Denodo Testing Tool - User Manual

Revision 20200924

Denodo Inc.: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA www.denodo.com

CONTENTS

1 OVERVIEW....4

2 INSTALLATION....5

3 TEST FORMAT....5
3.1 BASIC FEATURES....5
3.2 RESOURCE RESOLUTION....7
3.3 ESCAPING SPECIAL CHARACTERS....7
3.4 EXTENDS DIRECTIVE....8
3.5 NAME DIRECTIVE....9
3.6 DESCRIPTION DIRECTIVE....9
3.7 CONTEXT DIRECTIVE....10
3.8 SETUP AND TEARDOWN DIRECTIVES....11
3.9 EXECUTION DIRECTIVE....12
3.10 RESULTS DIRECTIVE....16
3.11 TRACE DIRECTIVE....24

4 TESTING TOOL EXECUTION....27
4.2 CONFIGURATION FILE....28
4.3 TEST LOCATORS....33
4.4 JUNIT INTEGRATION....34

5 TEST RESULTS....35
5.1 TEST RESULTS: CONSOLE TEST REPORTER....35
5.2 TEST RESULTS: CSV TEST REPORTER....36
5.3 TEST RESULTS: EMAIL TEST REPORTER....36
5.4 TEST RESULTS: HTML TEST REPORTER....40

6 TEST REPORTERS....42
6.1 CONSOLE TEST REPORTER....42
6.2 CSV TEST REPORTER....43
6.3 EMAIL TEST REPORTER....43
6.4 HTML TEST REPORTER....43

7 DEBUGGING....43

8 HOW TO ENCRYPT PASSWORDS....44

9 TROUBLESHOOTING.....46

10 APPENDIX: BENCHMARKING.....47

1 OVERVIEW

The Denodo Testing Tool allows Denodo users to easily automate the testing of their data virtualization scenarios, both during the development of their virtualized solutions and during their maintenance, acting as a *safety net* before applying any significant changes to these environments.

The Denodo Testing Tool is a lightweight standalone application, executable from the command line and with a very simple user interface. Some quick facts:

- Tests are specified in text files and organized in folders. No Java programming needed.
- Test sets are executed in a completely automated manner. Test set executions can be integrated as a part of existing continuous integration (CI) processes.
- Test inheritance supported for easier creation of large sets of heavily related tests.
- Pluggable test result reporting system.

2 INSTALLATION

The Denodo Testing Tool distribution consists of:

- A set of command-line executable scripts for Windows and Linux (`/bin` folder)
- Example configuration files for both the tool and its logging system (`/conf` folder)
- A documentation folder containing this user manual (`/doc` folder)
- A series of JDBC driver jar binary files (`/drivers` folder)
- A series of Java jar binary files (`/lib` folder)
- A folder containing some sample test sets (`/samples` folder)

It does not require any special installation. Just download the zip file and extract the tool into the desired folder.

For running the Denodo Testing Tool you will need a **JRE** or **JDK version 7** or later, and a `PATH` environment variable correctly configured to run it from the console. You can check your Java installation by running the following command on a console.

```
$ java -version
```

If you get a version number, which needs to be **1.7 or later**, you are ready to start using the Denodo Testing Tool.

It is also recommended that you have a `JAVA_HOME` environment variable set to the installation folder of the Java Runtime Environment being used.

3 TEST FORMAT

3.1 BASIC FEATURES

A test is simply a plain text file with the `.denodotest` extension and a set of valued directives. The available directives are:

- `EXTENDS`
- `NAME`
- `DESCRIPTION`
- `CONTEXT`
- `SETUP and TEARDOWN`
- `EXECUTION`
- `RESULTS`
- `TRACE`

A very quick sample:

```
# --------------
# Lines starting with # are comments
# --------------
%EXTENDS ../jointest.denodotest
# --------------
%NAME JoinTest2
# --------------
%DESCRIPTION
This test verifies the join of customer tables from the db2 and extws
sources.
# --------------
%CONTEXT
username = jdoe
# --------------
%SETUP[script] {ds:vdp} file:../testscripts/join/add_all_users.vql
# --------------
%EXECUTION[query] {ds:vdp}
SELECT id, name, login, role FROM users_full
```

```
WHERE manager_login = '${username}'
# --------------
%RESULTS[data]
id,name,login,role
4,Jane Doe,jdoe,accounting
5,John Doe,jdoe,finance
# --------------
%TRACE
EXECUTION   PLAN.PROJECTION   PLAN.INNER   JOIN   PLAN.BASE   PLAN[0].JDBC
WRAPPER.numRows == 200
# --------------
%TEARDOWN[script] {ds:vdp} file:../testscripts/join/remove_all_users.vql
```

Note how, for each directive, we can differentiate four parts:

- The directive name, starting with %: **%NAME**, **%SETUP**, **%EXECUTION**
- The (optional) directive qualifier between brackets, which acts as a specialization of the directive: **[query]**, **[data]**, **[script]**
- The (optional) directive context between braces, required in some cases for specifying needed directive configuration: **{ds:vdp, ordered:no}**
- The directive value, which can be single- or multi-lined.

Variables are available and they use the notation `${some.variable}` like **${username}** in the example.

Also, note how comments can appear anywhere in a test file by writing # as the first character of a line.

3.2 RESOURCE RESOLUTION

Locating test resources, both from the command line execution scripts or from within test files themselves, can be done easily by using *resource locators*. From the example above:

```
%TEARDOWN[script] {ds:vdp} file:../testscripts/join/remove_all_users.vql
```

Resources can be located both as filesystem items or as classpath resources. The locator prefix will indicate the method that should be used for accessing the resource:

- **file:**`/home/user/DenodoTesting/tests/mytest.denodotest`
- **classpath:**`/resources/tests/mytest.denodotest`

If a prefix is not specified, `file:` will be used as default.

Also note that resource locators can be relative, which is specially convenient inside test files for linking to other files in the same folder structure in an installation-independent manner:

```
%EXTENDS ../jointest.denodotest
```

3.3 ESCAPING SPECIAL CHARACTERS

Six types of escape operations should be applied in a test file.

First, **unicode escaping**, which allows writing any unicode character by means of its U-HEXA escape code: **\uXXXX** being XXXX its four-digit hexadecimal character code. These escapes can be applied anywhere in a test file and will be processed by the Testing Tools file parsing mechanism, so any executed directive will always see the already-unescaped equivalent. Unicode escaping keeps tests files ASCII-compliant. This is recommended for a better cross-platform interoperation.

Second, given the possibility to specify variables defined by a CONTEXT directive anywhere in a SETUP, EXECUTION, RESULT, TRACE or TEARDOWN directive (see the **CONTEXT directive section** for more information), **the ${ sequence should be escaped** as \${. See:

```
%RESULTS[data]
parameter
"File C:/Denodo Platform/logs/vdp/vdp\${vdp.instance.log}.log"
```

Third, directive contexts are comma separated, therefore values containing commas like e.g. date patterns, require to escape them. See:

```
%RESULTS[data]
{type:subset, datepattern:MMM d\, yyyy h:mm:ss a z}
actor_id,first_name,last_name,last_update
1,PENELOPE,GUINESS,"Feb 14, 2006 6:34:33 PM PST"
```

Fourth, writing **SQL/VQL queries** might need to follow the escaping rules of these languages. The same applies for **CSV data**.

Fifth, **MVEL expressions**, in the TRACE directive, use literals surrounded with quotes (single or double), therefore literals containing quotes should be escaped. Backslashes should be escaped too. See:

```
%TRACE
# escape single quotes and backslash
EXECUTION   PLAN.VIRTUAL   PLAN.JDBC   WRAPPER.JDBC   ROUTE.parameters   ==
'[o\'donnell, a\\b]'
```

Sixth, elements inside of **complex elements** are delimited by single quotes. If an element has a single quote it must be escaped using twice this symbol. See:

```
%RESULTS[data]
name,struct
Smith,"[{'John Smith', '2'}, {'Diane O''Donnell', '3'}]"
```

3.4 EXTENDS DIRECTIVE

EXTENDS is an optional directive that defines child/parent relationships for **test inheritance**.

Test inheritance allows you to build a base *skeleton* test that contains all the common directives and data for testing that child tests can override or aggregate.

Inheritance works in a different manner depending on each directive. See their documentation for more information.

The EXTENDS directive expects a path to locate the parent of the current test, which can be relative or absolute.

Example:

```
%EXTENDS ../jointest.denodotest
```

Note that, although test file names are required to have the .denodotest extension, tests can inherit directives from any file, whatever the extension and disregarding the fact that the parent test is actually a complete and valid test or not. This is useful for using some files as mere repositories of directive values which are common to several tests:

```
%EXTENDS ../../common/suite.denodocommon
```

3.5 NAME DIRECTIVE

NAME is an optional directive that allows the specification of a **test name**. It can be useful for identifying the test at the results reported by the Testing Tool.

If no NAME directive is configured the test is identified by its file path at the test report. However, if it is a test with multivalued variables (see the **CONTEXT directive section**) one identifier per variable will be added to the name in order to show the particular value used in the execution. The notation is TestName_varname0 and it means that the first value of the list specified for varname was used. When there is not a NAME directive, test reports may show MultiTest as the name of the test followed by the variable identifiers.

NAME can inherit values between tests using the EXTENDS directive. Obviously a test can only have one name. Therefore if this directive exists in both parent and child, the child's name is used.

Example:

```
%NAME JoinTest2
```

3.6 DESCRIPTION DIRECTIVE

DESCRIPTION is an optional directive that adds detailed information about the test, mainly for **documentation purposes**.

Test descriptions will be mostly ignored by the Testing Tool, but could be used in custom *Test Reporters* developed by the users to fulfill specific requirements. See the **Test Reporters section** for more information.

DESCRIPTION can inherit values between tests using the EXTENDS directive. Obviously a test can only have one description. Therefore if this directive exists in both parent and child the child description is used.

Example:

```
%DESCRIPTION
This test verifies the join of customer tables from the db2 and extws
sources.
```

3.7 CONTEXT DIRECTIVE

CONTEXT is an optional directive that declares **a list of variables with their values**. These variables can be used in other directives (SETUP, EXECUTION, RESULT, TRACE and TEARDOWN) of the same test or its children using the notation `${some.variable}`.

The CONTEXT directive is a list of `key = value` lines specified in a format equivalent to Java `.properties` files, including multi-line values. See the Java documentation for the `java.util.Properties` class for more detail on this format.

The values of these variables can be plain text or references to properties declared in the global configuration file. In the latter case values are defined as `#{some.global.property}` which allows test files to use **encrypted values**, since only the global configuration file can handle encrypted values (more on this later).

The CONTEXT directive enables test developers to create *template tests* which can be executed with different parameters without duplicating the entire test, just by creating new tests that **extend** the original one changing the **context** variable **values**.

CONTEXT can inherit values between tests using the EXTENDS directive. Context variables are aggregated but in case of duplicated names only the child value is kept.

Example:

```
%CONTEXT
username = jdoe
identifier = 23
password = #{some.global.password}
```

The variables of this directive can also be **multivalued**. A list of values may be specified for a variable and the test will be executed once for each value. Multivalued variables must end with the suffix `[]`.

Example:

```
%CONTEXT
identifier[] = 23, 36
```

If the directive has more than one multivalued variable the test will be run with each possible combination of values.

Example:

```
%CONTEXT
identifier[] = 23, 36
ds[] = mysql, oracle, db2
```

A test with this context will be executed six times using the values in the list below:

1. {identifier = 23, ds = mysql}
2. {identifier = 36, ds = mysql}
3. {identifier = 23, ds = oracle}
4. {identifier = 36, ds = oracle}
5. {identifier = 23, ds = db2}
6. {identifier = 36, ds = db2}

3.8 SETUP AND TEARDOWN DIRECTIVES

SETUP and TEARDOWN are optional directives. They specify the SQL/VQL queries or scripts that **prepare and clean the environment** before and after the execution of a test.

Both directives can inherit values between tests using the EXTENDS directive. Values are aggregated and the order of appearance in test files determines their execution order. Child values will therefore always be applied after parent values.

These directives **require a qualifier**. Two are supported: `query` and `script`.

3.8.1 SETUP[query] and TEARDOWN[query]

query defines an SQL/VQL query that will be executed on a JDBC data source. The specific data source to be used is specified using a ds:datasource_id value in the directive context.

Data sources are declared at the Testing Tool configuration file. See the **Data Sources configuration section** for more information.

Example:

```
%SETUP[query]
{ds:vdp}
INSERT INTO users (id,name) VALUES (8, 'Laetitia Cauliflower')
```

```
%TEARDOWN[query] {ds:vdp} REMOVE FROM users WHERE id = 8
```

These samples above will execute the specified queries on the vdp data source.

3.8.2 SETUP[script] and TEARDOWN[script]

script points to an SQL/VQL script file that will be executed on the JDBC data source. The specific data source to be used is specified using a ds:datasource_id value in the directive context.

Data sources are declared at the Testing Tool configuration file. See the **Data Source configuration section** for more information.

Script files are referenced using prefix-based resource locators. See the **Test locator section** for more information.

Example:

```
%SETUP[script] {ds:vdp} classpath:/testscripts/join/add_all_users.vql
```

```
%TEARDOWN[script]                                                {ds:vdp}
classpath:/testscripts/join/remove_all_users.vql
```

These samples above will execute the given scripts, located in the classpath, on the vdp data source.

3.8.3 General *SETUP* and *TEARDOWN*

Directives are executed for single tests. But in the case of SETUP and TEARDOWN directives it could be useful to prepare and clean the environment **only once for a set of tests**.

If a test folder contains a file named **`first.denodotest`** this test file will be executed at the beginning of the set of tests contained in the folder. `first.denodotest` should contain only the `SETUP` directive/s that prepares the general environment needed for executing the rest of the tests in the folder.

If a test folder contains a file named **`last.denodotest`** this test file will be executed at the end of the set of tests contained in the folder. `last.denodotest` should contain only the `TEARDOWN` directive/s that cleans the general environment needed for executing the rest of the tests.

3.9 `EXECUTION` DIRECTIVE

`EXECUTION` is a **mandatory** directive. It defines the **`SQL/VQL`** query or script or the **REST** operation that will be **executed on VDP** returning a set of results. These results will be compared with the expected results specified at the `RESULTS` directive in order to determine whether the test has been successful or not.

`EXECUTION` can inherit values between tests using the `EXTENDS` directive. Only one instance of this directive can be specified at a test file. Therefore if this directive exists in both parent and child the child value is used.

This directive **requires a qualifier**. Three qualifiers are supported: `query`, `script` and `ws`.

3.9.1 `EXECUTION[query]`

`query` defines an `SQL/VQL` query that will be executed on the VDP data source. The specific data source to be used is specified using a `ds:datasource_id` value in the directive context.

VDP data sources are declared at the Testing Tool configuration file. See the **Data Sources configuration section** for more information.

Several `datepattern` context values can optionally be specified determining which date pattern/s the dates follow in case **they are specified in textual mode** in the data set instead of using Date types.

A `complex_ordered` context value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%EXECUTION[query]
{ds:vdp}
SELECT id, name, login, role
FROM users_full WHERE manager_login = '${username}'
```

The sample above will execute the specified query on the `vdp` data source. Note that the variable `${username}` will be resolved from the `CONTEXT` directive and its value --in String form-- put in place before executing the query.

Example with `datepattern`:

row_id	date_as_text	date_as_text_2	date_as_text_3
1	Thu, 09 Apr 2015 05:19:41 PDT	19/12/2012	2010-01-01T12:00:00+0100
2	Mon, 01 Dec 2014 11:43:02 PST	29/04/2013	2007-11-21T12:00:30+0500
3	Wed, 25 Sep 2013 10:59:41 PDT	05/03/2014	2000-01-24T07:23:56-0100
4	Sat, 11 Feb 2012 04:10:41 PST	09/04/2015	2014-10-03T05:39:00-0500

```
%EXECUTION[query]
{ds:vdp,
datepattern:E\, d MMM yyyy H:mm:ss z,
datepattern:d/MM/yyyy,
datepattern: iso8601
}
select * from dates_as_text where row_id = 3
```

The sample above shows the table `date_as_text`. This table is an ad-hoc sample to illustrate the `datepattern` context value feature. The table has three columns containing dates of type `text`, each column following a different date pattern:

- `date_as_text` follows the pattern marked in red
- `date_as_text_2` follows the pattern marked in purple
- `date_as_text_3` follows the pattern marked in green.

3.9.2 EXECUTION[script]

`script` specifies an `SQL/VQL` script file that will be executed on the VDP data source. The specific data source to be used is specified using a `ds:datasource_id` value in the directive context.

VDP data sources are declared at the Testing Tool configuration file. See the **Data Sources configuration section** for more information.

Script files are referenced using prefix-based locators that determine where to search for files: in the classpath or in the file system. See the **Test locator section** for more information.

Several `datepattern` context values can optionally be specified determining which date pattern/s follow the dates in case **they are specified in textual mode** in the data set instead of using Date types.

A `complex_ordered` context value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%EXECUTION[script] {ds:vdp} file:testscripts/join/do_big_join.vql
```

The sample above will execute the `testscripts/join/do_big_join.vql` script, located relatively in the file system, on the `vdp` data source.

3.9.3 EXECUTION[ws]

`ws` specifies an HTTP GET request that will be sent to a published REST Web service in VDP.

REST web services in VDP return data in four different formats: JSON, XML, HTML and RSS. For testing a different format from the default representation the URL should contain the `$format` parameter as in the example below.

If the access to the web service is protected the directive context variables `user` and `password` configure the credentials needed for the **web service authentication**. The Testing Tool supports three authentication methods:

- HTTP Basic
- HTTP Basic with VDP
- HTTP Digest

Example:

```
%EXECUTION[ws]
{user:admin, password:${pwd}}
http://localhost:9090/server/admin/staff/views/staff?$format=JSON
```

The sample above invokes the published web service `staff` that lists the elements of the view `staff` in JSON. The web service uses HTTP Basic authentication, the login is `admin` and the password is provided encrypted in the general configuration file of the Testing Tool.

Several `datepattern` context values can optionally be specified determining which date pattern/s follow the dates in the data set.

A `complex_ordered` context value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

3.9.3.1 RSS format

RSS mappings provided to configure the web service in VDP are also required by the Tool for testing the RSS format. They are denoted using a mandatory directive context called `mappings`.

Mappings are expressed using the MVEL syntax for maps: `[rss_field1:field1, …]`. For example: `["guid":"film_id", "pubDate":"release_year", "category#domain":"taxonomy"]`.

The character `#` denotes mappings with tag attributes: `category#domain`, `guid#isPermaLink`, ... This syntax is optional, you could use `domain`, `isPermaLink`... but in the case of mapping `enclosure#url` or `source#url`, `#` is mandatory to distinguish which `url` is being mapped.

When a blob field of VDP is mapped to an RSS field the mappings should state explicitly that we are mapping a binary field with the suffix **`:binary`** e.g., `"author" : "photo:binary"`.

Example:

```
%EXECUTION[ws]
{mappings: ["title":"film_title"\, "description":"film_desc"\,
 "link":"film_link"\, "author":"director"] }

http://localhost:9090/server/admin/film/views/film?$format=rss
```

Note that directive contexts are comma separated, therefore commas between mappings require to be **escaped**.

3.10 RESULTS DIRECTIVE

`RESULTS` is a **mandatory** directive that declares the **expected results of a test.** Returned results will be compared against the results obtained at the `EXECUTION` directive in order to determine whether the test has been successful or not.

`RESULTS` can inherit values between tests using the `EXTENDS` directive. Only one instance of this directive can be specified at a test file. Therefore if this directive exists in both parent and child the child value is used.

This directive **requires a qualifier**. Six qualifiers are supported: `query`, `script`, `csv`, `data`, `ws` and `exception`.

3.10.1 RESULTS[query]

`query` defines an SQL/VQL query that will be executed on the JDBC data source. The specific data source to be used is specified using a `ds:datasource_id` value in the directive context.

Data sources are declared at the Testing Tool configuration file. See the **Data sources configuration section** for more information.

Actual and expected result matching will be performed by object equality, at the Java objectual level. Some specific tweaks however will be applied for helping the match of numeric and or date-related data.

A `type` context value can optionally be specified determining whether the Testing Tool should expect the results of this query to be exactly the entire data set returned by the `EXECUTION` directive, a subset or a superset of it. Values are therefore `type:full` (default), `type:subset` and `type:superset`.

An `ordered` value can also be set to indicate whether the order of the result rows is important for the result matching, `ordered:true` or not, `ordered:false`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. `false` is the default value for results coming from SQL databases.

Several `datepattern` context values can optionally be specified determining which date pattern/s follow the dates in case **they are specified in textual mode** in the data set instead of using Date types.

A `complex_ordered` value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%RESULTS[query]
{ds:db2, ordered:yes}
SELECT id, name, login, role
FROM users_full WHERE manager_login = '${username}'
```

The sample above will execute the specified query on the `db2` data source. Note that the `${username}` variable will be resolved from the `CONTEXT` directive and its value put in place before executing the query.

In the example above, due to `type` not having been specified the test execution will be considered OK if tuples obtained are exactly the same as the results obtained from the `EXECUTION` directive, no matter the order of their rows. Also, having `ordered:yes` having been specified, results will only be considered to match if they come in the same order.

3.10.2 RESULTS[script]

`script` specifies an `SQL/VQL` query that should be executed on the JDBC data source. The specific data source to be used is specified using a `ds:datasource_id` value in the directive context.

Data sources are declared at the Testing Tool configuration file. See the **Data Sources configuration section** for more information.

Script files are referenced using prefix-based locators that determine where to search for files: in the classpath or in the file system. See the **Test locator section** for more information.

Actual and expected result matching will be performed by object equality, at the Java objectual level. Some specific tweaks however will be applied for helping the match of numeric and or date-related data.

A `type` context value can optionally be specified determining whether the Testing Tool should expect the results of this query to be exactly the entire data set returned by the `EXECUTION` directive, a subset or a superset of it. Values are therefore `type:full` (default), `type:subset` and `type:superset`.

An `ordered` value can also be set to indicate whether the order of the result rows is important for result matching, `ordered:true` or not, `ordered:false`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. `false` is the default value for results coming from SQL databases.

Several `datepattern` context values can optionally be specified determining which date pattern/s follow the dates in case **they are specified in textual mode** in the data set instead of using Date types.

A `complex_ordered` value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%RESULTS[script]
{ds:db2, type:subset}
file:testscripts/join/do_big_join_results.vql
```

The example above will execute the `testscripts/join/do_big_join_results.vql` script, located relatively in the file system, on the `db2` data source.

Because of the `type:subset` context value the test execution will be considered OK if all the tuples obtained are present in the results obtained from the `EXECUTION` directive. Row order will not be considered important due to the `ordered` context value not being specified (and `[script]` being executed against a SQL data source).

3.10.3 RESULTS[csv]

`csv` defines the expected result set by means of a CSV file, in the variant of CSV exported by Microsoft Excel as specified at RFC 4180. CSV file should contain the **names of the columns** as its first line.

Note also that Microsoft Excel installations in non-English setups might use the comma (`,`) as a decimal separator instead of its standard use as a field separator (will use `;` instead). Files might have to be manually reviewed and modified in such case.

CSV files are referenced using prefix-based locators that determine where to search for files: in the classpath or in the file system. See the **Test locator section** for more information.

Result matching will be performed in-memory. Given the textual nature of CSV data and the lack of data type-related metadata, matching some kinds of data will need the specification of date-related and complex-type data in the following text-friendly formats:

- Dates should be specified in ISO8601 format.
  Although several `datepattern` context values can be specified defining an alternate date format/s when Dates do not follow the ISO8601 format or it is quite laborious to convert them to ISO8601 format.
  Notice that `ISO8601` is also a legal value for the `datepattern` context value.

  Since Denodo Testing Tool for Denodo 7.0 the following **patterns are included** out of the box:

  - `yyyy-MM-dd` for `localdate` type
  - `yyyy-MM-dd HH:mm:ss.SSS` for `timestamp` type
  - `yyyy-MM-dd HH:mm:ss.SSSZ` for `timestamptz` type
  - `HH:mm:s` for `time` type

- Complex objects should be specified using their own string format. `Struct` types are delimited by braces (`{}`). `Array` types are delimited by brackets (`[]`) and all its elements are `Struct` type. Simple elements are surrounded by single quotes (`''`).

For more info, see the **RESULTS[data] section** for examples of how to specify `Date`, `Array` and `Struct` types in CSV format.

A `type` context value can optionally be specified determining whether the Testing Tool should expect the results of this query to be exactly the entire data set returned by the `EXECUTION` directive, a subset or a superset of it. Values are therefore `type:full` (default), `type:subset` and `type:superset`.

An `ordered` value can also be set to indicate whether the order of the result rows is important for result matching, `ordered:true` or not, `ordered:false`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. `true` is the default value for results coming from CSV.

A `complex_ordered` value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%RESULTS[csv] { type:full } file:testscripts/data/join_results.csv
```

The sample above will read the `testscripts/data/join_results.csv` file, searched in the file system.

Because of the `type:full` value the test execution will be considered OK if data obtained are exactly the same as the results obtained from the `EXECUTION` directive. Row order is important due to `ordered` not being specified.

3.10.4 RESULTS[data]

`data` specifies the expected result data set inlined in the test file itself, in CSV format, in the variant of CSV exported by Microsoft Excel as specified at RFC 4180. CSV data should include the **columns names** as its first row. Note that the comma will be used as a field separator, according to the RFC standard.

Result matching will be performed in-memory. Given the textual nature of CSV data and the lack of data type-related metadata, matching some kinds of data will need the specification of date-related and complex-type data in the following text-friendly formats:

- Dates should be specified in ISO8601 format.
  Although several `datepattern` context values can be specified defining an alternate date format/s when Dates do not follow the ISO8601 format or it is quite laborious to convert them to ISO8601 format.
  Notice that `ISO8601` is also a legal value for the `datepattern` context value.

  Since Denodo Testing Tool for Denodo 7.0 the following **patterns are included** out of the box:

  - `yyyy-MM-dd` for `localdate` type
  - `yyyy-MM-dd HH:mm:ss.SSS` for `timestamp` type
  - `yyyy-MM-dd HH:mm:ss.SSSZ` for `timestamptz` type
  - `HH:mm:s` for `time` type

- Complex objects should be specified using their own string format. `Struct` types are delimited by braces (`{}`). `Array` types are delimited by brackets (`[]`) and all its elements are `Struct` type. Simple elements are surrounded by single quotes (`''`).

Examples of how to specify `Date` in CSV format according to ISO8601:

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINESS	Wed Feb 15 03:34:33 CET 2006
2	NICK	WAHLBERG	Wed Feb 15 03:34:33 CET 2006
3	ED	CHASE	Wed Feb 15 03:34:33 CET 2006
4	JENNIFER	DAVIS	Wed Feb 15 03:34:33 CET 2006

```
%RESULTS[data]
actor_id,first_name,last_name,last_update
1,PENELOPE,GUINESS,2006-02-15T03:34:33+01:00
2,NICK,WAHLBERG,2006-02-15T03:34:33+01:00
3,ED,CHASE,2006-02-15T03:34:33+01:00
4,JENNIFER,DAVIS,2006-02-15T03:34:33+01:00
```

Examples of how to specify `Date` in CSV format with a date pattern:

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINESS	Wed Feb 15 03:34:33 CET 2006
2	NICK	WAHLBERG	Wed Feb 15 03:34:33 CET 2006
3	ED	CHASE	Wed Feb 15 03:34:33 CET 2006
4	JENNIFER	DAVIS	Wed Feb 15 03:34:33 CET 2006

```
%RESULTS[data]
{datepattern : EEE MMM d HH:mm:ss ZZZ yyyy}
actor_id,first_name,last_name,last_update
1,PENELOPE,GUINESS,Wed Feb 15 03:34:33 CET 2006
2,NICK,WAHLBERG,Wed Feb 15 03:34:33 CET 2006
3,ED,CHASE,Wed Feb 15 03:34:33 CET 2006
4,JENNIFER,DAVIS,Wed Feb 15 03:34:33 CET 2006
```

Examples of how to specify `Array` types in CSV format:

RESU... -> appender -> param

name	value
encoding	UTF-8
File	C:/Denodo Platform/logs/vdp/vdp${vdp.instance.log}.log
Append	true
MaxFileSize	10MB
MaxBackupIndex	7

```
%RESULTS[data]
param
"[{'encoding',  'UTF-8'},  {'File',  'C:/Denodo  Platform/logs/vdp/vdp\$
{vdp.instance.log}.log'},  {'Append',  'true'},  {'MaxFileSize',  '10MB'},
{'MaxBackupIndex', '7'}]"
```

Examples of how to specify `Struct` types in CSV format:

RESU... -> appender -> layout

class	param
org.apache.log4j.PatternLayout	[Register]...

RESU... -> appender -> layout -> param

name	value
ConversionPattern	%-4r [%t] %-5p %d{yyyyMMddHHmmssSSS} %c %x - %m %n

```
%RESULTS[data]
layout
"{'org.apache.log4j.PatternLayout', {'ConversionPattern', '%-4r [%t] %-5p
%d{yyyyMMddHHmmssSSS} %c %x - %m  %n'}}"
```

A `complex_ordered` value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. In the example below, where `complex_ordered` is false, the test execution will be considered OK if the array elements are the same, no matter the order of them.

Example of `Array` type using `complex_ordered`:

RESU... -> appender -> param

name	value
encoding	UTF-8
File	C:/Denodo Platform/logs/vdp/vdp${vdp.instance.log}.log
Append	true
MaxFileSize	10MB
MaxBackupIndex	7

```
%RESULTS[data]
{complex_ordered: false}
param
"[{'File',   'C:/Denodo   Platform/logs/vdp/vdp\${vdp.instance.log}.log'},
{'encoding',   'UTF-8'},   {'Append','true'},   {'MaxFileSize',   '10MB'},
{'MaxBackupIndex', '7'}]"
```

A `type` context value can optionally be specified determining whether the Testing Tool should expect the results of this query to be exactly the entire data set returned by the `EXECUTION` directive, a subset or a superset of it. Values are therefore `type:full` (default), `type:subset` and `type:superset`.

An `ordered` value can also be set to indicate whether the order of the result rows is important for result matching, `ordered:true` or not, `ordered:false`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. `true` is the default value for results coming from CSV.

Example:

```
%RESULTS[data]
{type:subset}
id,name,role
7,Joe Tomato,admin
234,Katherine Carrot,hr
```

The sample above will read the CSV-formatted data from the test file itself.

Because of the `type:subset` context value, the test execution will be considered OK if all the tuples specified in `RESULTS` are present in the results obtained from the `EXECUTION` directive. Row order is important due to `ordered` value not being specified.

3.10.5 RESULTS[ws]

`ws` specifies an HTTP GET request that will be sent to a REST Web service.

A `type` context value can optionally be specified determining whether the Testing Tool should expect the results of this query to be exactly the entire data set returned by the `EXECUTION` directive, a subset or a superset of it. Values are therefore `type:full` (default), `type:subset` and `type:superset`.

An `ordered` value can also be set to indicate whether the order of the result rows is important for the result matching, `ordered:true` or not, `ordered:false`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`. `false` is the default value for results coming from web services.

Several `datepattern` context values can optionally be specified determining which date pattern/s follow the dates in the data set.

Since Denodo Testing Tool for Denodo 7.0 the following patterns are included out of the box:

- `ISO8601`
- `RFC 822` date format for `RSS`
- `yyyy-MM-dd'T'HH:mm:ss` for `timestamp` type

A `complex_ordered` value can be specified in order to disregard the order of the elements of complex data types. When `complex_ordered` is false the result matching does not take into account the order of the items. Default value is `true`. Values `on`, `y`, `t` or `yes` (case insensitive) are also considered as `true`.

Example:

```
%RESULTS[ws]
{ordered:yes}
http://www.acme.com/phonebook/UserDetails?lastName=Doe
```

The sample above will search for the users details with the specified last name in the URL parameter.

In the example above, due to `type` not having been specified the test execution will be considered OK if tuples obtained are exactly the same as the results obtained from the `EXECUTION` directive. Also, the results will only be considered to match if they come in the same order due to `ordered:yes`.

3.10.6 RESULTS[exception]

`exception` **expects an exception** as a result of the execution of the `EXECUTION` directive. The test is successful when such an exception is thrown and it fails if a different one or no exception is thrown.

This can be useful for testing security restrictions.

The value specified will be matched against the exception obtained during the test execution (if any) in the following way:

1. The test will be considered successful if the specified value is the name of an exception class that the raised exception (or any of its causes) implements or extends.

2. Otherwise, the test will be considered successful if the specified value (as text) is found in the exception message or any of its causes.

3. The test will fail in any other case.

Example:

```
%RESULTS[exception] User ${username} has not enough privileges
```

The sample above will search for this message in the exception raised, if any, during test execution.

Also:

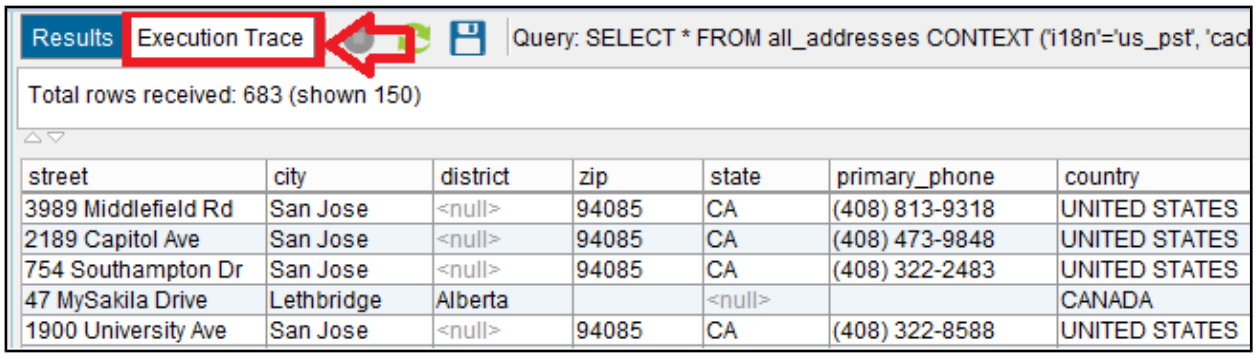
```
%RESULTS[exception] java.lang.IOException
```

The sample above determines that the test should succeed if a `java.lang.IOException` (or a subclass) is thrown.

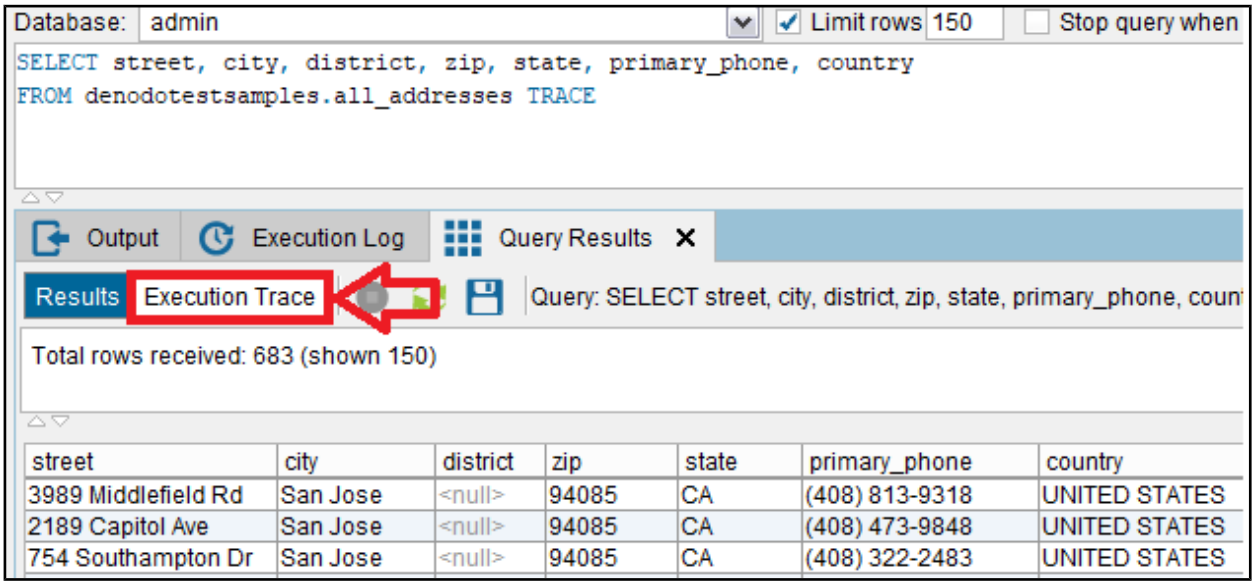
3.11 TRACE DIRECTIVE

TRACE is an optional directive that **checks values from the VDP execution trace** such as number of processed rows, state, etc. using MVEL expressions.

The trace of a query is obtained from the VDP Administration Tool clicking the button `View execution trace` once the execution of the statement has finished. `View execution trace` is only enabled if the `Execute with TRACE` checkbox was selected in the query execution dialog. Or, when using VQL Shell if the statement have the TRACE clause at the end.

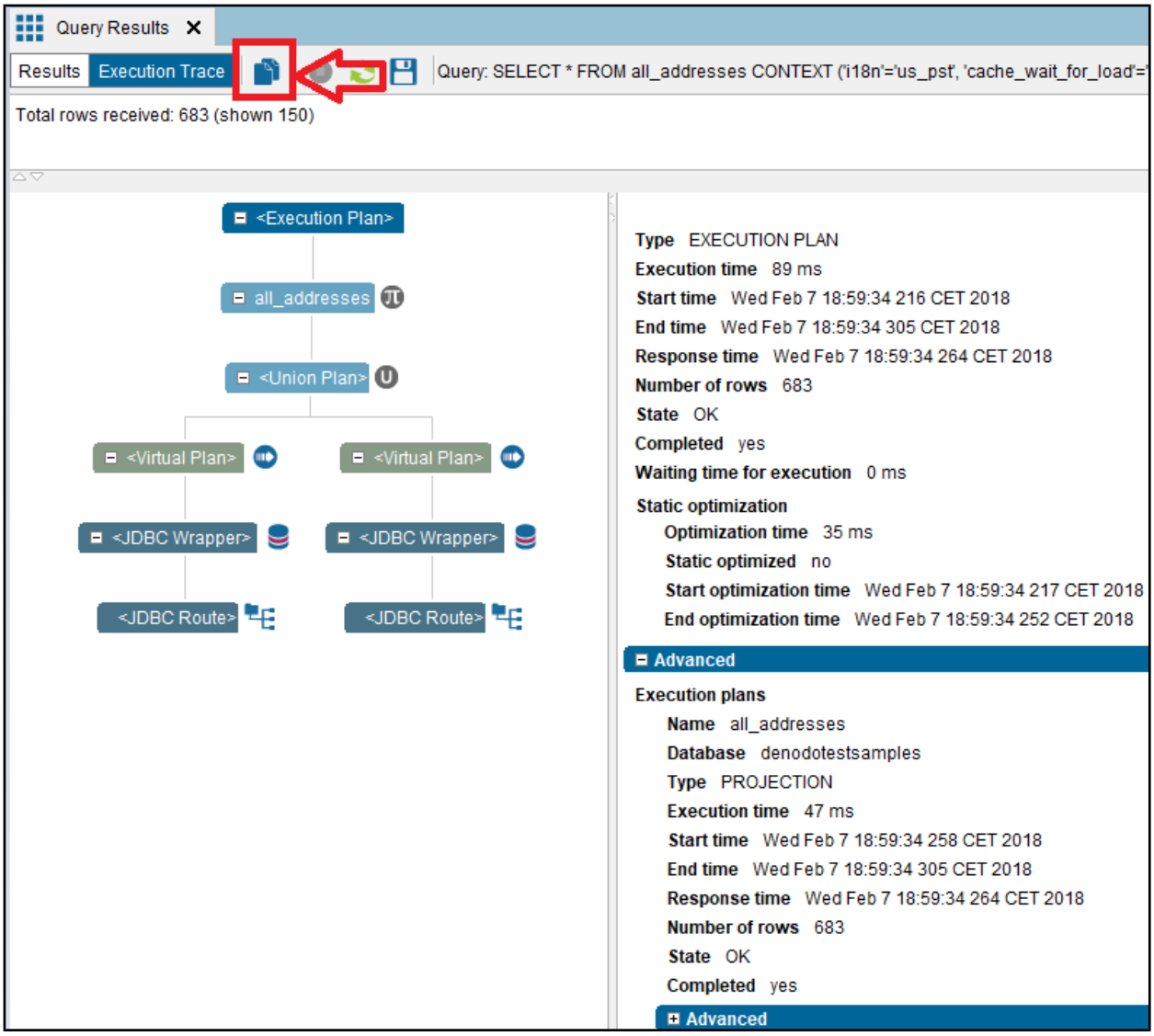


Access the trace from execution dialog



Access to the trace from VQL Shell

The VDP execution trace:



Execution trace

is a tree where each node represents one of these elements:

- The tree root, which is called the “Execution Plan”. It contains information about the query.
- An intermediate view involved in the execution of the statement
- A retrieval of data from a source.

Each node has its own attributes whose values can be checked using MVEL expressions to navigate the tree in its textual form. This textual form can be obtained by clicking the `Copy trace to clipboard` button from the Execution trace view.

The following is a text fragment of the trace of the image above:

```
EXECUTION PLAN (
      ...
      numRows = 683
      state = OK
     ...
      PROJECTION PLAN (
            numRows = 683
            fields = [street, city, district, zip, state,
                      primary_phone, country]
            memoryLimitReached = false
           projectedFields = [street, city, district, zip,
                       upper(state) AS state, primary_phone,
                       upper(country) AS country]
           ...
            UNION PLAN (
                  numRows = 683
                ...
                  VIRTUAL PLAN (
                        name = _p_acme_address
                       numRows = 80
                        state = OK
                      ...
                )
                VIRTUAL PLAN (
                       name = _p__sakila_complete_address
                        numRows = 603
                        state = OK
                      ...
                )
           )
       )
  )
```

With MVEL you can reference node attributes from the trace and check their values. `EXECUTION PLAN` is the tree root. The properties of the root can be referenced using the dot notation, e.g. `EXECUTION PLAN.PROJECTION PLAN.numRows`. The multivalued attributes, like `BASE PLAN` in the example, are referenced using the indexed notation e.g. `VIRTUAL PLAN[0]` and `VIRTUAL PLAN[1]`.

The following is an example of several `TRACE` directives that return true when checking their MVEL expressions against the VDP trace displayed above:

```
%TRACE
EXECUTION PLAN.PROJECTION PLAN.UNION PLAN.VIRTUAL PLAN[1].numRows == 603

%TRACE
# String literals may be denoted by single or double quotes
EXECUTION PLAN.state == 'OK'

%TRACE
```

```
# Checks one trace fragment is equal to another trace fragment
EXECUTION  PLAN.PROJECTION  PLAN.UNION  PLAN.VIRTUAL  PLAN[0].state  ==
EXECUTION PLAN.PROJECTION PLAN.UNION PLAN.VIRTUAL PLAN[1].state

%TRACE
# Uses variables from the context directive ${viewname}
Execution  Plan.PROJECTION  PLAN.UNION  PLAN.VIRTUAL  PLAN[0].name  ==  '$
{viewname}'
```

TRACE can inherit values between tests using the EXTENDS directive.

TRACE **cannot be used with the** ws context qualifier.

4 TESTING TOOL EXECUTION

The Denodo Testing Tool is started using a command-line script which requires the following parameters:

- A configuration file (the route to it).
- A test file or folder. In the latter case the Testing Tool will execute everything it finds inside, at any level, in sequence.

These parameters are referenced using prefix-based resource locators that determine where to search for files: in the classpath or in the file system. See the **Test locator section** for more information.

Example:

```
bin> denodo-test.bat file:../conf/configuration.properties file:<TEST_DIRECTORY>
```

Note that the `file:` prefix can be omitted as it is the default.

4.1.1 Optional parameters

`--failed-resumable` Only failed tests in previous execution are launched.

With this option, if there are failed tests, a metafile is created in the root folder of the tests. The name of this file is `#denodo-failed-tests-execution.metafile`. This file contains all tests that fails in the execution. So, in the following execution, only the tests that failed, will be launched again. After reading this metafile, the file is deleted. If it is necessary the file will be created again. This file only is deleted if you use the option `--failed-resumable`. It has to be the third parameter.

```
bin> denodo-test.bat file:../conf/conf.properties file:C:/Work/vdp_tests
--failed-resumable
```

4.2 CONFIGURATION FILE

This is a `.properties` file containing general tool execution data: configuration parameters, data source definitions, global variables...

Example:

```
# ------------------------------------------------------------------
# GENERAL CONFIGURATION PARAMETERS
# ------------------------------------------------------------------
encoding=UTF-8
maxRowsInMemoryForMatching=10000

#Optional property
#resultsDatePattern=MMM d, yyyy h:mm:ss zzzz

# ------------------------------------------------------------------
# TEST REPORTER
# ------------------------------------------------------------------
reporter=com.denodo.connect.testing.reporter.ConsoleTestReporter

# Example multiple Test Reporters
# reporter.con=com.denodo.connect.testing.reporter.ConsoleTestReporter
# reporter.csv=com.denodo.connect.testing.reporter.CSVTestReporter
# reporter.csv.pathToOutputFile=src//test//resources//CSVReporter.csv
# reporter.csv.overwriteOutputFileIfExists=true

# ------------------------------------------------------------------
# GLOBAL CONTEXT VARIABLES
# ------------------------------------------------------------------
# some.variable=some value

# ------------------------------------------------------------------
# DATA SOURCE CONFIGURATION
# ------------------------------------------------------------------

# Example VDP data source using the JDBC driver v5.5.0_9
vdp-55.driverClassName=com.denodo.vdp.jdbc.Driver
vdp-55.dbAdapter=denodo-5.5.0_9
vdp-55.jdbcUrl=jdbc:vdb://localhost:9999/admin
vdp-55.username=admin
vdp-55.password=ENC(s2FdirMK4QORq1HZ6tcTTQ==)
vdp-55.connectionTestQuery=select * from dual()

# Example VDP data source using the JDBC driver v6.0.8
vdp-60.driverClassName=com.denodo.vdp.jdbc.Driver
vdp-60.dbAdapter=denodo-6.0.8
vdp-60.jdbcUrl=jdbc:vdb://localhost:9999/admin
vdp-60.username=admin
vdp-60.password=ENC(s2FdirMK4QORq1HZ6tcTTQ==)
vdp-60.connectionTestQuery=select * from dual()

# Example MySQL data source
```

```
mysql.driverClassName=com.mysql.jdbc.Driver
mysql.dbAdapter=mysql-5
mysql.jdbcUrl=jdbc:mysql://localhost/test
mysql.username=test
mysql.password=test
```

- `encoding` is the character encoding that should be used for reading the test files.
- `maxRowsInMemoryForMatching` is the maximum number of rows that should be kept in memory during matching operations for unordered data. The real amount of memory used depends on the real size in bytes of such rows, but this parameter will help limit the amount of memory used. The higher this parameter, the less temporary files the Testing Tool will need for performing unordered data matching.
- `resultsDatePattern` is the date format of the expected results when their `Date` type values does not follow the ISO8601 format. This property can be overridden by the context variable `datepattern` in the `RESULTS` directive of each individual test file. Notice that `ISO8601` is also a legal value of:
  - the `resultsDatePattern` property, for documentation purposes.
  - the `datepattern` context value, for overriding the `resultsDatePattern` property.
- `reporter` is the Test Reporter implementation that will be in charge of reporting all test events (tool start/end, test start/end, etc.). The class declared here must be an implementation of the `com.denodo.connect.testing.reporter.ITestReporter` interface. The Testing Tool provides the console reporter, the CSV reporter, the email reporter and the HTML reporter. But it also allows the implementation of a new Test Reporter in order to fulfill other user requirements. It is also possible to run multiple Test Reporter implementations simultaneously. In order to use this feature it is necessary to specify a name for every Test Reporter (`reporter.name`) and you might give values for properties of a specific Test Reporter (`reporter.name.variable`). See the **Test Reporters section** for more information.
- Note that no global variables are specified in this example. If they were, they could be accessed from any `CONTEXT` directive with `#{some.variable}`

4.2.1 Reporters configuration

The CSV reporter, the email reporter and the HTML reporter need to establish some properties in order to run properly.

4.2.1.1 CSV reporter configuration

This reporter has two properties that must be specified:

- `pathToOutputFile` is the path to the file where the information is going to be written
- `overwriteOutputFileIfExists` is the value to indicate whether the output file must be overwritten if it already exists. If the value is false new data will be added at the end of the existing file

CSV reporter configuration sample:

```
...
reporter.csv=com.denodo.connect.testing.reporter.CSVTestReporter
reporter.csv.pathToOutputFile=src//test//resources//CSVReporter.csv
reporter.csv.overwriteOutputFileIfExists=true
...
```

4.2.1.2 Email reporter configuration

The email-based Test Reporter needs several configuration parameters:

- `username` is the user address from which the email is sent
- `password` is the user's password
- `serverHost` is the SMTP server to connect to
- `serverPort` is the SMTP server port to connect to
- `serverProtocol` is the server protocol
- `smtpAuth` is a boolean to specify whether it has to authenticate. If true, attempt to authenticate the user using the AUTH command. Defaults to false
- `smtpStarttlsEnable` is a boolean to specify whether a TLS-protected connection will be established. If true, the connection will be secured by TLS. Defaults to false
- `smtpQuitwait` is a boolean to specify whether the connection has to be closed. If set to false, the QUIT command is sent and the connection is immediately closed. If set to true, causes the transport to wait for the response to the QUIT command. Defaults to false
- `from` is the value that the server will set in the from field if the configured SMTP server allow to establish a different value from the username
- `recipient` is the recipient or recipients of the email. The addresses can be separated by spaces or commas

- `messageDetailLevel` is the detail level that may be used to send the email reports. Available values are: full, summary and failed. See the **Email Test Reporter section**.

Email reporter configuration sample:

```
...
reporter.email=com.denodo.connect.testing.reporter.EmailTestReporter
reporter.email.username=user_1@example.net
reporter.email.password=ENC(s2FdirMK4QORq1HZ6tcTTQ==)
reporter.email.serverHost=smtp.gmail.com
reporter.email.serverPort=587
reporter.email.serverProtocol=smtp
reporter.email.smtpAuth=true
reporter.email.smtpStarttlsEnable=true
reporter.email.smtpQuitwait=false
reporter.email.from=user_1@example.net
reporter.email.recipient=user_2@example.net,user_3@example.net
reporter.email.messageDetailLevel=full
...
```

4.2.1.3 HTML reporter configuration

This reporter has two properties that must be specified:

- `pathToOutputFile` is the path to the file where the information is going to be written
- `overwriteOutputFileIfExists` is the value to indicate whether the output file must be overwritten if it already exists. If the value is false and the output file already exists an exception will be thrown

HTML reporter configuration sample:

```
...
reporter.html=com.denodo.connect.testing.reporter.HTMLTestReporter
reporter.html.pathToOutputFile=src//test//resources//HTMLReporter.html
reporter.html.overwriteOutputFileIfExists=true
...
```

4.2.2 Data sources configuration

The configuration file also includes the data sources declaration. For each data source referenced from the test files by means of a `ds:x` value the following properties should be configured in this file:

- `x.driverClassName` is the full package name of JDBC driver class

- `x.dbAdapter` is the name of the specific driver that you want to use. It is optional and when it is stated the jar file of the required driver should be in the folder `<TESTING_TOOL_HOME>/drivers/value_of_this_property/`. Otherwise the driver jar should be in the `<TESTING_TOOL_HOME>/lib/` folder.

- `x.jdbcUrl` is the database URL

- `x.username` is the authentication username

- `x.password` is the authentication password

- `x.connectionTestQuery` is the query that will be executed just before a connection is given to you from the pool to validate that the connection to the database is still alive. This property is only required for drivers that do not implement the JDBC4's `Connection.isValid` method, like the VDP JDBC driver

Note that for each configured data source, other than VDP, you will have to **download the JDBC driver** from the corresponding website or repository. Afterwards, in order that the Testing Tool work properly you have three options:

- Copy the jar file to a folder inside `<TESTING_TOOL_HOME>/drivers` and use the `dbAdapter` property to indicate the name of that folder.
  This option allows you to test using different versions of the same driver, e.g. testing two different versions or updates of the Denodo Platform.
  For that you have to copy to `<TESTING_TOOL_HOME>/drivers` the jar files which correspond to the versions you want to use and declare a data source for each one

- Copy the jar file to `<TESTING_TOOL_HOME>/lib`

- Add the JDBC driver path to the `DENODO_TEST_CLASSPATH` environment variable

These three options are mutually exclusive. When you choose to use the driver from `<TESTING_TOOL_HOME>/drivers` using the `dbAdapter` property it cannot be in `<TESTING_TOOL_HOME>/lib` and the JDBC driver path must not be added to the `DENODO_TEST_CLASSPATH` environment variable.

The configuration file supports **encrypted** values, since data source passwords are declared inside. The Testing Tool transparently decrypts encrypted values in the `.properties` file, allowing the mix of both encrypted and not-encrypted values in the same file, using Jasypt.

The Testing Tool expects encrypted configuration parameters to appear surrounded by `ENC(...)`. You can compute these values using the Jasypt CLI tools, and use the `DENODO_TEST_ENCRYPTION_PASSWORD` environment variable or Java VM system property.

This way, you can use:

```
...
vdp.password=ENC(s2FdirMK4QORq1HZ6tcTTQ==)
```

```
...
```

Example of ciphering "admin" password using "mypassword" as the encryption password with Jasypt CLI tools:

```
C:\Sources\jasypt-1.9.2\bin>encrypt.bat input=admin password=mypassword

----ENVIRONMENT-----------------

Runtime: Sun Microsystems Inc. Java HotSpot(TM) 64-Bit Server VM 20.45-b01


----ARGUMENTS-------------------

input: admin
password: mypassword


----OUTPUT----------------------

zrass64ls4LIx5hdFoXXyA==
```

See the **How to encrypt passwords section** for a more detailed explanation.

4.3 TEST LOCATORS

Test resources like files, folders and indexes are located by the Denodo Testing Tool using:

- Prefix-based test locators:
  - `classpath` prefix like `classpath:tests/joins` indicates that the file should be looked up in the classpath
  - `file` prefix like `file:/home/denodo/test/joins` indicates that the file should be looked up in the file system. This is the default prefix is none is specified.

  Prefix-based locators are used with the Testing Tool runner script parameters and with the values of the `script` and `data` qualifiers. Prefix-based locators can be absolute or relative.
- Relative test locators like `../files/jointest.denodotest`, used in:
  - Index files, for locating the test files being indexed
  - EXTENDS directive, for locating the parent test file

4.3.1 Folders and indexes

Folders containing test files and, possibly, other subfolders are considered **test sequences** since test files inside will be executed sequentially.

The test sequence defined by a folder can be modified if the folder contains an index file named: **`index.denodoidx`**. A test index file is simply a list of test files, one in each line, using resource locators. It might include comments.

Index file format sample:

```
sectionone/bigjoin.denodotest
sectiontwo/security
# This part is for the Customer Care department
customer_structures.denodotest
shop_features/features.denodoidx
```

4.4 JUNIT INTEGRATION

Although the command line is the primary way of executing the Testing Tool, there can be cases in which it could be useful to integrate test set executions in Java unit testing infrastructure.

This can be easily done by means of the `com.denodo.connect.testing.TestRunner` class, which can be executed by specifying the same two commands as the command line scripts:

```
@Test
public void testSamples1() throws Exception {
    Assert.assertTrue(
        TestRunner.run(
            "classpath:samples1/conf/configuration.properties",
            "classpath:samples1/tests"));
}
```

The `run()` method will return true if all the specified test executions are successful.

Note also that tests can be created for specific test files, not only folders, which can be convenient for debugging specific scenarios:

```
@Test
public void testSamples1_07() throws Exception {
    Assert.assertTrue(
        TestRunner.run(
            "classpath:samples1/conf/configuration.properties",
            "classpath:samples1/tests/s1test07.denodotest"));
}
```

5 TEST RESULTS

5.1 TEST RESULTS: CONSOLE TEST REPORTER

After executing a test or a sequence of tests using the `ConsoleTestReporter` as the Test Reporter the console shows the results of each test execution.

Example of successful test:

```
[EXECUTION:START]
-------------------------------------------------------
 T E S T S
-------------------------------------------------------

[SEQUENCE:START][C:\Work\vdp_tests] Sequence started: C:\Work\vdp_tests.
--[TEST:END][OK][C:\Work\vdp_tests\client.denodotest][Test-client][502ms][7] Test
run: Test-client. Test executed OK: Obtained and expected results match.
Obtained and expected traces match. Time elapsed: 502ms.
[SEQUENCE:END][OK][C:\Work\vdp_tests][1][1][502ms][7] Sequence finished. Tests
run: 1, OK: 1, Total tuples: 7, Time elapsed: 502ms.
[EXECUTION:END][OK][1][1][502ms][7]
-------------------------------------------------------
Results:
Tests run: 1, OK: 1
Total tuples: 7, Zero-tuple tests: 0
Suite executed in 502ms.
-------------------------------------------------------
```

Example of failed test:

```
[EXECUTION:START]
-------------------------------------------------------
 T E S T S
-------------------------------------------------------

[SEQUENCE:START][C:\Work\vdp_tests] Sequence started: C:\Work\vdp_tests.
--[TEST:END][FAILED][C:\Work\vdp_tests\other.denodotest][Test-other][780ms][7]
Test run: Test-other. Test FAILED!: Row 2, col 2 of the obtained result contained
[GUINESS], but [DONEGAN] was expected at that position.
Obtained trace do NOT match the MVEL expression "EXECUTION PLAN.PROJECTIONPLAN
.INNER JOIN PLAN. base plan[0].JDBC WRAPPER.numRows == 2100". Time elapsed:
780ms.
[SEQUENCE:END][FAILED][C:\Work\vdp_tests][1][0][780ms][7] Sequence finished.
Tests run: 1, OK: 0 (FAILED: 1), Total tuples: 7, Time elapsed: 780ms.
[EXECUTION:END][FAILED][1][0][780ms][7]
-------------------------------------------------------
Results:

Failed tests:
- Test-other(C:\Work\vdp_tests):  Row 2, col 2 of the obtained result contained
[GUINESS], but [DONEGAN] was expected at that position.
Obtained trace do NOT match the MVEL expression "EXECUTION PLAN.PROJECTIONPLAN
.INNER JOIN PLAN. base plan[0].JDBC WRAPPER.numRows == 2100".
```

```
Tests run: 1, OK: 0 (FAILED: 1)
Total tuples: 7, Zero-tuple tests: 0
Suite executed in 780ms.
-------------------------------------------------------
```

5.2 TEST RESULTS: CSV TEST REPORTER

After executing a test or a sequence of tests using the `CSVTestReporter` as the Test Reporter a CSV file will be created with the results of each test execution.

Example of successful test:

	A	B	C	D	E	F
1	Test Result	Resource Name	Test Name	Time Elapsed	Tuples	Test Message
2	OK	C:\Work\tests\test01.denodotest	Test-001	647ms	5	Obtained and expected results match.

Example of failed test:

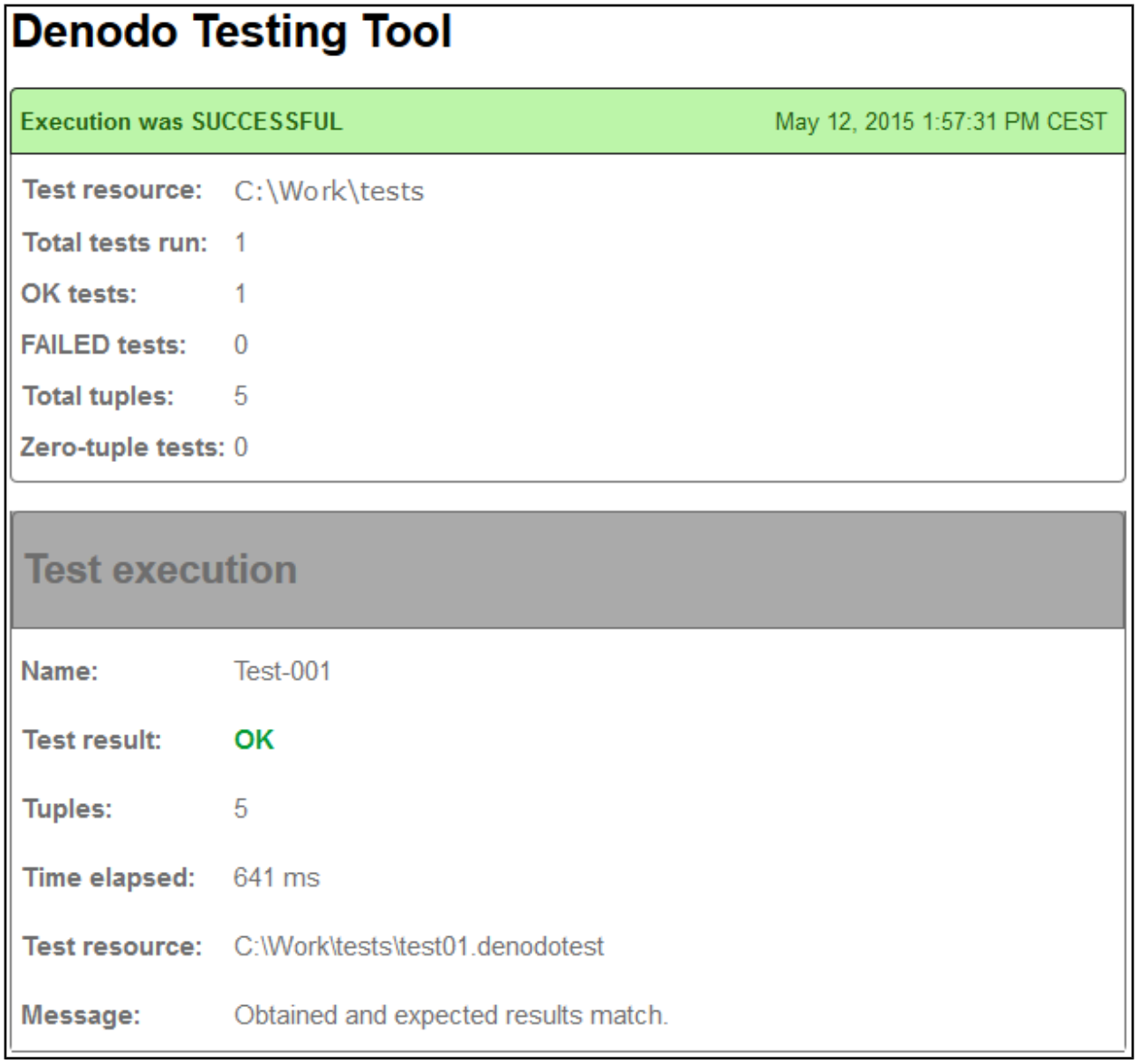
	A	B	C	D	E	F
1	Test Result	Resource Name	Test Name	Time Elapsed	Tuples	Test Message
2	FAILED	C:\Work\tests\test01.denodotest	Test-001	634ms	0	Expected data columns/headers do NOT match those returned by test after execution: expected [[client_identifier, street, city, zip, state, primary_phone, country]] but obtained [[]].

5.3 TEST RESULTS: EMAIL TEST REPORTER

When you run a test or a sequence of tests using the `EmailTestReporter` as the Test Reporter the resulting email depends on the level of detail specified in the configuration, see **Email Test Reporter section**.

5.3.1 Full level

Example of successful test:



Denodo Testing Tool

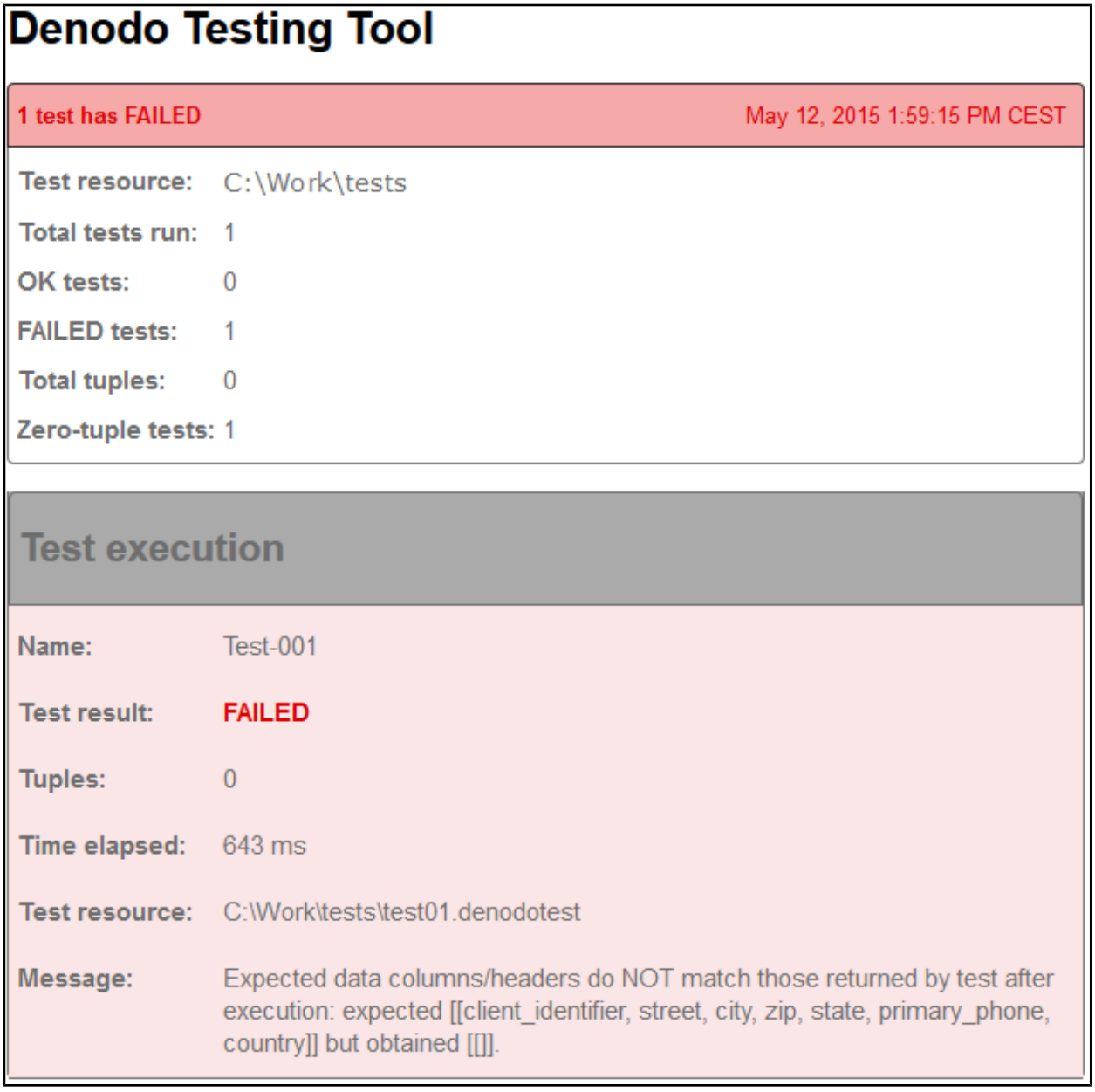
Execution was SUCCESSFUL May 12, 2015 1:57:31 PM CEST

Test resource:	C:\Work\tests
Total tests run:	1
OK tests:	1
FAILED tests:	0
Total tuples:	5
Zero-tuple tests:	0

Test execution

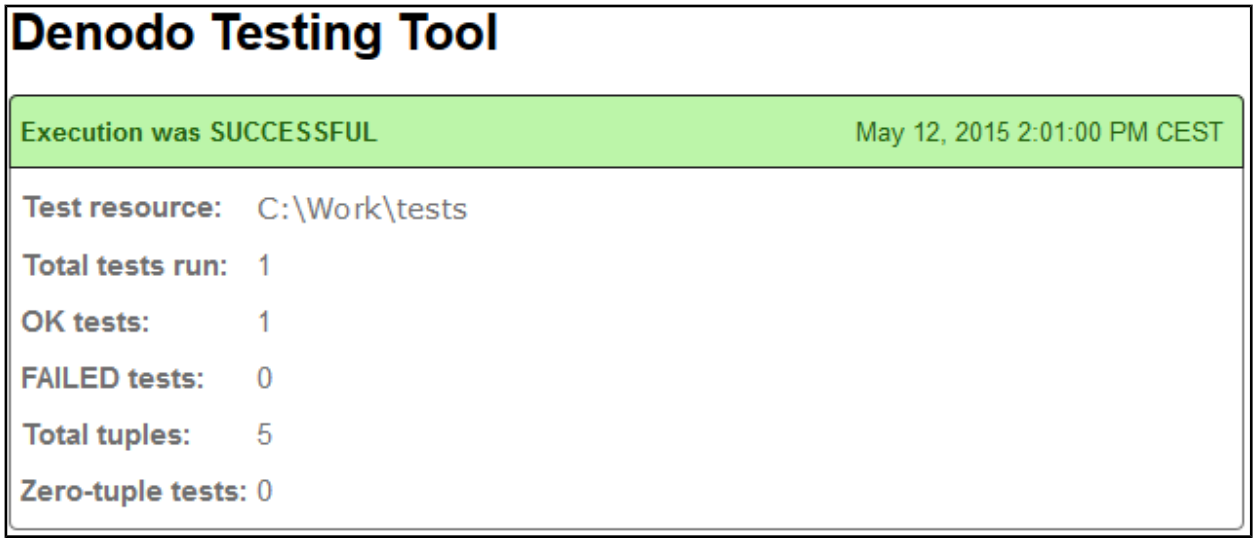
Name:	Test-001
Test result:	**OK**
Tuples:	5
Time elapsed:	641 ms
Test resource:	C:\Work\tests\test01.denodotest
Message:	Obtained and expected results match.

Example of failed test:



5.3.2 Summary level

Example of successful test:

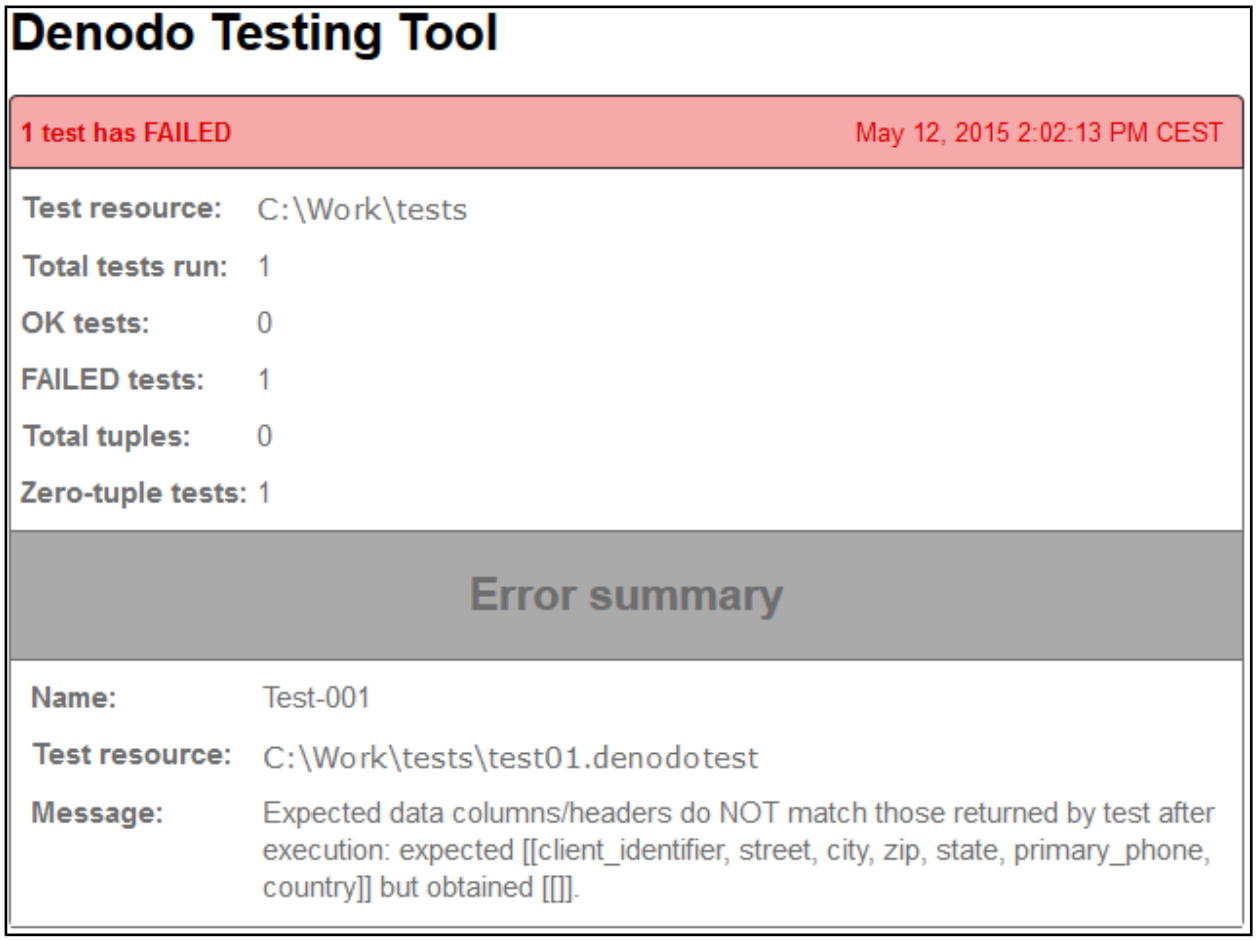


Denodo Testing Tool

Execution was SUCCESSFUL — May 12, 2015 2:01:00 PM CEST

Test resource:	C:\Work\tests
Total tests run:	1
OK tests:	1
FAILED tests:	0
Total tuples:	5
Zero-tuple tests:	0

Example of failed test:



Denodo Testing Tool

1 test has FAILED — May 12, 2015 2:02:13 PM CEST

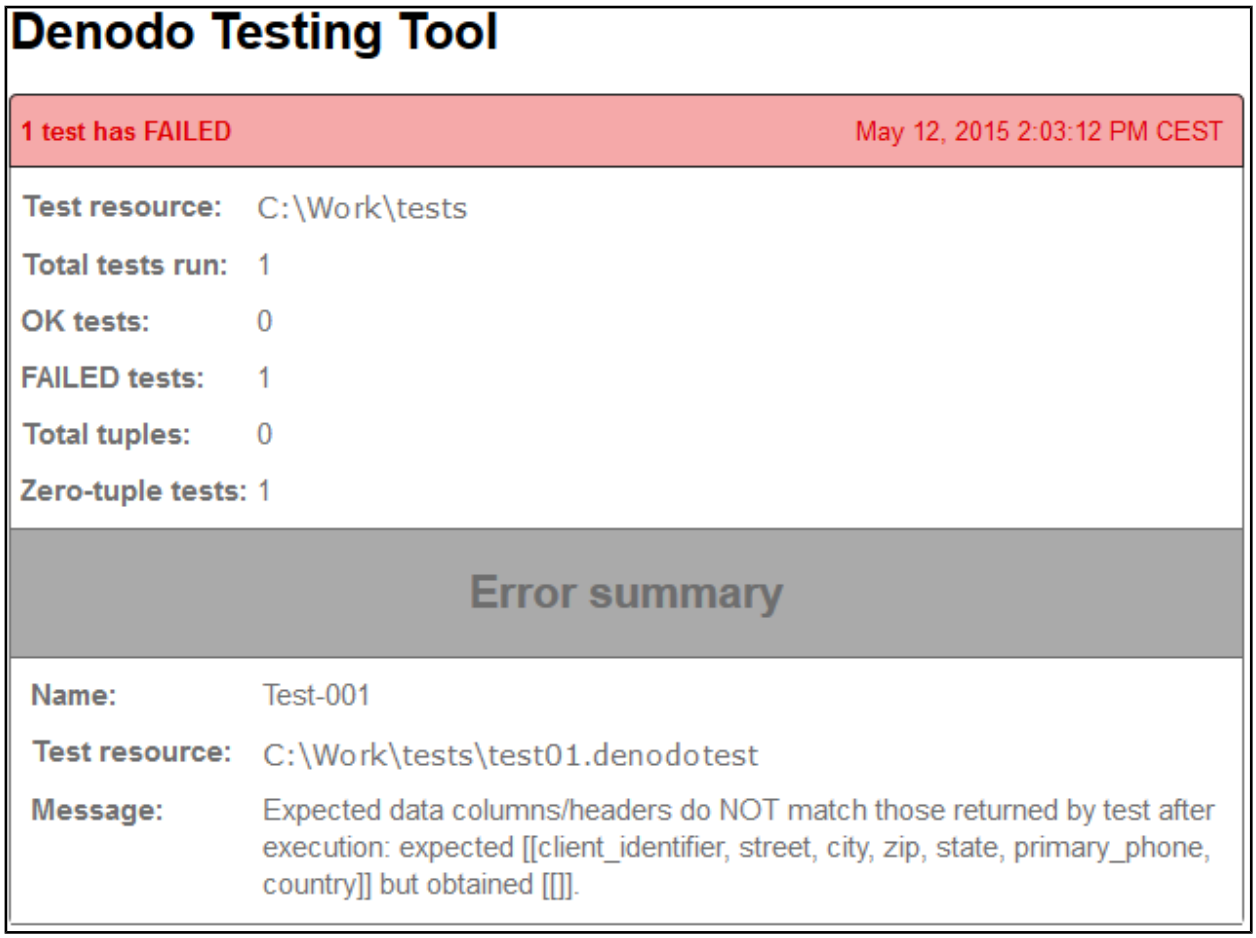
Test resource:	C:\Work\tests
Total tests run:	1
OK tests:	0
FAILED tests:	1
Total tuples:	0
Zero-tuple tests:	1

Error summary

Name:	Test-001
Test resource:	C:\Work\tests\test01.denodotest
Message:	Expected data columns/headers do NOT match those returned by test after execution: expected [[client_identifier, street, city, zip, state, primary_phone, country]] but obtained [[]].

5.3.3 Failed level

Example of failed test:



5.4 TEST RESULTS: HTML TEST REPORTER

After executing a test or a sequence of tests using the `HTMLTestReporter` as the Test Reporter an HTML file will be created with the results of each test execution.

Example of successful test:

Denodo Testing Tool Report

Execution was SUCCESSFUL May 12, 2015 2:04:37 PM CEST

Test resource:	C:\Work\tests
Total tests run:	1
OK tests:	1
FAILED tests:	0
Total tuples:	5
Zero-tuple tests:	0
Time elapsed:	704 ms

Result	Name	Test Resource	Tuples	Time(ms)	Message
OK	Test-001	C:\Work\tests\test01.denodotest	5	704 ms	Obtained and expected results match.

Example of failed test:

Denodo Testing Tool Report

1 test has FAILED May 12, 2015 2:07:25 PM CEST

Test resource:	C:\Work\tests
Total tests run:	1
OK tests:	0
FAILED tests:	1
Total tuples:	0
Zero-tuple tests:	1
Time elapsed:	699 ms

Result	Name	Test Resource	Tuples	Time(ms)	Message
FAILED	Test-001	C:\Work\tests\test01.denodotest	0	699 ms	Expected data columns/headers do NOT match those returned by test after execution: expected [[client_identifier, street, city, zip, state, primary_phone, country]] but obtained [[]].

6 TEST REPORTERS

The Test Reporter handles the following events of the test execution:

- Tool start/end
- Sequence start/end
- Test start/end

Each reported event includes:

- Test/sequence name and the name of its physical resource, e.g. file path
- Test description, if applies

End events also include:

- Test result (OK/FAILED), aggregated in the case of sequences. With:
  - An explanatory message of the result
  - An exception, if any error occurs
- Time spent processing the `EXECUTION` directive
- Number of data tuples compared during the test execution, aggregated in the case of sequences

The default console-based Test Reporter is pretty verbose. Creating a new one is just as simple as implementing the `com.denodo.connect.testing.reporter.ITestReporter` interface and adding it to the configuration, see the `reporter` property in the **Configuration file section**. Alternatively, instead of implementing this whole interface users might prefer to extend `com.denodo.connect.testing.reporter.AbstractTestReporter` in order to override only the event handler methods of interest, as this abstract class implements the whole interface with empty methods.

It is available the option of reporting test events using multiple Test Reporter implementations, see the `reporter` property in the **Configuration file section**. If you use this feature with particular properties for a Test Reporter implementation then you have to declare the variable with the given name in the properties file and its setter method in the Test Reporter implementation.

6.1 CONSOLE TEST REPORTER

This test reporter, `ConsoleTestReporter`, outputs the execution information using the console. See the **Test Results section**.

6.2 CSV TEST REPORTER

The CSV test reporter uses the `CSVTestReporter` as the Test Reporter and it is in charge of reporting the results of each test execution in a CSV file. The path to this output file may be specified in the configuration file as well as a boolean indicating if you want to overwrite the file if it already exists. See **CSV reporter configuration section**.

6.3 EMAIL TEST REPORTER

The email-based Test Reporter, `EmailTestReporter`, sends the report of the execution by email. This reporter has three levels of detail that are configurable (see **Email reporter configuration section**):

1. `Full` level. A complete report will be sent with information of each test execution.
2. `Summary` level. A summarized report will be sent showing detailed information only for failed tests.
3. `Failed` level. A summarized report, as in summary level, will be sent when there have been any failed test during the execution.

6.4 HTML TEST REPORTER

The HTML Test Reporter, `HTMLTestReporter`, creates an HTML file with the report of the execution. The path to this output file may be specified in the configuration file as well as a boolean indicating if you want to overwrite the file if it already exists. If it exists and the overwriting value is false, an exception will be raised. See **HTML reporter configuration section**.

7 DEBUGGING

The Denodo Testing Tool uses Apache log4j to generate a log file of its execution. The log is located in the installation folder, under `logs/testing-tool.log`. This is the first place to look when an error takes place in the tests execution.

The log4j configuration can be modified, changing the logging levels, editing the file `conf/log4j.properties`:

```
log4j.rootLogger = WARN, file

log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File = ../log/testing-tool.log
log4j.appender.file.MaxFileSize = 10MB
log4j.appender.file.MaxBackupIndex = 10
log4j.appender.file.layout = org.apache.log4j.PatternLayout
```

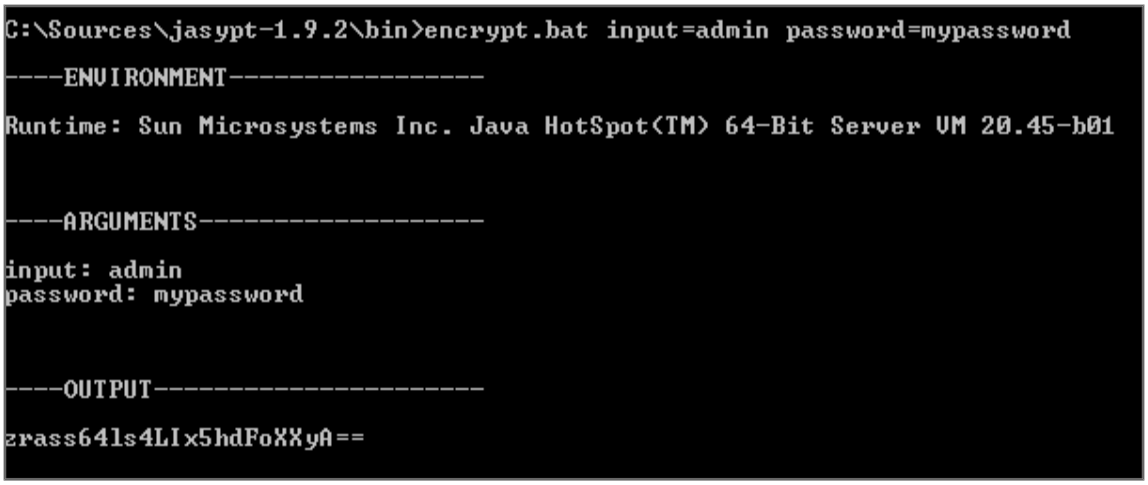
```
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n

log4j.logger.com.denodo.connect.testing = DEBUG

log4j.logger.com.denodo.vdb = FATAL
```

8 HOW TO ENCRYPT PASSWORDS

These are the steps for encrypting passwords in the configuration file of the Testing Tool:

1. Download Jasypt CLI tools.
2. Choose an encryption password, e.g., `mypassword`.
3. Go to `jasypt/bin`.
4. Run `encrypt.bat` with the `input` parameter and `password` parameter:
   `input` parameter - this is the string you want to encrypt.
   `password` parameter - this is the password that Jasypt is going to use to encrypt and decrypt the input parameter.

   Your command should look like this:



   Take note of the output. Example output: `zrass64ls4LIx5hdFoXXyA==`.

5. Open your `configuration.properties` file, replace the password you want to encrypt with `ENC(zrass64ls4LIx5hdFoXXyA==)`.
   Instead of `zrass64ls4LIx5hdFoXXyA==` you should use the output from Step 4.

```
# Before Jasypt
vdp.password=admin
# After Jasypt
vdp.password=ENC(zrass64ls4LIx5hdFoXXyA==)
```

6. Add an environment variable, `DENODO_TEST_ENCRYPTION_PASSWORD`, with a value of `mypassword`, but use your real encryption password.

7. Run your tests.

9 TROUBLESHOOTING

Symptom

Error message: *"driverClassName specified class '<driver class name>' could not be loaded. Caused by: java.lang.ClassNotFoundException: <driver class name>."*

Resolution

The Denodo Testing Tool only distributes the VDP JDBC driver for Denodo Platform 5.5 and Denodo Platform 6.0. They are in denodo-5.5.0_9 folder and denodo-6.0.8 folder, respectively, and you can find them in `<TESTING_TOOL_HOME>/drivers`. If your tests connect to databases other than VDP you will have to download the JDBC driver from the corresponding website or repository and choose one of these options:

- Copy the jar file to a folder in `<TESTING_TOOL_HOME>/drivers` and use the `dbAdapter` property to indicate the name of that folder. See **Configuration file section**.
- Copy the jar file to `<TESTING_TOOL_HOME>/lib`
- Add the JDBC driver path to the `DENODO_TEST_CLASSPATH` environment variable

These three options are mutually exclusive. When you choose to use the driver from `<TESTING_TOOL_HOME>/drivers` using the `dpAdapter` property it cannot be in `<TESTING_TOOL_HOME>/lib` and the JDBC driver path must not be added to the `DENODO_TEST_CLASSPATH` environment variable.

Symptom

Error message: *"Fail-fast during pool initialization. Caused by: java.sql.SQLException: connection error: unrecognized method hash: method not supported by remote object."*

Resolution

An VDP JDBC driver from a latter version is being used to access an earlier Denodo server. E.g.

JDBC driver from VDP server 5.5 update 2 - Connect to VDP server 5.5 update 1 => **not supported**.

10 APPENDIX: BENCHMARKING

Please note that using the Denodo Testing Tool for **benchmarking is not recommended** due to intrinsic constraints on the architecture and capabilities of a tool of this kind.

The Testing Tool does report execution times for the `%EXECUTION` directives of its tests, it does so in order to provide a way to quickly identify possible problems in the execution of specific tests, especially when these are a part of a very large automated test suites. But these execution times are not recommended for benchmarking use.

A purpose-specific tool such as Apache JMeter should be used instead for accurate benchmarking. It is an application specifically designed to load test functional behavior and measure performance.