DVPD Reference Implementation Installation And Users Guide

Licence and Credits

Introduction

The reference implementation serves two major goals:

- Test and prove the concept of the DVPD
- Provide orientation and testsets for other implementations

It is implemented mostly with data base objects, since databases and expertise about it can be expected in every data warehouse project.

Architecture and content of the reference implementation

The reference implementation focuses on the interpretation of the DVPD. The implementation platform is a postgreSQL database extended by some python scripts for automatic deployment of the database objects. The following functions are provided:

- Loading and compiling of a DVPD
    - Loading and parsing the JSON into a relational Model (Transform)
    - Checking the DVPD structure about completeness and consistency (Check)
    - Compiling the DVPD to provide all information to generate the model and drive the fetch/load processes (main views and tables)
- Automatic test of the compiler implementation
    - Load reference data about the expected results of the compiler
    - compare compiler resultes with executed results
    - wide range of Testsets (in the directory datamodel/dv_pipeline_description/tests_and_demos)
- Automatic deployment of compiler and automatic tests
- cimt framework for job execution logging (This is a standard pattern at cimt, when implementing load processes, it is not required for the DVDP compiler itself)

All database objects reside in the **database schema "dv_pipeline_description"**.

Installation Guide

This project can be installed nearly automatically on a PostgreSQL Database by using the provided python scripts. **Knowledge about adminstration of postgreSQL and using Phython is required.**

Due to the huge number and the complex dependencies between the various database objects, it is recommended to use the automatic deployment. It is definitly required, when you want to prepare releases of the DVPD, since a full automatic deployment is a mandatory step before releasing.

*Note: all directories and files mentioned in the upcoming description are part of the git repository and declared from the root of the git repository.

prepare the database

Installation operations are using a database user which is owner of the target database or at least is allowed to create schemas on the database. If you don't alreday have a user and database to be used, create it as follows:

- connect to the postgres instance as admin (eg. user "postgres")
- create a user, that will own the database using the script "datamodel/database_creation/user_owner_data_vault.sql"
- create the database by adapting and executing the provided script "datamodel/database_creation/database_data_vault.sql"

configure python script environment

The python project needs some environment information for connecting to the database and retrieving the ddl scripts.

- create a copy of the directory "config_template" as "config"
- edit "config/basic.ini"
    - set "ddl_root_path" to the full path pointing to the "datamodel" directory of the repository. This depends on the location of the project.
- edit "config/pg_connect.ini"
    - adapt all connection parameters to your meet you DB configuration (DB, user, password)
- the following steps depend on your python environment/ide:
    - install psycopg2 module to your python environment
    - declare the repositories root directory as root for the search of python scripts. (e.g. add the full path to the environment variable PYTHONPATH)

Deploy the project to the database

- execute the python script "processes/jobless_deployment/__main__.py". This will deploy all objects listed in the files in datamodel/jobless_deployment in alphabetical order of the file names and the row order in the files.
- Check the end of the log output. The final summary should list only successfully deployed files
- in case of errors, you can target the deployment to a specific deployment file by adapting the __main__.py. Check out the commented examples at the bottom of the script.
- in the database open the view dvpd_atmtst_catalog. This should list more then 40 tests
- open the view dvpd_atmsts_issue_all. This should list only a tests with number 99 (Test, that has issues on purpose)

Procedures

Compile a DVPD and retrieve result

- insert the DVPD document into the table **dvpd_dictionary** (Example statements can be found in the Testsets) - this migh result in a format error, when the DVDP is no well formed json document.
- transform the DVDP document into the relational input structure of the compiler by executing "select dv_pipeline_description.**DVPD_LOAD_PIPELINE_TO_RAW**('#name of the pipeline#');"
- Open the view "**dvpd_check**" - If there are errormessages for your pipeline, adapt your DVPD accordingly and reload it. NOTE: Even with errors in the check, there will be compiler results available, but they are not valid.
- The compilers **results are available in the following views** (you need to filter for your pipeline name):
    - **dvpd_pipline_properties**: General properties of the pipeline (e.g. fetch and load module, name of the stage table)
    - **dvpd_pipeline_field_properties**: List of the fields and all properties needed for parsing the fields
    - **dvpd_pipeline_table** : Data Vault Tables defined for the pipeline
    - **dvpd_pipeline_column**: Columns (including type and other properties) of all data vault tables loaded by the pipeline
    - **dvpd_pipeline_table_driving_key**: Names of the driving key columns for every satellite table
    - **dvpd_pipeline_process_stage_to_dv_model_mapping**: Mapping of fields to stage and to data vault columns for everey process of every table of the pipeline
    - **dpvd_pipeline_stage_table_columns**: Stage table columns and their properties
    - **dvpd_pipeline_hash_input_field**: For every hash column in the stage table, the list of fields to concatenate + attributes to establish a proper order for concatenation

Add objects to the implementation

- Add a new script with an appropriate file name
- Add SQL DDL instructions to the script and test it
- Add necessary test cases to the test case catalog
- Add new script filename to the appropriate deployment list
- Add filenames of new test to the appropriate deployment list
- Delete object from the database and test the automatic deployment of the new object

Change objects of the implementation

- Change definition of the object in the appropriate script
- Test changes, and adapt/add test cases
- run automatic deployment to deploy all objects that might have been removed by cascading drop operations
- Check result of automated test

Prepare a new release

- Drop complete dv_pipeline_description schema from the database
- run full automated deployment
- Review content of the compliler check view: dvpd_check

- Review content of dvpd_atmtst_catalog
- Review content of dpvd_atmtst_issues

Implementation documentation

Database object hierarchy

For better understanding about the dependencies of the objects in the implementation please check out this diagram: Reference implementation object structure.drawio