

The command-line parameter –vtl2.0 (see the command-line options) can be used to tell the interpreter (and the wizards) to disable all additional statements.

In summary, the language implemented is composed of 3 main types of operators and statements as described below.

*The VTL operators*

The interpreter implements all operators defined in VTL 2.0 (see [SDMX VTL page](https://sdmx.org/?page_id=5096)) with a few limitations for the following operators:

* join operator: only identifier components can be used to join datasets (see the using clause)
* check\_hierarchy: only the modes non\_null and partial\_null are implemented, and only output=dataset is implemented
* hierarchy: only the modes non\_null and partial\_null are implemented, and only output=computed is implemented
* cast: not all cast operations are implemented, in particular this operator never raises exceptions when applied to a dataset (e.g. a cast from number to integer carries out the conversion to integer but never fails)
* dataset type: the multiplicity modifier \_+ and \_\* are recognised by the parser but their behaviour is not implemented

If you try to execute options that are not implemented then you receive a message like "Internal error, option not yet implemented".

*Control-flow statements*

In addition to the VTL non-persistent assignment and persistent assignment, the interpreter has additional control flow statements for implementing a process, e.g. for managing a decision point in the process, for repeating a sub-process, for managing and throwing run-time exceptions, ad for loading data from external files.

*Data Definition statements*

In addition to the VTL statements **define datapoint ruleset**, **define hierarchical ruleset** and **define operator**, the interpreter has additional Data Definitionstatements for creating, modifying and dropping persistent objects and for granting/revoking read/update privileges on them.

Automatic propagation of the attribute values

The automatic propagation of the value of the attributes declared with the role viral attribute (see the VTL Reference Manual) is implemented in the unary and binary operators, in the aggregate and analytic operators. For example, if a dataset has a component with role viral attribute and you aggregate the data in the dataset with an aggregate operator, then the system applies the propagation method to compute a value for that attribute.

The propagation method is implemented with a predefined PL/SQL function mdt\_merge\_flags that has two string parameters. This method concatenates two strings, eliminates the duplicated characters and orders the characters in alphabetical order.

For example:

eval ( mdt\_merge\_flags ( "E", "PR" ) language "PL/SQL" returns string )

returns the string “EPR”. The returned characters are ordered so that they can be compared using a simple string equality operation. For example, assuming that ds has a viral attribute, the following expression:

sum ( ds group by ref\_area )

returns a dataset having the identifier component ref\_area, the measures of ds and the viral attribute.

The behaviour of the propagation method can be customised by replacing the PL/SQL function mdt\_merge\_flags with another function (with the same name) having two CHAR parameters and returning a CHAR value.

The data types time\_period, date and time

The valid formats for vales of type time\_period are reported below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Periodicity of the data* | *Format* | *Period* | *The load statement accepts also* | *Period indicator* |
| Date | yyyy-mm-ddThh:mm:ss | mm between 01 and 12  dd between 01 and 31 | yyyymmddhhmmss | T |
| Daily | yyyy-mm-dd | mm between 01 and 12  dd between 01 and 31 | yyyymmdd | D |
| Weekly | YYYYWww | ww between 01 and 52 |  | W |
| Monthly | yyyyMmm | mm between 01 and 12 |  | M |
| Quarterly | yyyyQq | q between 1 and 4 | qq between 01 and 04 | Q |
| Semesterly | yyyySs | s between 1 and 2 | ss between 01 and 02 | S |
| Annual | yyyy |  | yyyyA | A |

For date values the systems accepts the formats described above in the Date row.

The type time is a synonym for time period.

Architecture

A VTL program (list of statements or expression) is executed in two steps:

|  |  |  |
| --- | --- | --- |
| Step 1 | The parser reads the text of the VTL program and creates a syntax tree (or generates a syntax error) | |
| Step 2 | For a Manipulation Language statement or expression:  the interpreter executes the syntax tree (or generates a type error or run-time error) using the metadata stored in the VTL metabase. | For a Definition Language statement:  the interpreter stores the metadata into the VTL metabase and creates the necessary database datasets (or generates a type error) |

For example the picture below shows the abstract syntax tree created by the parser for the expression ds1 / ds2 \* 100:

The interpreter then retrieves the definition of ds1 and ds2 from the metabase and generates the following SQL statement (assuming that ds1 and ds2 have the identifier components ref\_area, partner and time, and the measure obs\_value):

SELECT a.ref\_area, a.partner, a.time, a.obs\_value / b.obs\_value \* 100

FROM ds1 a JOIN ds2 b ON a.ref\_area = b.ref\_area AND a.partner = b.partner AND a.time = b.time

The interpreter has been developed using the following products:

* Java 7 classes for developing the logic of the application
* JDBC for executing the SQL statement
* Oracle 11.2 for storing data and metadata. The interpreter uses when possible standard SQL syntax but in a few cases it uses Oracle proprietary extensions (e.g. a sequence to generate serial numbers) therefore, at least in the current version, it runs on Oracle only. It could be extended in the future to work with other systems (in particular, open source).

All data and metadata transformations are executed by running SQL statements. There are no in-memory data manipulations, and consequently no explicit caching of data (this is carried out by the DBMS). As a result, the interpreter provides good performance even for datasets with medium-high data volumes.

For a system in which the performance of the data retrieval is more important than the update operations (e.g. in a dissemination database) there is a specific option (not documented here) that tells the interpreter to use bitmap indexes instead of a primary key for indexing the data.

All data and metadata are stored in relational tables of the underlying database management system. No data or metadata are stored on files.

Due to its architectural choices, the interpreter is suitable for developing both interactive tools or validation and transformation services. It can be embedded in client applications or web applications.

Compared with the compiled approach (i.e. the VTL program is compiled into another language e.g. Java) the interpreter provides more flexibility, e.g. can be used for interactive tools in which the user can test on the fly different parameters of the validation rules for assessing the plausibility of the data.

The interpreter has no practical limitations in terms of data volume. It has a limitation due to the fact that the underlying DBMS imposes a maximum number of 1000 table columns (i.e. 1000 components in the same dataset). This limitation has an impact on the pivot operator (the number of values to be pivoted + number of identifier components cannot exceed 1000), the check\_hierarchy and the hierarchy operators which are implemented using a pivot operation (hopefully this will be rewritten in the future to avoid such limitations).

How the data are stored

When the user creates a dataset, the intepreter creates a relational table in the underlying database management system. For each component of the dataset, the relational table has a column whose type is the basic type of the component (if it is a basic type, otherwise the basic type of the valuedomain). In addition, a primary key is created on the components whose role is identifier.

When the interpreter creates a view, it creates a relational view in the underlying database management system. The view is created by translating the VTL expression defining the view into an SQL SELECT statement.

In addition, the system stores the structural metadata associated to a dataset or view as described below.

How the metadata are stored

The VTL metabase is a set of relational tables stored in the underlying database management system that contain the metadata relative to the VTL persistent objects. For example, the metabase tells the interpreter whether a dataset with a given name exists or not, which are the components etc.

The definition of the persistent objects created by the interpreter is stored in the VTL metabase. The definition of a dataset (or valuedomain) is stored in a table containing the description of the dataset components. If a component has a scalar constraint then the constraint is evaluated (at dataset creation time) to store a set of scalar values. Whenever the dataset is updated, the interpreter verifies that the constraint is satisfied, i.e. if the dataset component is:

identifier reporter ref\_area { “IT”, “FR” }

then the system verifies that no values other than “IT” and “FR” are stored in the reporter component of the dataset.

The metadata of the VTL persistent objects can be queried using the predefined datasets:

|  |  |
| --- | --- |
| Dataset | Contains |
| vtl\_all\_objects | The attributes (at VTL object level) of the persistent objects contained in the whole database |
| vtl\_all\_datasets | The structural definition (components and constraints) of the persistent datasets contained in the whole database |
| vtl\_user\_objects | The attributes (at VTL object level) of the persistent objects owned (i.e. created) by the current user |
| vtl\_user\_datasets | The structural definition (components and constraints) of the persistent objects owned (i.e. created) by the current user |

Those predefined datasets can be used, e.g. to know which are the datasets having a component with a given name or type, or which are the datasets having a component that includes a given code in its list of possible values.

All objects have properties attached (at object level) e.g. general description, date of creation, last modification, etc. Those properties can be configured in each database.

How the programs are stored

The objects containing VTL code (rulesets, user-defined operator, user-defined function) are stored in the underlying database management system by storing the syntax tree into a relational table (VTL system table). As a result, the objects are parsed only once and executed multiple times.

User management

It is a multi-user system. Each user has its own module (database schema) in which the data and metadata can be created. The owner of an object can grant/revoke read/update privileges to other users.

An object of which the current user is not owner can be denoted with the VTL notation:

owner **\** object

History and audit information

The system stores the following information:

* The statements executed in each session.
* When the data contained in a dataset are updated: the data relative to time of update, the min/max values of the time\_period identifier component affected, the number of cells updated etc.
* When a dataset is modified (using the alter statement) the type of change carried out.
* The start/end timestamp of the user sessions (login/logout).

Remote databases

It is possible to access datasets that are stored in other VTL databases, provided that an Oracle database link has been created in the current database and the remote database is a valid VTL database (with a metabase created). Under these assumptions then the remote dataset is available with the syntax ds@dblink. A remote dataset can be used within an expression as any other dataset e.g. ds1@dblink + ds2 is a valid expression. The database link must be created directly in SQL by executing a “CREATE DATABASE LINK” statement.

The metabase

The metabase is made of a few SQL tables containing:

* The names of the persistent objects and their object-level properties: description, date of creation etc. (mdt\_objects). A new object-level property can be created by adding a column to this table.
* the structural definition of datasets, valuedomains and views (mdt\_objects, mdt\_dimensions and mdt\_positions)
* the program code of user-defined functions and operators, datapoint rulesets and hierarchical rulesets (mdt\_syntax\_trees).

The different tables are linked using the column object\_id.

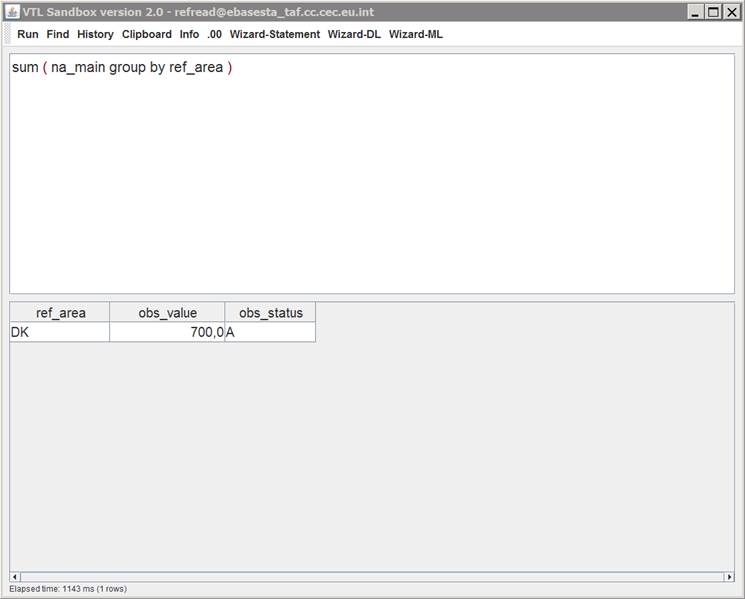
The user interface

This section describes the simple user interface of the sandbox.

The user interface is divided into 2 areas, the Input Area and the Data Window.

*The Input Area*

The Input Area is the upper part of the screen where the user enters the text to be executed.



Press CTRL-C and CTRL-V to copy/paste the selected text to/from the clipboard. Press CTRL-Z to undo changes. Press CTRL-A to select the whole text in the Input Area.

The keywords are displayed in red color, while the comments are displayed in grey color. The coloring is automatically applied when the user types some text in the Input Area. The keywords are colored if they are preceded and followed by a space.

*The Data Window*

The Data Window is the lower part of the screen where where the data are displayed.

The data presentation in the Data Window is flat, i.e. the window has a column for each component of the dataset resulting from the evaluation of the text in the Input Area.

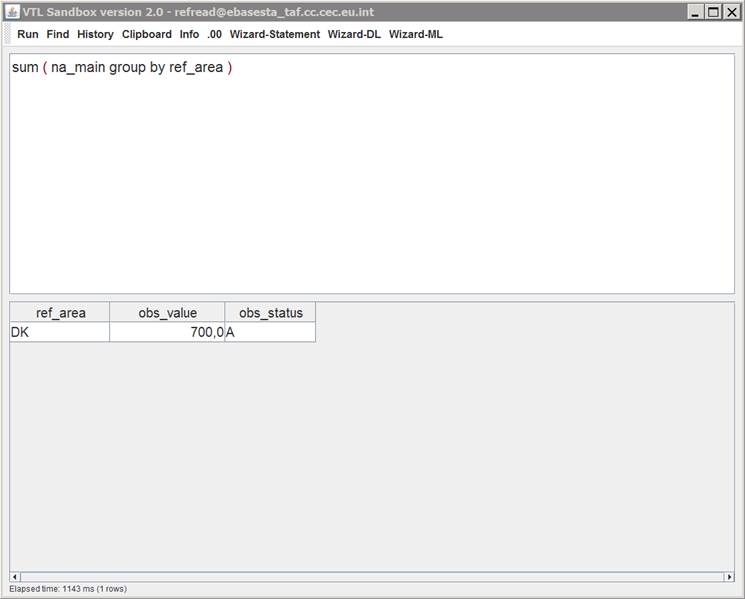
If you move the mouse on a cell in the Data Window then the system shows the role of the component and, if the component type is a valuedomain, then the system also shows the value of the measure (label\_en by default, but can be changed) associated to the code contained in the cell. In this way it is possible to have a description of the codes like e.g. description in several languages. The Label section below shows how to change the label name.

The following operations are available:

|  |  |
| --- | --- |
| **Button** | **Action** |
| Run | Execute the VTL code entered in the console and displays the resulting data in a table |
| Find | Show the syntax of an existing database object |
| History | Show the history of user commands |
| Clipboard | Copies the data contained in the Data Window to the clipboard |
| Info | Show the SQL query, the syntax tree and the data structure generated by the last execution |
| .00 | Change the number of decimal places displayed for numeric data |
| Export all | Export the definition of all VTL objects and a data file for each dataset or valuedomain |
| Label | Chose the label (description of a valuedomain item) |
| Wizard ML | Guide the user in writing the correct VTL syntax of expressions. When the user selects a dataset, the Wizard shows the list of existing datasets, the dimensions of the selected dataset, and the possible values of the selected dimension of the dataset |
| Wizard DL | Guide the user in writing the correct VTL syntax of the Definition Language statements |
| Wizard Statements | Guide the user in writing control-flow statements |

Run

Executes the code in the VTL Input Area. The resulting data (if any) are displayed in the Data Window.



Find

Finds an object in the database and shows the syntax needed to create the object.

Example for the dataset na\_main:

define dataset na\_main is

  identifier freq freq {"A"} ;

  identifier adjustment adjustment {"N"} ;

  identifier ref\_area ref\_area {"DK"} ;

  identifier counterpart\_area counterpart\_area {"W2"} ;

  identifier ref\_sector ref\_sector {"S1","S11","S12","S13","S14","S15","S1N"} ;

  identifier counterpart\_sector counterpart\_sector {"S1"} ;

  identifier accounting\_entry accounting\_entry {"B","C","D"} ;

  identifier sto sto {"B1G","B1GQ","D21","D21X31","D31","YA1"} ;

  identifier instr\_asset instr\_asset {"\_Z"} ;

  identifier activity activity {"A","BTE","C","F","GTI","J","K","L","M\_N","OTQ","RTU","\_T","\_Z"} ;

  identifier expenditure expenditure {"\_Z"} ;

  identifier unit\_measure unit\_measure {"XDC"} ;

  identifier prices prices {"L"} ;

  identifier transformation transformation {"N"} ;

  identifier ref\_year\_price integer {"2010"} ;

  identifier table\_identifier table\_identifier {"T0101"} ;

  identifier time time\_period {"2014","2015","2016","2017"} ;

  measure obs\_value number ;

  viral attribute obs\_status obs\_status ;

  attribute conf\_status conf\_status ;

  attribute embargo\_date date

end dataset ;

description of na\_main is [ object\_comment:="(New)" ] ;

History

Shows and allows to select the statements executed.

|  |  |
| --- | --- |
| Choose: | To show: |
| Current session | Statements executed in the current session |
| Past sessions | Statements executed in the previous sessions |
| Last statement | The last statement executed in the last previous section |

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| session\_id | time\_logon | history\_id | statement | elapsed\_time |
| 4349 | 2018-07-03 16:07:21.0 | 1 | sum ( na\_main group by ref\_area ) | 0 |

Clipboard

Copies the data contained in the Data Window to the clipboard. Rows are separated by the newline characters and columns are separated by the tab character.

Info

Shows information about the last statement executed:

SQL

The SQL statement sent to the underlying database management system.

Syntax tree

A human-readable representation of the abstract syntax tree relative to the last statement executed.

Data structure

The data structure of the result returned by the last statement

Example relative to the query described above:

SQL query:

SELECT a1."REF\_AREA" AS ref\_area,a1."OBS\_VALUE" AS "OBS\_VALUE",a1."OBS\_STATUS" AS "OBS\_STATUS" FROM (SELECT a0.ref\_area AS ref\_area,sum(a0.obs\_value) AS "OBS\_VALUE",substr (REFREAD.mdt\_merge\_flags('0',LISTAGG(a0.obs\_status) WITHIN GROUP (ORDER BY a0.obs\_status)),2) AS "OBS\_STATUS" FROM refread.na\_main a0 GROUP BY a0.ref\_area HAVING COUNT(\*) > 0)a1 ORDER BY 1

Syntax tree:

   print

      (aggregate) sum

         arguments

            (identifier) na\_main

         (group by)

            (identifier) ref\_area

         (empty)

      (empty)

      (empty)

Data structure:

identifier ref\_area ref\_area {DK}

measure obs\_value number

attribute obs\_status obs\_status

Referenced datasets:

refread\na\_main

Numeric precision ( .00 )

The numeric precision ( indicated as “.00” in the menu) allows to specify the number of decimal places displayed by the system.

Export all

Exports the definition of all objects owned by the current user to the specified file and, upon confirmation by the user, creates a data file for each dataset or valuedomain.

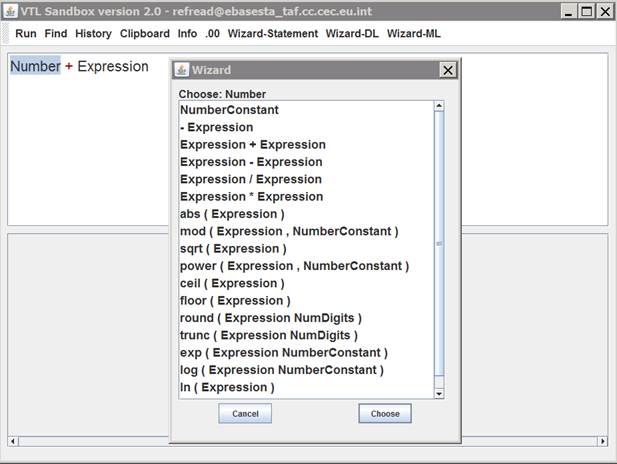
Label

The user chooses the name of the label that the usre interface will use to show information about a valuedomain item in the Data Window. The default is label\_en (description of a valuedomain item in English language). The user chooses the name of the label from the measures of the valuedomains currently available in the database. The label chosen is assumed to contain a textual description of the items of all valuedomains.

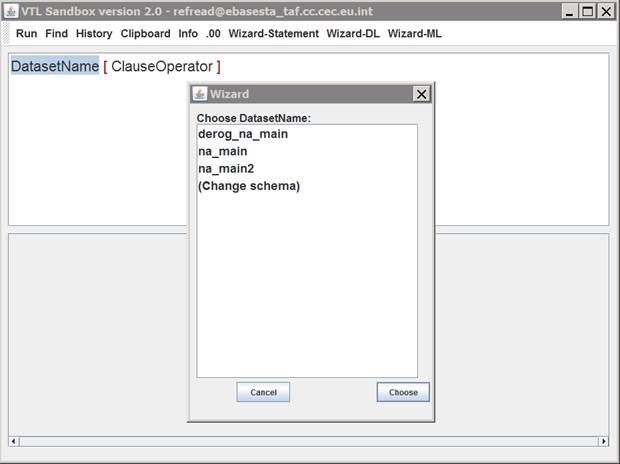
Wizard-ML

Guides the user in specifying a VTL expression as defined in the VTL 2.0 Reference Manual.

Starting with a generic expression, the user refines incrementally the operands of the expression. The example below shows the alternatives to expand a Number sub-expression.



When the user selects DatasetName, the Wizard shows the list of existing datasets, the identifier components of the selected dataset, and the possible values of the selected identifier component of the dataset.



Wizard DL

Guides the user in specifying a statement of the Definition Language.

Wizard Statements

Guides the user in specifying a control-flow statement.

Assignments and control-flow statements

This section describes the available statements. In addition to the VTL persistent and non-persistent assignments, the control-flow statements are available. The control-flow statements can be disabled using the –vtl2.0 command-line parameter

case

The **case** statement implements a decision point in the process.

*Syntax*

**case**

**when** booleanCondition **then** statementList **{ when** booleanCondition **then** statementList}\*

{ **else** statementList}

**end case**

statementList ::= statement **;** { statement **;** }\*

NB: the last semicolon can be omitted.

*Parameters*

booleanCondition a VTL Boolean condition

statement a statement

*Constraints*

None.

*Semantic specification*

This statement executes the list of statements associated to the first boolean condition (in the specified order) that evaluates to **true**.

*Examples*

case when A > 0 then proc1 ( ) when A = 0 then proc2 ( ) else proc3 ( ) end case

for-do

The **for-do** statement repeats a fragment of code for each value of a set of values.

*Syntax*

**for** tempVar **in** collection **do** statementList **end for**

collection ::= set | valueDomainName

statementList ::= statement **;** { statement **;** }\*

NB: the last semicolon can be omitted.

*Parameters*

tempVar name of the temporary variable to be used within statementList

scalarSet a set of scalar values, or a valuedomain name

*Constraints*

None.

*Semantic specification*

The list of statement is evaluated repeatedly for each value of scalarSet.

*Examples*

for x in { 1, 2, 3 } do proc1 ( x ) end for

load

The **load** statement loads a data file into a dataset.

*Syntax*

**load** fileName **into** dsName loadMode **separator** stringExpr

loadOption::= **replace** | **merge**

*Parameters*

fileName a string containing the file name (path) to which the data are written

loadOption either replace or merge (default is replace)

dsName the name of an existing dataset

stringExpr a string containing the separator character (tab, comma or semicolon)

*Constraints*

dsName is an existing dataset.

*Semantic specification*

This statement loads data from the specified filename to the specified dataset. In case of errors the error messages are printed on the screen and no data are loaded (the contents of the dataset is not changed).

*Examples*

load “myfile” into ds1

Non-persistent assignment

The non-persistent assignment creates a temporary dataset that can be used in other statements or expressions.

*Syntax*

tempVariable **:=** expression

*Constraints*

tempVariablecan be used in the left side of only one assignment. expression can use the temporary datasets created with previous assignments. There can be no cycles.

*Examples*

ds1 := ds2 + ds3

Persistent assignment

The persistent assignment replaces the contents of a persistent dataset with the data calculated by an expression.

*Syntax*

dsName { **[ sub** subspace **]** } { **[ filter** booleanCondition **]** } **<-** expression

subspace ::= identifier **=** value{ **,**identifier**=**value}\*

*Constraints*

* The two expressions at the left side and the right side of **<-** have the same identifier components. If subspace is specified then expression has the identifiers components of dsName that are not specified in subspace.
* The two expressions at the left side and the right side of **<-** have at least a common measure.

*Semantic specification*

The data points resulting from the evaluation of expression are first filtered with the existing ranges (if any) and then used to replace the data in dsName.

If subspace is specified then only the data specified by the subspace are replaced in dsName.

If booleanCondition is specified then only the data points of dsName for which the condition evaluates to **true** are replaced.

*Examples*

ds1 <- ds2 + ds3

ds1 [sub ref\_area = “BLEU” ] <- ds1 [sub ref\_area = “BE” ] + ds1 [sub ref\_area = “LU” ]

ds1 [filter obs\_value < 0] <- ds2 + ds3

ds1 [sub ref\_area = “BE” ] [filter obs\_value < 0] <- ds2 [sub ref\_area = “BE” ] + ds3 [sub ref\_area = “LU” ]

print

The **print** statement writes the data returned by an expression to a file.

*Syntax*

**print** ds { **order by** comp { , comp }\* } { **to** fileName }

*Parameters*

ds dataset expression

comp a component of the dataset

fileName a string containing the file name (path) to which the data are written

*Constraints*

* comp is a component of ds

*Semantic specification*

The data points contained in the dataset are printed to the file. The data points are optionally sorted by the components specified by the **order by** clause. The file is a local file, i.e. stored on the machine where the user interface runs.

Any ranges existing on the variables of *ds* are applied to filter the data to be printed.

If filename is not specified then the result is printed on the screen, but only if the **print** statement is the only statement to be executed, or the last statement in a statement list (otherwise the **print** statement has no effects).

*Examples*

print ds1 + ds2 order by cmp1, cmp2 to “mydir/myfile”

range

The **range** restricts the values of a variable to the specified range in the subsequent statements.

*Syntax*

**range** variablecollection

collection ::= set | valueDomainName

*Parameters*

variable name of an existing variable, i.e. the component of an existing dataset

collection a set of scalar values, or a valuedomain name

*Constraints*

* the basic scalar type of collection is compatible with the type of variable. Note that a variable has always the same type (basic scalar type or valuedomain) in the database.

*Semantic specification*

In the subsequent print and persistent assignment statements, the values of the variable are restricted to the values specified in collection, i.e. the filter “[filter variable in collection]” is added implicitly to those statements.

Every time that a user-defined function is called, it inherits the ranges specified in the calling function. However, the scope of a range statement appearing within the user-defined function do not affect the calling function (i.e. the effect of range ends at the end of the function in which the range is specified).

*Examples*

// the following statement is equivalent to: ds1 [ filter me1 > 0 ] [ filter ref\_area in { “IT”,”FR” } ]

range ref\_area { “IT”, “FR” } ; ds1 [ filter me1 > 0 ]

return

The **range** statement is used within a user-defined function to end the function and return a value.

*Syntax*

**return (** expression **)**

*Parameters*

expression a VTL expression

*Constraints*

* The return statement can appear only in a user-defined function
* The type of expression is compatible with the return type specified in the user-defined function

*Semantic specification*

This statement ends the execution of a user-defined function and returns the value of the specified expression.

*Examples*

define function p1 ( number x, number y ) is z := x + y ; return ( z ) end function

sql

The **sql** statement executes an SQL statement and prints the returned data on the screen.

*Syntax*

**sql** sqlQuery

*Parameters*

sqlQuery a string containing the SQL query to be executed

*Constraints*

sqlQuery is a string containing a valid SQL SELECT statement. It may contain tab characters and newlines.

*Semantic specification*

The sqlQuery is executed by the underlying database management system and the resulting data are printed on the screen, but only if the **sql** statement is the only statement to be executed, or the last statement in a statement list (otherwise the **sql** statement has no effects).

*Examples*

sql “SELECT \* FROM t WHERE me1 > 0”

throw

The **throw** statement throws an exception.

*Syntax*

**throw (** stringExpression **)**

*Parameters*

stringExpression a VTL string expression

*Constraints*

stringExpressionhas type string

*Semantic specification*

This statement throws an exception identified by the specified message.

*Examples*

throw ( “Parameter value should be positive or zero” )

try-catch

The **try-catch** statement is used to handle exceptions.

*Syntax*

**try** tryStatementList **catch** **(** parameterName **string )** catchStatementList **end try**

tryStatementList ::= statement **;** { statement **;** }\*

catchStatementList ::= statement **;** { statement **;** }\*

NB: the last semicolon can be omitted.

*Parameters*

parameterName name of the temporary variable to be used within catchStatementList

*Constraints*

None.

*Semantic specification*

The list of statement tryStatementList is executed. If any statement raises an exception then the catchStatementList is executed.

*Examples*

try proc1 () catch ( msg string ) proc2 ( msg ) end try

Data definition statements

This section describes the available Data Definition statements. In addition to the VTL statements **define datapoint ruleset**, **define hierarchical ruleset** and **define operator**, other non-VTL statements are available. If needed, the usage of the non-VTL statements can be disabled using the –vtl2.0 command-line parameter. The system can create and use the following persistent objects:

|  |  |
| --- | --- |
| Persistent object | Used to define |
| datapoint ruleset | a set of data point (horizontal) validation rules |
| dataset | a dataset (set of data with a given structure) |
| hierarchical ruleset | set of hierarchical (vertical) validation rules |
| user-defined function | a user-defined function (defined by a set of statements) |
| user-defined operator | a user-defined operator (defined by an expression) |
| synonym | a synonym for a persistent object |
| valuedomain | a valuedomain (set of scalar values) |
| view | a dataset whose data are dynamically retrieved by executing an expression |

alter

*Syntax*

**alter** dsName alterClause

alterClause ::=

**add** newComponentName scalarType { **,** newComponentName scalarType }\*

| **modify** componentName scalarType { **,** componentName scalarType }\*

| **drop** componentName

| **rename** componentName **to** newComponentName

| **move** componentName { **before** componentName }

| **storage** storageOptions

*Parameters*

dsName name of the dataset to be altered

componentName name of a component of the dataset

scalarType a scalar type as defined in the VTL Reference Manual

*Constraints*

* objectName is the name of an existing VTL object
* componentName is the name of a component of the dataset

*Semantic specification*

This operator modifies the structural definition of a dataset according to the parameters.

* **add** adds a new component to the dataset
* **drop** componentName removes the specified component. It raises an exception if componentName is an identifier component and takes more than one value in the dataset
* **rename** changes the name of a component of the dataset
* **move** moves the component before another component. If **before** is not specified then the component is moved to be the last component of the dataset. Please note that moving a component has no impact on the behaviour.
* **storage** changes the storage phisical parameters of the dataset. This parameter has no impact on the behaviour but can influence the performance of the operations.

*Examples*

alter dataset ds1 add measure me1 integer

alter dataset ds1 drop me1

alter dataset ds1 rename me1 to me2

alter dataset ds1 modify me1 integer

alter dataset ds1 move me1 before me2

copy

*Syntax*

**copy** objectName **to** newObjectName

*Parameters*

objectName name of the object to be copied

newObjectName name of the new object to be created

*Constraints*

* objectName is the name of an existing VTL object
* newObjectName is not used by any existing database objects

*Semantic specification*

This operator creates a new database object defined as the existing object. For a dataset or a valuedomain it copies also the data contained. For a valuedomain, it changes the name of the identifier component to the name of the new object.

*Examples*

copy ds1 to ds 2

copy op1 to op2

define datapoint ruleset

Please see the VTL 2.0 Reference Manual.

This statement creates a data point ruleset with the specified name or replaces an existing datapoint ruleset with the specified name. The keyword **create** can be specified instead of **define**: **create** raises an exception if an object named newObjectName already exists. **define** raises an exception if an object named newObjectName already exists and is not a datapoint ruleset.

define dataset

*Syntax*

{ **define** | **create** } **dataset** dsName **is**

componentName scalarType { dataLength } { **;** componentName scalarType { dataLength } }\*

**end dataset**

scalarType ::= { basicScalarType | valueDomainName } { scalarTypeConstraint } { { **not** } **null** }

basicScalarType ::= **number** | **integer** | **string** | **boolean** | **time** | **date** | **time\_period** | **duration**

scalarTypeConstraint ::= **[** valueBooleanCondition **]** | **{** scalarLiteral { **,** scalarLiteral }\* **}**

*Parameters*

dsName the name of the dataset to be created

scalarType a scalar type as defined in the VTL Reference Manual

dataLength for a component of type **string**, the maximum number of characters

*Constraints*

* dsName is not used by any persistent object
* the names of the components are unique within the dataset
* the type of a component is identical to the type of an existing component (of an existing dataset) with that name (if any). The role can be different.
* The dataLength specified for a string component must be <= 4000.

*Semantic specification*

This operator creates a dataset having the specified components.

The keywords **create** and **define** have the same semantics.

*Examples*

define dataset ds1 is

identifier id1 vd1,

measure me1 number,

attribute at1 (100)

end dataset

// A, B and C are valid values for valuedomain vd1

define dataset ds1 is

identifier id1 vd1 { “A”, “B”, “C” },

measure me1 number,

attribute at1 (100)

end dataset

define dataset like

*Syntax*

{ **define** | **create** } **dataset** dsName **like** existingDataset

**end dataset**

*Parameters*

dsName the name of the dataset to be created

existingDataset the name of a persistent dataset

*Constraints*

* dsName is not used by any persistent object
* existingDataset is the name a persistent dataset

*Semantic specification*

This operator creates a dataset having the components of existingDataset.

The keywords **create** and **define** have the same semantics.

*Examples*

define dataset ds1 like ds2

end dataset

define hierarchical ruleset

Please see the VTL 2.0 Reference Manual.

This statement creates a hierarchical ruleset with the specified name or replaces an existing hierarchical ruleset with the specified name. The keyword **create** can be specified instead of **define**: **create** raises an exception if an object named newObjectName already exists. **define** raises an exception if an object named newObjectName already exists and is not a hierarchical ruleset.

define operator

Please see the VTL 2.0 Reference Manual.

This statement creates a user-defined operator with the specified name or replaces an existing user-defined operator with the specified name. The keyword **create** can be specified instead of **define**: **create** raises an exception if an object named newObjectName already exists. **define** raises an exception if an object named newObjectName already exists and is not a user-defined operator.

define function

*Syntax*

{ **define** | **create** } **function** functionName **(** {parameter{ **,**parameter}\*}**)**

{ **returns** outputType}  
**is** statementList

**end function**

parameter ::= parameterName parameterType { **default** parameterDefaultValue }

statementList ::= statement { **;** statement}\*

NB: the last semicolon can be omitted.

*Parameters*

functionName the name of the user-defined function to be created

outputType a VTL data type as defined in paramResultType (see VTL Data Type Syntax)

operatorBody a VTL expression

parameterName the name of the parameter

parameterType a VTL data type as defined in paramResultType (see VTL Data Type Syntax)

parameterDefaultValue the default value for the parameter (optional)

returnType a VTL data type as defined in paramResultType (see VTL Data Type Syntax)

*Constraints*

* functionName is not the name of an existing persistent object (**create**) or is not the name of a persistent object that is not a user-defined function (**define**)
* parameterName must be unique within the list of parameter names
* parameterDefaultValue must be of the same data type as the corresponding parameter
* the type of operatorBody must be compatible with outputType (it can be a sub-type of outputType)
* If outputType is omitted then the type of operatorBody expression is assumed
* If parameterDefaultValue is specified then the parameter is optional in the function call (the symbol “\_” tells VTL to use parameterDefaultValue for the parameter). It can be provided only when the data type of the parameter is a scalar type.

*Semantic specification*

This statement creates a user-defined function with the specified name or replaces an existing user-defined operator with the specified name. A user-defined function is a list of statements and has parameters. The parameters and return type are defined as in **define operator**.

If returnType is not specified then the user-defined function does not return any value.

*Examples*

define function f1 ( ds1 { identifier <string> id1, measure <number> me1 }, ds2 { identifier <string> id1, measure <number> me1 } ) is

x := ds2 \* 100 ;

ds1 <= x ;

end function

define function f2 ( ds1 { identifier <string> id1, measure <number> me1 }, ds2 { identifier <string> id1, measure <number> me1 } ) returns boolean is

case when ( count ( ds2 ) > 0 ) then

x := ds2 \* 100 ;

ds1 <= x ;

return true ;

else

return false ;

end function

define synonym

*Syntax*

{ **define** | **create** } **synonym** newObjectName **for** objectName

NB: the last semicolon can be omitted.

*Parameters*

newObjectName the name of the synonym to be created

objectName the name of an existing object

*Constraints*

* newObjectName is not the name of a persistent object
* objectName cannot be a valuedomain

*Semantic specification*

This operator defines a synonym for a existing object.

The keyword **create** can be specified instead of **define**, with the same semantics.

*Examples*

define synonym syn1 for ds1

define valuedomain

*Syntax*

{ **define** | **create** } **valuedomain** newObjectName **is**

componentName scalarType { **(** dataLength **)** } { **;** componentName scalarType { **(** dataLength **)** } }\*

**end valuedomain**

scalarType ::= { basicScalarType | valueDomainName } { scalarTypeConstraint } { { **not** } **null** }

basicScalarType ::= **number** | **integer** | **string** | **boolean** | **time** | **date** | **time\_period** | **duration**

scalarTypeConstraint ::= **[** valueBooleanCondition **]** | **{** scalarLiteral { **,** scalarLiteral }\* **}**

*Parameters*

newObjectName the name of the valuedomain to be created

scalarType a scalar type as defined in the VTL Reference Manual

dataLength for a component of type string, the maximum length (number of characters allowed)

*Constraints*

* newObjectName is not the name of any persistent object
* only one identifier component is allowed and its name must be identical to newObjectName
* the names of the components are unique within the valuedomain
* The maximum length specified for a string component must be <= 4000.

*Semantic specification*

This operator creates a valuedomain having the specified components.

The scalarTypeConstraint is evaluated to create a list of scalar values that the legal values of the valuedomain.

From a structural point of view, a valuedomain has the structure of a dataset with the constraint that it has exactly one identifier component with the same name of the valuedomain. The other measures and attributes can store a textual description or other useful information about the values of the valuedomain, e.g. descriptions in multiple languages can be stored in different measures or attributes.

A valuedomain can be used in any context in which a dataset can be used: data can be loaded and updated using a persistent assignment or the load statement, can be printed etc.

The keywords **create** and **define** have the same semantics.

*Examples*

Assuming that we have the following file *adjustment.dat*, tab-separated:

adjustment label\_en

C Trend-cycle data, calendar adjusted

I Irregular component

K Calendar component

M Seasonal and calendar components

N Neither seasonally adjusted nor calendar adjusted data

R Trend-cycle data, not calendar adjusted

S Seasonally adjusted data, not calendar adjusted

T Trend

W Calendar adjusted data, not seasonally adjusted

X Seasonal component

Y Calendar and seasonally adjusted data

The following statements:

define valuedomain adjustment is

identifier adjustment string ( 10 ) {"C","I","K","M","N","R","S","T","W","X","Y"} ;

measure label\_en string ( 1000 )

end valuedomain ;

load adjustment.dat into adjustment separated by “ “ ;

print adjustment order by adjustment

produce the following result:

adjustment label\_en

C Trend-cycle data, calendar adjusted

I Irregular component

K Calendar component

M Seasonal and calendar components

N Neither seasonally adjusted nor calendar adjusted data

R Trend-cycle data, not calendar adjusted

S Seasonally adjusted data, not calendar adjusted

T Trend

W Calendar adjusted data, not seasonally adjusted

X Seasonal component

Y Calendar and seasonally adjusted data

Note that several languages can be managed by adding other measures to the valuedomain.

define valuedomain subset

*Syntax*

{ **define** | **create** } **valuedomain** vdName **subset of**

valueDomainName { scalarConstraint }

**end valuedomain**

scalarTypeConstraint ::= **[** valueBooleanCondition **]** | **{** scalarLiteral { **,** scalarLiteral }\* **}**

*Parameters*

vdName the name of the valuedomain to be created

valueDomainName the name of an existing valuedomain

scalarTypeConstraint a scalar costraint as defined in the VTL Reference Manual

dataLength for a component of type string, the maximum length

*Constraints*

* vdName is not the name of an existing persistent object (**create**) or is not the name of a persistent object that is not a valuedomain subset (**define**)
* only one identifier component is allowed and its name must be identical to newObjectName
* the names of the components are unique within the dataset
* the type of a component is identical to the type of an existing component (of an existing dataset) with that name (if any). The role can be different.
* The length specified for a string component must be <= 4000.

*Semantic specification*

This statement creates a valuedomain subset with the specified name or replaces an existing valuedomain subset with the specified name. This statement creates a valuedomain as a subset of an existing valuedomain. The constraint is used to create a list of valid values for newly created valuedomain. A valuedomain subset can be used in all data type specifications in which a valuedomain can be used.

*Examples*

define valuedomain subset vd2 subset of vd1 { “BE”, “LU”, “NL” } end valuedomain

define valuedomain subset vd2 subset of vd1 { value in { “BE”, “LU”, “NL” } } end valuedomain

define view

*Syntax*

{ **define** | **create** } **view** viewName **is**

expression

**end view**

*Parameters*

viewName the name of the view to be created

expression a VTL expression denoting a dataset or scalar

*Constraints*

viewName is not the name of a persistent object (**create**) or is not the name of a persistent object that is not a view (**define**).

*Semantic specification*

This statement creates a view with the specified name or replaces an existing view with the specified name. A view is a persistent dataset whose content is dynamically retrieved by executing the specified expression. The data retrieved may change (if the data re base data change)

*Examples*

define view v1 is ds1 + ds2

description

*Syntax*

**description of** objectName **is** **[** propName **=** propValue { propName **=** propValue }\* **]**

*Parameters*

objectName the name of a persistent object

propName the name of a property defined at object level

propValue the value assigned to property

*Constraints*

* objectName is the name of a persistent object
* the type of propValue must be compatible with the type of propName

*Semantic specification*

This operator changes the properties defined at object level (e.g. textual description of the object).

The properties are associated to all persistent objects, and the same for all objects.

The properties must be created in the underlying database management system by adding new columns to a specific metabase table. Once created their value can be set and changed using the description statement.

*Examples*

description of ds1 is [ title\_en = “My test 1”, title\_fr = “Mon test 1” ]

drop

*Syntax*

**drop** objectName { **purge** }

*Parameters*

objectName name of the object to be dropped

purge specifies that the object is permanently dropped

*Constraints*

* objectName is the name of a persistent VTL object
* objectName is not the name of a valuedomain that is used in a persistent dataset

*Semantic specification*

This operator drops an existing database object. If purge is specified then the object is dropped permanently, otherwise the object is moved to the recyclebin from which it can be restored.

*Examples*

drop ds1

drop ds1 purge

grant

*Syntax*

**grant** privilege **on** objectName **to** userRole

privilege = **read** | **update**

*Parameters*

objectName the name of a dataset, valuedomain or view

privilege either **read** or **update**

userRole the name of a user or role defined in the database

*Constraints*

* objectName is the name of an existing VTL object

*Semantic specification*

This operator grants the speficied privilege to the specified user or role. Note that role must be created directly in SQL.

*Examples*

grant read on ds1 to user1

purge recyclebin

*Syntax*

**purge recyclebin**

*Semantic specification*

This operator drops permanently all objects contained in the recyblebin, i.e. all objects previously dropped with no **purge** option.

*Examples*

purge recyclebin

rename

*Syntax*

**rename** objectName **to** newObjectName

*Parameters*

objectName the name of the object to be renamed

newObjectName the new name to be created

*Constraints*

* objectName is the name of an existing VTL object
* newObjectName is not used by any existing database objects

*Semantic specification*

This operator renames an existing database. For a valuedomain, it changes the name of the identifier component to the name of the new object.

*Examples*

rename ds1 to ds 2

rename op1 to op2

restore

*Syntax*

**restore** objectName { **rename to** newObjectName }

*Parameters*

objectName the name of the object to be restored

*Constraints*

* objectName is the name of a previously dropped object
* newObjectName is not used by any existing database objects

*Semantic specification*

This operator restores a previously dropped object. IfobjectName is the name of an existing object and newObjectName is not specified then an error is raised. Specify newObjectName to avoid this error.

*Examples*

restore ds2

restore ds2 rename to ds1

revoke

*Syntax*

**revoke** privilege **on** objectName **from** userRole

privilege = **read** | **update**

*Parameters*

objectName the name of a dataset, valuedomain or view

privilege either **read** or **update**

userRole the name of a user or role defined in the database

*Constraints*

* objectName is the name of an existing VTL object

*Semantic specification*

This operator revokes the specified privilege from the specified user or role. Note that role must be created directly in SQL.

*Examples*

revoke read on ds1 from user1