Installation and users Guide for the reference implementation

Licence and Credits

Introduction

The reference implementation serves the following goals:

- Test and prove the concept of the DVPD
- Provide reference about the transformation ruleset for other implementations
- Provide examples for model and mapping variation
- Provide examples / templates for code and documentation generators

It is implemented completely in python and provided under the Apache 2.0 licence:
http://www.apache.org/licenses/LICENSE-2.0

Architecture and content of the reference implementation

The reference implementation is part of the DVPD git repository. It consists of the following assets:

- DVPD Compiler (DVPDC) (processes/dvpdc/__main__.py)
- DVPI Crosscheck (dvpd_dvpi_crosscheck) (processes/dvpi_crosscheck/main.py)
- Some assisting libraries (lib/*.py)
- Set of dvpd testcases and example dvpd (testcases_and_examples/dvpd/*.dvpd, testcases_and_examples/model_profiles/*)
- Automated test of the DVPDC (processes/test_dvpdc/__main__.py)
- Reference results for the automated testing ((testcases_and_examples/reference/*.dvpd))
- Examples of
  - ddl Render script (processes/render_ddl/__main__.py)
  - documentation render scripts (processes/render_documentation /__main__.py) (processes/render_dev_sheet /__main__.py)

    - dvpd generator (processes/generate_basic_dvpd_from_db_table_pg/__main__.py)) -dbt generator script (processes/generate_dbt_models/__main__.py)
- easy to call windows batch scripts or bash shell scripts for all examples (commands/)

The scripts are using a central configuration file (dvpdc.ini file), to declare further directory structures (See "Decisions about file locations" below)

- template for the configuration file (config_template/dvpdc.ini)

Installation Guide

Requirements

- python 3.10 or higher must be available
- please install missing python packages on demand (via python pip or equivalent package manager)
- A text editor (hopefully capable of JSON syntax highlighting and hierarchy folding)
- some knowledge about shell/batch file scripting and operating system behavior

If you want to modify, debug or extend the dvpd toolset or documentation

- A python ide
- A text editor supporting Markdown documents
- "Draw.io" for optimal view of documentation diagrams

Directory and file locations

It is recommended keep the dvpd scripts separate from the files of the dwh project (dvpds, ddls, loading code etc.) The dvpd scripts read the configuration files from the current directory, if not told otherwise. The main purpose of the configuration file is the declaration of all directories, the scripts search or write to.

- DVPD files
- "model profile" configuration files
- compiler results files (Reports, dvpi files)
- results from generators (e.g. the generated DDL files, documentation)

The dvpd files, and all the result, except for the compiler logs, should be under version control of the dwh project. It is up to you, if you generate directly into your "hot" project directories or if you use some intermediate directories as build targets and copy files later. Just be careful when building and committing automatically, since all dvpd interact on the level of the data vault model. A mistake in one dvpd can be the cause of a bad ddl file, was already created correctly by another dvdp. Be sure to include the crosscheck step in your build, to detect and prevent these kind of problems.

A dwh project, using dvpd must have at least the following dedicated directories:

- directory for all dvpds

- directory for all model profiles (at least the default.model_profile.json)
- directory for all dvpis (result of dvpd compile)
- directory for generated outcome (e.g. DDL scripts)

Providing a project specific dvpd build setup

This is only one solution, how to provide a build setup, by just using the basics of the operating system of a workstation. When using CI/CD tools, there might be many more. Use this as a blueprint, for a minimal solution

A very simple way, to provide a project specific build environment, is to create a dedicated "build shell directory" with the project specific dvpdc.ini in that directory. To execute a build command for the project, you just open a shell/commandline in that directory and type in the command

Example of a directory structure

All in all this would be example structure (names can be chosen freely and are only suggestions)

```
\dvpd_compiler             <- the compiler project (this project)
     \commands
     \documentation
     \processes
     ...
\dwh_project_x             <- the git repository of your dwh project
     \dvpd_model_profiles  <- dvpd model profiles
     \dvpd                 <- dvpd files
     \dvpi                 <- dvpi files
     \datamodel            <- genrated DDL files
\dvpd_shell_4_project_x    <- a dedicated place for setting up the build scripts (contains dvpdc.in
     \dvpdc_report         <- log output of dvpdc
     \...                  <- intermediate output for dvpd generated files
```

Download and set up the compiler project for the first project

- install dvpd on your workstation
  - Download or clone the DVPD repository in the directory for the DVPD compiler
  - add the directory "/commands" from the compiler project to your path environment variable
- prepare your project
  - add directories for the model profiles and the dvpds to your projects asset directory tree
  - copy the file "default.model_profile.json" from "testset_and_examples\model_profiles" to your desired location for model profiles
  - adapt the default model profile to your project needs (see Reference_of_model_profile_syntax.md)
  - add a first dvpd file to your project (probably just a copy of t0020_simple_hub_sat.dvpd.json from the testset_and_examples\dvpd directory)
- create a build shell

- create an empty directory for the projects specific shell (the build shell directory for the project). This must be outside the DVPD compiler project.
- copy the file **dvpdc.ini** from the "config_template" directory of the build shell directory for the project
- adapt all properties in the **dvpdc.ini** to point to the desired directories of your projects asset directory tree

Test the setup

- open a command line in the build shell directory
- run `dvpdc -h` . This should show the help text of the compiler
- run `dvdpc t0020_simple_hub_sat.dvpd.json .` - This should write out its messages and success state to the console and a log file in the log output directory
- if compiling is successfull you should now have a dvpi file in the directory for dvpis
- with the dvpi run `dvpd_dvpi_crosscheck t0020_simple_hub_sat.dvpi.json`
- this should also provide a correct result
- run all other renderer and generators you need, to check if the configuration is correct

Choose and adapt build script

There are multiple build scripts provided in the command directory. The build script combine all steps to compile a dvpd and generate all wanted elements, in one call. (see below).

It is recommended, to copy the build script, that fits your need the most, into the build shell directory with a proper name, and align it with your requirements and project principles.

By using your adapted copy, an update of the build script blueprint in dvdp project, will not affect your build settings.

To work properly, the build script needs the environment variable **DVPDC_BASE** to point to the dvpd repository. In windows a convenient way to achieve this, is to create a batch script in the build shell directory (suggested name "open_build_shell.bat") with the following content

```
set DVPDC_BASE=<absolute path to your dvpd install directory>
set path=%DVPDC_BASE%\commands;%PATH%
cmd
```

Start the batch file, to open the shell in that directory.

Now you are ready to start generating assets with dpvds in your project. Since also the path is modified in that shell, it is assured, that the commands from the desired DVPD installation are used.

Setup another project to use the compiler and generator

For another project to use the compiler follow this checklist

- prepare your project
  - add directories for the model profiles and the dvpds to your projects asset directory tree
  - copy the file "default.model_profile.json" from "testset_and_examples\model_profiles" to your desired location for model profiles in the project
  - adapt the default model profile to your project needs (see Reference_of_model_profile_syntax.md)
  - add a first dvpd file to your project (probably just a copy of t0020_simple_hub_sat.dvpd.json from the testset_and_examples\dvpd directory)
- create a build shell
  - create an empty directory for the projects specific shell (the build shell directory for the project). This must be outside the DVPD compiler project.
  - copy the file **dvpdc.ini** from the "config_template" directory of the build shell directory for the project
  - adapt all properties in the **dvpdc.ini** to point to the desired directories of your projects asset directory tree
  - create a new open_build_shell.bat or copy it from the builds shell directory of the existing project into your new build shell directory
  - create a custom build script in your build shell by copying and adapting an appropriate build script template from the "commands" directory to the build shell directory
  - open the shell via your open_build_shell.bat
  - test the build script with the example dvpd file

Adapt the ddl generator

As with all generators, the ddl generator is only an example/template and might not be sufficient for your specific database or project conventions.

Please be aware, that many design decisions about your platform (e.g. names and types of metadata and hash columns) can be adjusted in the model profile and will already be applied to the compilers result in the dvpi. Only the final formatting, sorting, ghost record creation and SQL Syntax of the ddl files lies in the responsibility of the ddl generator.

To adapt the ddl generator it is recommended to copy the template and make it your own code. How to integrate this into the build pipeline is beyond this installation guide.

Users Guides for all commands

dvpd compiler (dvpdc)

The dvpdc compiler is started on the command line with

```
dvdpc <name of the dvpd file> options
```

When using the file name "@youngest", the compiler uses the youngest file in the dvpd default directory

Options:

- --ini_file=<path of ini file>:Defines the ini file to use (default is dvpdc.ini in the local directory)
- --model_profile_directory=<directory>: Sets the location of the model profile directory (instead of the location configured by the ini file)
- --dvpi_directory=<directory>: Sets the location for the output of the dvpi file (instead of the location configured by the ini file)
- --report_directory=<directory>: Sets the location for the log and report file (instead of the location configured by the ini file)
- --verbose: prints some internal progress messages to the console
- --print_brain: prints the internal "memory" of the compiler for diagnosis

result

The compiler creates a log file and, when compilation is successfull, 2 result files

- report directory
  - log file: Contains the same messages, that have been written to the console
  - dvpisum.txt: Contains a summarized version of the dvpi data for fast overview about all essentials, created by the compiler
- dvpi directory:
  - dvpi file: Contains the dvpi json file, that has been generated from the dvpd. This can be used as a base for all further generators (see Reference_and_usage_of_dvpi_syntax.md)

Crosscheck of DVPI

The **Crosscheck Feature** is a component of DVPD reference implementation. It ensures data integrity, schema alignment and configuration consistency by identifying differences across multiple pipelines.

The crosscheck identifies inconsistencies between different dvpi's (=compiled dvpd's) in by just compare the structure:

1. **Table Structures**: Schema differences like column types, sizes, and constraints.
2. **Hash Keys**: Variations in definitions and usage (mostly caused by different field mapping properties).
3. **Field Types**: Mismatched declarations and usage.

Execution

The crosscheck is started on the command line with:

```
dvpd_dvpi_crosscheck <name of dvpi file> [options]
```

The declared file is used as focus. Comparison will only be done on tables, that are defined in this file.

- When using the file name **"@youngest"**, the script uses the youngest file in the dvpi default directory as focus.

- When using the file name **"@all"**, the script compares all files in the dvpi default directory

Options:

- -h, --help show this help message and exit
- --ini_file=<path of ini file>:Defines the ini file to use (default is dvpdc.ini in the local directory)
- --tests_only restricts the dvpi to a hard coded set of files (Name Pattern "t120*.json"). In the set of reference tests, these test contain specific scenarios to test the crosscheck

Settings read from dvpdc.ini file, section "dvpdc":

- dvpi_default_directory - The directory where the crosscheck searches for the dvpi-files

Exit codes:

- 0: everything is fine
- 5: There are similarity warnings
- 8: there are conflicts
- 9: There are inconsistencies in the dvpi files, that prevent an analysis
- 1: something very bad happened

The crosscheck checks all *.json files, in the configured directory. If conflicts are detected, they will be reported to the console and the crosscheck exits with exit code 8.

The output contains 4 sections:

- Table name similarity analysis: lists table names that are nearly the same (Might indicate a typo)
- Table / Column property conflicts: lists for every table with a conflict a detailed description of every conflict
- Summary Report: Statistics about the analysis
- Summary report line: comprehensive statistics about the analys in one single string (thought to be used as commit message)

ddl generator (dvpd_ddl_render)

The ddl generator renders all create statements for a given dvpi file. Since this is an example and not core part of the dvpd concept, syntax and structure are also only example (mainly taken from a current project). To adapt this to your needs, you should copy and adapt the script.

The ddl generator is started on the command line with

```
dvdp_ddl_render <name of the dvpi file> options
```

When using the file name "@youngest", the script uses the youngest file in the dvpi default directory

Options:

- --ini_file=<path of ini file>:Defines the ini file to use (default is dvpdc.ini in the local directory)
- --print: prints the full ddl set to the console

- --add_ghost_records: adds ghost record inserts to the script
- --no_primary_keys: omit rendering of primary key constraints
- --ddl_file_naming_pattern: declares the pattern for the file name. Valid settings:
  `lll` = everything lowercase
  `lUl` = table name upper case
- --stage_column_naming_rule={stage|combined} : stage = pure generated stage column names are used in the stage combined= combination of target column names and stage column names

Settings read from .ini file:

- dvpi_default_directory
- ddl_root_directory
- stage_column_naming_rule

Purpose of the "combined" stage table generation rule

Some data vault loading frameworks (e.g. cimt talend framework) use similarity of column names between stage table and target table for automatic mapping. In that case it is more convenient to generate a stage table with the target column names. This will work well, until the same column name for different content is used in different tables of the pipeline or different source fields are mapped to the same target.

The "combined" stage column naming rule resolves this as follows:

- one or more target columns with same name and same single source field: stage column name = Target name
- one target column with a unique name and multiple source fields: stage column name = Target name + field name
- multiple target columns with different names have the same single source field: stage column name = all target field names concatenated
- multiple target columns sharing the same name using different source fields: stage column name = field name

Documentation generator (dvdp_doc_render)

The documentation generator creates a simple html table of the field mapping, ready to be copied into a documentation tool of your choice (confluence, one Note / Word, ...)

The documentation generator is started on the command line with:

```
dvpd_doc_render <name of the dvpd file> options
```

When using the file name "@youngest", the script uses the youngest file in the dvpd default directory

Options:

- --ini_file=<path of ini file>:Defines the ini file to use (default is dvpdc.ini in the local directory)
- --print: prints the html text to the console

Developer sheet generator (dvpd_devsheet_render)

The developer sheet is a human-readable text file, providing the essential key information needed to implement the loading process.

- pipeline name
- record source
- source field structure
- formatted list of tables involved (targets + stage)
- stage table structure and mapping of fields to stage table
- hash columns and how to assemble it
- load operations for every table and the stage column to target column mapping

The developer sheet generator is started on the command line with:

```
dvpd_devsheet_render <name of the dvpi file> [options]
```

When using the file name "@youngest", the script uses the youngest file in the dvpi default directory

Options:

- --ini_file=<path of ini file>:Defines the ini file to use (default is dvpdc.ini in the local directory)
- --stage_column_naming_rule={stage|combined} : stage = pure generated stage column names are used in the stage combined= combination of target column names and stage column names

see "stage_column_naming_rule" in the ddl generator for explanation of the naming rules.

Settings read from .ini file:

- dvpi_default_directory
- stage_column_naming_rule

Generator for dvpd templates from database tables (dvpd_generate_from_db)

This command generates a template dvpd file from the table structure of a database tables.

For the given table, it determines all columns including their attribution to be part of the primary key or a foreign key. Also it measures some indicators about cardinality and completeness for all columns.

As a result it writes a dvpd file with all fields, a simple Data Vault model (Hub+sat/link/hub) and uses a best guess to map the fields to the model. The field analysis data is also provided in the dvpd. Additionally, it generates a summary of the data profiling as simple human-readable text file.

The dvpd generator is started on the command line with:

```
dvpd_generate_from_db <name of the table schema> <name of the table> [options]
```

options:

- -h, --help show this help message and exit
- --dvpdc_ini_file Name of the ini file of dvpdc
- --connection_ini_file Name of the ini file with the db connection properties of the source
- --connection_ini_section
  Name of the section of parameters in the connection file

Settings read from dvpdc.ini file:

- dvpd_generator_directory - Directory, the result file will be written to

The example is restricted to postgreSQL Databases. It reads its connection parameters from an ini file.

Generate DBT Models (generate_dbt_models)

This command generates DBT model files based on the DVPI files (=result of DVPD compile). The generated DBT models use the datavault4dbt syntax. Therefore, you need to install that package into the dbt environment to work.

The generator can be configured to write the model files into the dbt project directory. The target directory is configured with the ini parameter "model_directory". See all options below.

Currently, the script can generate hubs, links, satellites and stage models. Support for Ref-Tables will be added in upcoming releases.

The dbt model generator is started on the command line with:

```
dvpd_generate_dbt_models <name of the dvpi file> <path to the ini file> [options]
```

The generator creates or modifies all DBT models, that are affected by the declared dvpi file. By setting the dvpi file name to "@youngest", the script uses the youngest file in the dvpi default directory.

"Modify" means, that only the DBT declarations in the model file, that resulting from the given DVPI file will be added or changed by the generator.

By setting the option -a, the generator will replace the DBT models affected by the DVPI completely, but also includes the information of all other DVPI files that have an impact on these models.

When using the dvpi file name "@all", the script creates/replaces dbt models for all dvpi files found in the dvpi default directory.

The generator **does not check overall model consistency** and might deliver "garbage" if different DVPI's declare different structures for the same target. It is highly recommended, to check the consistency between of all DVPI files first with the "dvpd_crosscheck" script.

options:

- -h, --help show this help message and exit

- -a, --use-all-dvpis Use specified dvpi only to identify the set of models to generate, but use all dvpis to actually create those models.

Settings read from dvpdc.ini file, section "datavault4dbt":

- dvpi_default_directory - The directory where the generator searches for the dvpi-files
- model_directory - The directory where the datavault4dbt model files will be written to

DVPD usage Workflow

Here is a comprehensive guide of the general workflow, when using DVPD in different scenarios:

Manual steps

Primary dvpd creation and compilation

1. Generate and edit the dvpd document of the desired pipeline
2. store the dvpd in the directory for dvpd's in your project
3. Compile the dvpd. This will result in
    - a log output with compiler messages on console and the reporting directory
    - a dvpi file in the designated dvpi directory
    - a dvpi summary report in the reporting directory
4. review the compiler messages and the dvpi summary. If you need to correct mistakes, loop back to step 3
5. run the model crosscheck. In case of model inconsistencies to other pipeline, align the models (looping back to step 3)
6. probably commit dvpd, dvpi and the ddl files in a git repository of the project

Platform specific generation of database structure

1. when using DBT as load processor:
    i. run the dbt renderer
    ii. execute the dbt models, to test consistency
    iii. probably commit dvpd, dvpi and the dbt files in a git repository of the project
2. in case you need ddl scripts:
    i. run the ddl render script for the new generated dvpi to generate the ddl files in the repository for ddl files
    ii. deploy the data model to the database (using the ddl files and the deployment procedure of your choice). This will test the db compatibility of your ddl files.
    iii. probably commit dvpd, dvpi and the ddl files in a git repository of the project

In case of problems with generated code (mostly bad names and types, that are not compatible with the Databases) correct the dvpd and got back to step 3 in the previous phase

Further generation of code and documentation

1. generate developer sheet
2. generate documentation
3. generate loading code

Automation via full build scripts

To provide guidance and examples, there are multiple build scripts included in the command directory. These script **combine all necessary steps** to generate all assets from a single dvpd **in one call**.

Currently you can find the following scenarios:

- dvpd_build4basics - Our base script, only generating ddls and documentation
- dvpd_build4cimtpy - Also create code snippets for the cimt python data vault framework (work in progress)
- dvpd_build4dbt - Renders documentation and DBT models (ddl not needed since DBT does it by itself)

All buildscripts share the same parameters, conventions and behavior:

- dvdp scripts must end with ".dvpd.json"
- You must declare the dvpd file name, that shout be build, without ".dvpd.json"
- You can declare @youngest as file name to trigger the build of dvpd with the youngest change date
- The script stops in case of a compile error or a crosscheck conflict to prevent accidental overwriting of approved assets with wrong settings
- The script writes a final warning, when the crosscheck has detected name similarities. Assets will still be created.
- The script uses the DVPD script set, addressed by the environment variable DVPDC_BASE. If the variable is not set, the script assumes to have be positioned in the DVPD project command directory.