Prenacs

Giorgio Gonnella & Ozan Kambertay

Jun 21, 2023

CONTENTS

CHAPTER

ONE

PRENACS

Prenacs is a system, which allows to run batch computations of attribute values for sets of entities, and to store the computation results in a database, alongside metadata which allows to track the data provenance.

The name is an acronym for "PRovenance-tracking ENtity Attribute Computation and Storage system".

1.1 Key concepts

Entities are thereby object of any kind, characterized by an identifier, which allows distinguishing the object from other objects of the same kind. For each entity, ProBatch allows to compute the values of attributes.

Attribute are properties of the entity. Anything whose value can be determined using a computation (in a general sense, thus e.g. also obtaining the value from an external data source) is an attribute. Each attribute may consist of a single value (*scalar attribute*) or or multiple values (*composite attribute*). The attribute system is open and flexible, i.e. new attributes can be added at any moment.

ProvBatch allows to store a *single instance* of each attribute for an entity: that is, it is not meant to store multiple measurements of a value or to store a journal of previous attribute computation results.

Alongside the computation results, ProBatch allows to track the *provenance* of attribute values. For achieving this, the code for attribute value computation must be implemented in form of a *computation plugin*. Multiple plugins may be available for computing the same attribute. A plugin may compute one or multiple attributes. Plugins contain the code for the attribute computation, as well as metadata, describing the input, output, supported paramaters and methods and implementation notes. Whenever a plugin code changes, a new *plugin version number* is assigned.

Batch computation metadata instances are identified by an unique identifier (UUID) and include a reference to a given plugin (plugin identifier and version number), alongside information such as the computation parameters, timestamps, identifier of the user starting the computation, and key system data. Thus, storing the batch computation UUIDs alongside with computation results, ProvBatch allows to keep track of the data provenance.

1.2 Related libraries

Prenacs has been developed as part of an ecosystem of Python libraries, including:

- *multiplug*, which implements the infrastructure for the plugin system
- *attrtables*, which implements the infrastructure for the storage of attribute values and metadata
- *snacli*, which is used for implementing double-purpose scripts, callable interactively from the command line, as well as inside a Snakemake pipeline

1.3 Usage

The usage of the library is explained in the user manual.

1.4 Acknowledgements

This software has been originally created in context of the DFG project GO 3192/1-1 “Automated characterization of microbial genomes and metagenomes by collection and verification of association rules”. The funders had no role in study design, data collection and analysis.

CHAPTER

TWO

PRENACS USER MANUAL

2.1 Implementing a plugin

In order to perform a computation, a *plugin* must be implemented. A plugin determines how one or multiple attributes of entities are computed. Furthermore, it contains metadata, which is stored alongside the computation results, in order to document the data provenance.

The plugin system is generic and implemented as a separate package, named `MultiPlug`. Plugins for Prenacs must implement a specific interface, which is described in the plugin implementation guide.

If a plugin which had been already used for computations changes, its version number must be incremented, so that a new plugin metadata record is created, when the new computation results are loaded into the database.

2.1.1 Checking the plugin implementation

In order to check if a plugin has been implemented correctly, it is possible to use the `prenacs-check-plugin` script. This loads the plugin module and analyses the exposed programming interface of the plugin, reporting any error to the user.

2.2 Database setup

The library was designed and tested using MariaDB. Other database management systems can be used, in which case some adjustments may be needed. The database must be already setup before the library can be used. The database server must be started.

The following parameters are passed to the scripts, which connect to the database:

- `--dbuser`: database user to use
- `--dbpass`: password of the database user
- `--dbname`: database name
- `--dbsocket`: connection socket file
- `--dbpfx`: database tables prefix

Before the first use, the script `prenacs-setup-database` must be run, which creates the necessary metadata tables.

2.3 Managing the attributes

Before running a computation, the attributes which are computed by the computation plugin must be added to the database.

For this a `prenacs-manage-attributes` script is provided. The attributes metadata must be described in a YAML file, which contains a dictionary, with one key for each attribute.

2.3.1 Attribute metadata

The attribute metadata files contains a dictionary where each key is a string - the attribute name - and each value is a dictionary - the attribute metadata.

The following keys must be defined in each attribute metadata:

- `definition`: a free text describing the attribute
- `datatype`: the datatype of the attribute.

Thereby, the datatype shall be expressed as described in the datatypes section of the attrtables package documentation

Furthermore, the following keys can be defined:

- `computation_group`: multiple attributes which are usually computed together by plugins shall contain a common unique computation group identifier in this field
- `ontology_xref`: link to an ontology term, which defines the same attribute
- `related_ontology_terms`: links to ontology terms, which do not directly describe the attribute, but are related to its definition
- `unit`: the measure unit for the attribute
- `remark`: a free text remark

2.3.2 Example

For example, a file `basic_seqstats.yaml` could contain:

```
gc_content:
  definition: fraction of the total bases of a sequence, which are G or C
  datatype: Float
  computation_group: basic_seqstats
seqlen:
  definition: seuqence length
  unit: bases
  datatype: Integer
  computation_group: basic_seqstats
```

The database can be prepared for storing the attributes defined in the file, using:

```
prenacs-manage-attributes <dbuser> <dbpass> <dbname> <dbsocket> \
                      basic_seqstats.yaml
```

2.3.3 Changing attribute metadata by constant datatype and computation group

If the `datatype` or `computation_group` of an attribute has not changed, the `prenacs-manage-attributes` script can be run using the `--update` option. This updates the attribute definition record.

For example, the metadata of GC content given in the previous section could be given slightly differently in a file `gc_content.yaml`:

```
gc_content:
  definition: portion of bases of a sequence, which are G or C
  datatype: Float
  computation_group: basic_seqstats
```

Since the `definition` changes, but not the `datatype` or `computation_group`, this metadata could be used for updating the `gc_content` attribute record, as follows:

```
prenacs-manage-attributes <dbuser> <dbpass> <dbname> <dbsocket> \
                        --update gc_content.yaml
```

2.3.4 Changing attribute metadata with different datatype and computation group

If the `datatype` or `computation_group` of an attribute has changed, the attribute must be dropped from the database, before adding it again. This means that the entire data for the attribute will be lost! This can be achieved by running the `prenacs-drop-attribute` script. After that, the attribute can be re-added by running `prenacs-manage-attributes` with the attribute definition file.

For example, the metadata of GC content given in the previous section could be given with a different `computation_group` value in a file `seq_composition_stats.yaml`:

```
gc_content:
  definition: fraction of the total bases of a sequence, which are G or C
  datatype: Float
  computation_group: seq_composition_stats
```

If we want to use this definition, the existing data for GC content must be destroyed, then the attribute definition is re-added, using:

```
# be careful: this deletes all the data for the attribute!
prenacs-drop-attribute <dbuser> <dbpass> <dbname> <dbsocket> gc_content
prenacs-manage-attributes <dbuser> <dbpass> <dbname> <dbsocket> \
                        seq_composition_stats.yaml
```

2.4 Batch computing

The `prenacs-batch-compute` script is used for running a computation, using a plugin. This assumes that the attributes computed by the plugin have been added to the database, as explained in the previous section. By default the computation is run in parallel, using the `mutliprocessing` module. If a serial computation is desired, the `--mode serial` option can be used. If desired, the computation can be run on a Slurm cluster (see below).

The batch compute function must locate the entities on which the computation shall be run. The input entities can be provided as a set of entity identifiers can be provided as a list or in a column of a tabular file. Alternatively a set of input files, one per entity, can be provided (specified using a globpattern). These different options are explained in the following sections.

For performance reasons, the computation results are not loaded directly into the database and instead, they are first saved to a tabular file. By default the results are output to the standard output, but the `--out` option can be used to specify a different file. The first column contains the entity IDs. The following columns are the results of the computation in the order specified by the plugin OUTPUT constant. Note that each of the attribute can in some cases include multiple columns.

Besides the attribute values, a computation report is output, which contains the computation metadata in YAML format. This file is necessary, in order to load the computation results into the database. The output is by default to the standar error, but a different file can be specified using the `--report` option. Additional metadata for the report can be provided using further options (see below).

Finally, a log file with additional information is output (by default to the standard error, but a different file can be specified using the `--log` option). This contains any further information about the computation returned by the plugin `compute` function. The output is tabular (tab-separated). The first column contains the the entity IDs. The following columns contains the information messages returned by the plugin. Each entity attribute computation can generate zero, one or multiple lines.

2.5 Running on a Slurm cluster

The computation can be run on a computer cluster managed by Slurm. For this to work the `prenacs-batch-compute` script must be called with the option `--mode slurm`.

The user must provide, using the option `--slurm-submitter <PATH_TO_SCRIPT>` the path to a Bash script which is passed to the `sbatch` command. An example is given in the library source code `prenacs/submit_array_job.sh`, and this file should be used as a template.

Temporary files are used to perform a computation on a Slurm cluster. By default, these files are created in the current directory and removed after the computation is done. The directory of these files can be specified with the option `--slurm-outdir`.

Slurm job arrays are utilized to perform computations for each task assigned to each input entity. If any of the tasks is failed, the ID of the relevant entity, the reason for the failure, and the task ID in the Slurm job array are written to a tab-separated file named `failed_tasks_{JOB_ID}.err`, where the JOB_ID corresponds to the Slurm job ID. If all tasks are failed, no such file is created, but the user is informed with a message. The status of the job array is checked and reported to the user every 15 seconds.

Even if some tasks are failed, the results for the remaining completed tasks are still collected and reported in the output file. The user can then attempt to compute the failed tasks again using the information provided in the file `failed_tasks_{JOB_ID}.err`. This can be especially useful for jobs that require a large amount of computation but have few failed tasks for some reason.

2.5.1 Input entities provided as a set of identifiers

If the input entities are specified as a set of entity IDs, the `ids` subcommand of `prenacs-batch-compute` is used. In this case the filename of the IDs file must be provided.

The file shall be either a list of entity IDs, with one ID per line, or a tabular file, in which the entity IDs are contained in a column. In the second case, the 1-based column number is also specified to `prenacs-batch-compute`.

The IDs are passed to the plugin `compute()` function as argument, and are used as first column of the output.

Computed entity IDs

It is possible also to use a list or tabular file is available, containing IDs from which the entity IDs can be computed, instead of the entity IDs themselves.

To achieve this, a module is created, which defines a `compute_id(str)->str` mapping function; the input of the function is the ID provided in the ID list and the output is the entity ID. The module filename is passed to `batch_compute.py` using the `idsproc` option.

2.5.2 Input entities provided as a set of input files

If the input entities are provided as a set of files, the `files` subcommand of `prenacs-batch-compute` is used.

The plugin `compute()` function of the plugin gets then the filename as first argument.

In most cases, it is desirable to compute the entity IDs which shall be used as a first column of the output. To achieve this, a module is created, which defines a `compute_id(str)->str` mapping function; the input of the function is the filename and the output is the entity ID to use in the output. The module filename is passed to `batch_compute.py` using the `idsproc` option.

If no `idsproc` module is provided, the first column of the output contains the input filename instead of an entity ID.

2.5.3 Incremental computations

It is possible to automatically avoid the recomputation for entities, for which a value has already been computed previously.

If a filename is provided using the `--skip` option, the entity IDs of previous computations are read from the first column of this tabular file. If the ID of an entity in the input (eventually after applying `idsproc`) is equal to one of the entity IDs of previous computations, then the computation is skipped.

If the `--out` option is used and the output file already exists, the existing output file is also used as `--skip` file.

2.5.4 Computation parameters

The computation parameters can be provided to the plugin as a YAML file, using the `--params` option.

The parameters are passed to the `compute()` function as keyword arguments. The type and meaning of the parameters must be described in the `PARAMETERS` constant of the plugin.

The computation parameters are assumed to remain constant during the computation, i.e. the same value is seen by each instance of `compute()`.

If a parameter is called `state`, it is handled in special way (see next section). For this parameter, it is not assumed that the value remains constant, and no description is necessary in the `PARAMETERS` plugin metadata constant.

2.5.5 Computation state

Differently from the computation parameters, a computation state can be defined. which possibly changes during the computation. When handling the computation state, it shall be thereby considered that the computation is run by default in parallel, using multiple processes.

The state is created by defining an`initialize()` function in the plugin. The function is run at the beginning of the batch computation and its return value of `initialize()` is passed as keyword parameter `state` to the `compute()` function. If necessary, a `finalize()` function can be defined by the plugin, which can perform teardown operations at the end of the batch computation.

State initialization parameters

It is possible to provide parameters to the initialization function. For this, a parameter in the file passed using the `--params` option shall be named `state`. This is handled in a special way: it is assumed to be a dictionary with string keys. Its content is passed to the `initialize()` function as keyword arguments. The value of `state` is then overwritten by the return value of `initialize()`.

2.5.6 Report file metadata

The report file contains the username of the user starting the computation. By default, this is computed using `getpass.getuser()`. However a different value can be provided using the `--user` option.

Furthermore, the name of the system on which the computation is run is added to the report. By default, this is computed using `socket.gethostname()`. However, a different value can be provided using the `--system` option.

Optionally, the reason for the computation can be stated and added to the report. The valid values for this are contained in the `REASONS` constants of the `Report` class of the `report.py` module and are:

- `new_entities`: the computation was run on entities, which did not exist yet in the database
- `new_attributes`: the computation was run on entities, which already existed in the database, in order to compute new attributes
- `recompute`: the computation was run on entities, which already existed in the database, for attributes for which a value was already computed before

2.6 Loading the computation results

In order to load the results of the computation into the database, the `prenacs-load-results` script is used. To it the output files of `prenacs-batch-compute` (results and computation report) are passed.

The same plugin used for the batch computing must also be provided, so that the plugin metadata can be stored in the database.

CHAPTER

THREE

PROVBATCH: PLUGIN IMPLEMENTATION GUIDE

This document describes the interface of plugins for ProvBatch.

Plugins are used for computing *attributes*, i.e. values associated to each *entity*. Each plugin computes one or multiple attributes. Each attribute can consist in a single value or a group of values (e.g. a computation result and a score).

Plugins can be written in Python, Nim, Rust and Bash. The plugins system is implemented as a separated package (MultiPlug).

3.1 Public Interface of ProvBatch plugins

A plugin must provide the following attributes:

- `ID`, `VERSION`, `INPUT`, `OUTPUT`: constants describing the plugin itself, the data returned by it and the resources necessary for the computation
- `compute()`: a function which computes the value of the attribute(s) for a given entity; it may accept computation parameters

Furthermore, a plugin may provide additiional attributes:

- `METHOD`, `IMPLEMENTATION`, `REQ_SOFTWARE`, `REQ_HARDWARE`, `ADVICE`, `PARAMETERS`: further metadata constants for plugin description
- `initialize()` and `finalize()`: functions for setting up and deleting resources used for an entire batch computation (e.g. library or database initializations); these functions are called only once (at the beginning and end of the batch computation), while `compute` is callled for each entity of the batch

3.1.1 Compute function

The plugin shall export the `compute(entity, **kwargs)` function:

- `entity`: identifier of the input data, or name of the file with the input data
- `kwargs`: additional optional named parameters (described in the PARAMETERS constant, see "Metadata constants section" below); note the special role of the `state` keyword argument if initialize() is defined, see "Shared resources for batch computations" below)

The return value of the method is a 2-tuple `(results, logs)` where:

- `results`: list of values, in the order specified by the `OUTPUT` constant (see below),
- `logs`: a possibly empty strings list, containing messages to be displayed to the user or stored in a log file; suggested format: "{key}\t{message}", where key identifies the type of log message.

3.1.2 Metadata constants

The plugin communicates its purpose, version and interface by defining the following constants.

Mandatory (their combination must be unique among all plugins):

- `ID`: name of the plugin [str, max len 256]
- `VERSION`: a version number or name [str, max len 64]

Mandatory:

- `INPUT`: short description of the input (type of data, format, source) [str, max len 512]
- `OUTPUT`: list of strings (attribute ids), describing the order or the attributes in the output; each of them must be defined in `attribute_definitions.yaml`.

Required if compute() accepts named parameters:

- `PARAMETERS`: optional named parameters of the compute function; list of 4-tuples of strings: (name, datatype, default_value, documentation)

Optional: (string, max len 4096)

- `METHOD`: how the results are computed, conceptually
- `IMPLEMENTATION`: how the method is implemented, technically
- `REQ_SOFTWARE`: required tools or libraries
- `REQ_HARDWARE`: required hardware resources (memory, GPUs…)
- `ADVICE`: when should this method used instead of others

3.1.3 Common resources for batch computations

Sometimes common resources are needed by multiple instances of a batch computation For example, it could be necessary to parse data files, to load some data into memory, or to initialize a connection to a database. Or some statistics could be collected during the computation and output at the end.

The convention in Prenacs is that such resources are passed around (if they exist) as a variable called `state`. Due to the dynamic nature of Python variables, this can be a single resource or a container (e.g. a dictionary) to access multiple heterogenous resources.

Creating common resources

There are different ways to create resources which are accessed by all instances of the batch computation.

First, the (initial) value of the `state` variable can be passed as one of the parameters (called `state`) of the batch computation.

Second, an initialization function can be implemented, with the signature `initialize(**kwargs)`. If this is provided, the function is called before the first instance of the batch computation. The batch computation parameters are passed to it as keyword arguments. The return value of the `initialize` function is passed to the `compute` function as a keyword argument called `state`.

Third, these two approaches can be combined: i.e. a value for a `state` variable can be specified in the batch computation parameters. If an `initialize` function is provided, this reads the `state` variable, along with any other parameter. The original value of `state` is then overwritten with the return value of the `initialize` function and can be accessed by the `compute` functions.

Accessing common resources

The common resources are accessed by the plugins `compute` function using a keyword parameter called `state`. Thus if common resources are needed, the function will take the signature: `compute(entity, state=None, **kwargs)`.

While accessing the common information (in particular when this is not read-only), it should be considered, that the computation can be started in parallel (this is the default mode in the prenacs-batch-compute script).

Finalizing common resources

In some cases, it is necessary to do finalization operations at the end of the batch computation - e.g. close some files or connections, or write some information, collected during the computation. In such cases a `finalize(state)` function can be implemented in the plugin. It takes the final value of the `state` variable, and it is executed after the last instance of `compute`.

3.2 Non-Python plugins

For details on how to implement plugins in Nim, Rust and Bash, please refer to the MultiPlug package user manual. This section provides some additional information which applies specifically to ProvBatch plugins.

3.2.1 Nim plugins

The compute function has the following signature, if there are no optional parameters:

```
proc compute(entity: string):
             tuple[results: seq[string]], logs: seq[string]] {.exportpy.}
```

Any accepted optional parameter must be defined in the signature. E.g. given two optional named parameters p1 of type t1 and default value d1 and p2 of type t2 and default value d2, the signature becomes:

```
proc compute(entity: string, p1: t1 = d1, p2: t2 = d2):
             tuple[results: seq[string]], logs: seq[string]] {.exportpy.}
```

Shared resources can be e.g. stored in a ref object, which is then passed back and forth to Python:

```
type
  PluginState = ref object of RootObj
    state: int

# initialize returns the state
proc initialize(): PluginState {.exportpy.} = PluginState(state = 0)
# it can have optional keyword arguments
proc initialize(s: int = 0): PluginState {.exportpy.} = PluginState(state = s)

# if initialize is provided, then compute must have a mandatory `state`
#   keyword argument of the same type as the return value of initialize
proc compute(filename: string, state: PluginState, ...

# finalize is optional and takes the state as only argument
proc finalize(s: PluginState) {.exportpy.} = discard
```

3.2.2 Rust plugins

The interface of the compute function when written in Rust is:

```rust
#[pyfunction] fn compute(filename: &str) -> PyResult<(...)> { ... Ok(...)
```

CHAPTER

FOUR

MULTIPLUG

The MultiPlug library is used for creating flexible plugin systems for Python programs, supporting multiple programming languages.

For example, a CLI script based on this library can take the name of a module as one of its command line arguments and import and use the module. The user of the script will then select a plugin e.g. from a plugin collection or writing it according to a given API specification.

4.1 Features

Main features of the library:

- the library allows the dynamic import of a plugin module, given its filename
- plugins can be written in Python, Nim, Rust and Bash
- the calling code is (in most cases) independent of the plugin implementation language
- plugin modules are automatically (re-)compiled, when necessary (thanks to the *nimporter* and *maturin* libraries)
- a wrapper mechanism allows to support Bash plugins
- systems are provided for defining module-level constants when importing Nim and Rust modules
- basic aspects of the plugin interface (names of the required and optional module-level public functions and constants) can be specified and automatically checked

4.2 Installation

The Python libraries listed in `requirements.txt` are installed automatically, if MultiPlug is installed using `pip`, and can otherwise be installed using `pip install -r requirements.txt`.

For supporting plugins written in Nim, the Nim compiler must be installed in the system and the *nimpy* library installed, e.g. using `nimble install nimpy`. Furthermore, for using the `exportpy_consts` macro in Nim plugins, run `nimble install` in the `multiplug_nim` directory.

For supporting plugins written in Rust, the Rust compiler must be installed in the system and the *PyO3* library installed, e.g. using `cargo install PyO3`.

4.3 Usage

The usage of the library is explained in the user manual.

CHAPTER

FIVE

MULTIPLUG: USAGE MANUAL

5.1 Importing a plugin module

To import a plugin module, the function `multiplug.importer(filename)` is used. The function automatically determines the plugin implementation language from the file extension (`.rs` for Rust, `.sh` for Bash, `.nim` for Nim, `.py` for Python). To load a plugin with a specific language only, use one of the functions `bash(filename)`, `rust(filename)`, `py(filename)` or `nim(filename)`.

Additionally to the filename the following named parameters are available:

- `verbose`: output verbose messages
- `disable_bash` (`importer()` function): if set, only Rust, Python and Nim plugins are supported
- `req_const`, `req_func`: list of names of constants and functions, which *must* be provided by the module; an exception is raised otherwise
- `opt_const`, `opt_func`: list of names of constants and functions, which *may* be provided by the module and are set to `None` if not provided
- `nim_const_pfx`: prefix to use in the Nim constants export system (default: `py_const_`)
- `rust_const_cls`: name of the class to use in the Rust constants export system (defailt: `Constants`)

5.2 Bash plugins limitations

For plugins written in Bash there are some limitations:

- the return type of the functions, and the constants type is limited to:
  - strings
  - lists of strings
  - lists of lists of string
- the function arguments are not checked, i.e. the wrapped function interface is always `(*args, **kwargs)`
- the script code is sourced multiple times in order to collect information about the defined functions and constants

If this limitations are undesired in an application, bash plugins can be disabled by setting `disable_bash`.

5.3 Plugin Functions

In Python plugins, functions are just defined as in any Python module. In Nim plugins, the `{.exportpy.}` pragma from the `nimpy` Nim library is used. Consult the `nimpy` documentation for more information. In Rust plugins, the `PyO3` library is used. Consult the library documentation for more information. In Bash plugins, functions are defined using some conventions, described below, which allow to automatically create their Python wrappers.

5.3.1 Rust plugin dependencies

Since the `Cargo.toml` for the plugin is autogenerated, dependencies other than pyo3 must be listed in the plugin code file, using the syntax:

```
// [dependencies] ..."
```

where `...` is what shall be writte in the `dependencies` section of `Cargo.toml`.

5.3.2 Bash functions arguments

All exported functions defined in Bash plugins are wrapped in a function which accepts any number of arguments and keyword arguments (`(**args, **kwargs)`).

The Bash function, however, may only accept a defined number of positional arguments. An eval-based system is used for passing the keyword arguments, e.g.:

```
function foo() {
  arg1=$1; shift; arg2=$1; shift # shift after each positional argument

  # the remaining arguments are the keyword arguments
  kwargs=$*; for kw in $kwargs; do eval $kw; done
  ...
}
```

5.3.3 Bash function return value

To return a value from Bash, a string is printed using "echo". For that reason function shall not print anything else than the return value (however, it may use the standard error freely).

If the returned string contains multiple lines or tabs, the wrapper splits it into an list of strings. If both newlines and tabs are returned, the string is splitted first by newline, then by tabs, and a list of lists of strings is returned by the wrapper. Examples:

```
  echo "123" # ==> "123" is returned (string)
  echo "1/t2" # ==> ["1", "2"] is returned (list)
  echo "1/n2/t3" # => [["1"], ["2", "3"]] is returned (nested list)
```

5.4 Constants

Actually, Python does not really have constants. However, by convention, variables names written in `UPPER_CASE` are considered constants.

Module-level constants are useful e.g. for defining special values, for defining hard-coded module parameters, or for storing metadata (for example, a plugin version number, or a description of possible plugin parameters).

In Python plugins, the constants are simply defined in the module code. For the other languages, some conventions are described below.

5.4.1 Bash

Constants are defined as variables in the script code. Only variables whose name does not start with an underscore are imported.

Scalar variables are imported as string constants. Arrays are imported as lists of strings. If any element of the array contain a tab, each of the elements is splitted by tab, so that the constant value is a list of lists.

Examples:

```
STRING_FOO1="bar1"
LIST_FOO2=( "bar21" "bar2 2" )
NESTED_FOO3=( "bar311"
              "bar3 21\tbar322" )
```

5.4.2 Nim

Since the `nimpy/nimporter` system does not allow to export Nim constants directly to Python, a workaround is used instead. The constants are defined as the return value of functions (which are exportable to Python).

The easiest way to import constants is to install the nimble package `multiplug_nim` (distributed with the source code of Multiplug) and use the macro `exportpy_consts()`, e.g.:

```
import multiplug_nim/exportpy_consts
const
  FOO="foo"
  BAR="bar"
exportpy_consts(FOO, BAR)
```

Implementation details

For each constant, a proc with no arguments is defined in the Nim code, which returns the value of the constant. The proc name has a prefix (by default `py_const_` which allows to recognize it and is stripped in the definition of the Python module attribute. E.g.: `proc py_const_FOO(): string {.exportpy.} = "bar"` defines the attribute `FOO` with value `"bar"` in Python.

Using a custom prefix

The prefix used in the proc names (by default `py_const_`) can be changed by setting the `nim_const_pfx` parameter of the importer function. In this case in the Nim code, the macro `exportpy_consts_wpfx(pfx, ...)` is used in place of the `exportpy_consts()` macro, where the first parameter is the same prefix used in the `nim_const_pfx` parameter in the importer function in the Python code.

5.4.3 Rust

Since the `PyO3/maturin` system does not allow to export Rust module-level constants to Python, a workaround is used instead. The constants are defined in a struct called `Constants` and exported as Python class

For example:

```
#[pyclass] struct Constants {}
#[pymethods]
impl Constants { #[classattr] const FOO: &'static str = "bar"; }

...

#[pymodule]
fn my_module(_py: Python, m: &PyModule) -> PyResult<()> {
  m.add_class::<Constants>()?;
  ...
```

Using a custom class name

Instead of `Constants` a different class name can be specified, by setting the `rust_const_cls` keyword parameter in the importer function to a different string.

5.5 Persistant state between function calls

To implement a state which shall persist between function calls, the plugin shall implement an initialization function, creating the state, a finalization function, destroying the state (if necessary). The other function calls shall then take the state as an argument.

Examples of plugins using a persistant state in all supported programming languages are in the `tests/testplugins` directory.

For Python and Nim it is not particularly challenging to have an initialization function, which creates a new state object, and pass the state as an argument to other functions.

5.5.1 Rust

In Rust the state can be implemented as a `struct`, e.g. `struct State`. The initialization function will return a `PyResult<Py<State>>`. Thereby the state is created using `Py::new(py, State {...})`, where `py` is obtained using `Python::acquire_gil().python()`.

The state is passed then to other functions (including the finalization function, if any) as the `state: Py<State>` argument. To change the state `let mut state = state.borrow_mut(py)` can be used, where `py` is obtained as explained above.

5.5.2 Bash

In Bash the state can be implemented by storing it to file. The initialization function can create a state file using $(mktemp) and storing some information to the file, then returning the filename. Other functions then get the state filename as an argument.

For example the information can be stored in form of variable assignments (e.g. `x=1`). Functions other than the initialization get then the state filename as argument. The contents of the file can be executed using `eval $(cat $state_file)`. If the variable are modified, the previous state file can be overwritten with new variable assignment statements (e.g. `x=2`).

CHAPTER

SIX

ATTRTABLES

AttrTables is a library for storing attributes of entities in a database, where the entities are rows and the attributes are columns in an automatically managed set of tables.

6.1 Basic concepts

Each *entity* (identified by an unique ID) can have as many *attributes* as desired. These consist in one or multiple values, each stored in an independent database column. New attributes can be added at any time.

Alongside with the values, attributes can store a computation ID (which can refer to an external table of computation metadata). Computation IDs can be stored for individual attributes and optionally also for groups of attributes.

The attribute columns are automatically spread among multiple database tables, so that the total number of columns does not exceed a given limit.

6.2 Comparison to entity-attribute-value

An alternative to the model implemented by AttrTables is the entity-attribute-value (EAV) model, in which there is a single table, where entities are rows and the attribute name and value are two columns.

The EAV model has some disadvantages, compared to AttrTables:

- the values of a single attribute generally cannot be indexed
- the values must be stored in a generic data type (such as blobs), and the application must convert back and forth to the correct datatype

6.3 Setup

The library is based on SqlAlchemy, which must be installed (see `requirements.txt`).

Furthermore, a database must be setup. The connection to the database is done using SqlAlchemy, and the connectable is passed to the library, as explained below.

6.4 Running the tests suite

To run the test suite, a database is needed, where the test tables can be stored. The user must create such database.

The database configuration is provided using a YAML configuration file “config.yaml”, which shall be stored in the “tests” subdirectory (see as an example “tests/config.yaml”).

6.5 Usage

The usage of the library is explained in the user manual.

CHAPTER

SEVEN

ATTRTABLES: USAGE MANUAL

The main class of the library is `AttributeValueTables`, which represents the collection of tables, where the attributes are stored.

7.1 Creating the AttributeValueTables instance

An instance of the class is needed for all operations. The constructor requires, as a mandatory parameter, a SqlAlchemy *connectable*. The engine must be used with `future=True`.

For example:

```
engine = create_engine(connection_string, future=True)
connection = engine.connect()
avt = AttributeValueTables(connection)
```

Besides the connectable, further arguments can be passed to the constructor, explained below.

7.1.1 Computation IDs

By default, for each attribute, computation IDs can be stored alongside the attribute values, so that the computation metadata can be stored (e.g. in a different table).

The type of the computation ID column is by default `binary(16)` (which can be used for storing a UUID). It can be set to a different type by setting the `computation_id_type` argument. If the computation IDs are not needed, the `support_computation_ids` argument can be set to False.

When computation IDs are supported, computation groups can be defined, which group attributes together. If for an entity, all attributes of a group are computed at once, the computation ID of the group is stored instead of the single IDs, saving place (since all the computation IDs of the attributes will be set to NULL). To disable attribute computation groups set `support_computation_groups` to `False`.

7.1.2 Value tables

Attributes values are stored in multiple tables. The attributes are assigned automatically to tables, based on the number of columns required to store the values and computation ids.

The target number of columns per table is a parameter that can be set by setting the `target_n_columns` argument. The default is 64. Generally, the number of columns per table does not exceed this value. Only if a given attribute alone already requires a number of columns larger than this, the table for that attribute will have more columns that this value.

The names of the attribute value tables all have a common prefix, which can be passed as the `tablename_prefix` argument.

7.1.3 Attribute definitions table

A table for storing attribute names and database (and, optionally, attribute computation groups) is needed. A database model for such a class is provided (`AttributeDefinition`) and used by default by `AttributeValueTables`.

Optionally, the table can be defined using a different model (for example for defining further columns). In this case the model class is passed as the `attrdef_class` attribute to the `AttributeValueTables` constructor. The class must be a SqlAlchemy ORM model providing at least the attributes `name` and `datatypes`. If the computation groups are enabled (by default) also `computation_group` must be provided.

7.2 Creating attributes

Before values can be stored for an attribute, the attribute must be created.

This is done using the `create_attribute()` method of the `AttributeValueTables` instance.

The method has two mandatory arguments:

- `name`: must consist of letters, digits and underscores and may not start with a digit.
- `datatype`: a string describing the datatype

If computation groups are enabled (default), each attribute can be assigned to a computation group (not mandatory), by setting the `computation_group` argument.

If a different model for attribute definitions is used, which contain further columns, the values for these columns can be passed to `create_attribute()` using keyword arguments.

7.2.1 Datatype description

The datatype is described using SqlAlchemy column types (see e.g. https://docs.sqlalchemy.org/en/14/core/type_basics.html).

For example:

- `"Boolean"`
- `"Integer"`
- `"Float"`
- `"String(n)"` where n is an integer >= 1, e.g. `"String(50)"`
- `"Text"`

An attribute can consist in a single value, in which case the datatype is just a string containing such a column type name.

Furthermore, attributes can contain multiple values. If the values have different datatypes, they are joined using ";". For example:

- `"Boolean;Integer;String(50)"`

If an arary of values of the same type is desired, this can be specified, by adding a `[n]` suffix, e.g.

- `Integer[10]`
- `String(50)[10]`

These can be used also in combined definitions with ";", e.g.

- `Boolean;Integer[3];Float;String(50)[4]`

7.2.2 Attribute name reccomendations

Since in some systems the column names are case insensitive, it is recommended to use lower case letters.

The length of the name is limited by two factors:

- first, the length of the name column in the attribute definition table (by default 62).
- second, the maximum length of a column name in the database; it shall be remarked that a suffix is appended to the attribute name (e.g. `_v`):
  - for scalar attributes the suffixes have length 2;
  - for composed/array attributes, the suffix has length `2+ceil(log10(n_elements))` (e.g. 4 for 100 elements)

7.3 Setting values of an attribute

Once an attribute has been added, values of the attribute for a number of entities can be set using the `set_attribute()` method of the AttributeValueTables instance:

```
avt.set_attribute(attribute_name, values_dictionary, computation_id)
```

Thereby:

- `attribute_name` is the name of the attribute (which must have been already added, using `create_attribute()`, see above)
- `values_dictionary` is a dictionary of `{entity_id: values}`, where values is either a scalar (for scalar attributes) or a list of values (for compound/array attributes; it must have the correct size in that case)
- `computation_id` is not mandatory (and it may only be provided if support for computation IDs is not disabled)

7.3.1 Setting multiple attributes at once

If multiple attributes are computed at once, they can be set using the `set_attributes()` method of the AttributeValueTables instance:

```
avt.set_attributes(attribute_names, values_dictonary, computation_id)
```

Thereby the `attribute_names` is a list of names of attributes. The values dictionary and computation ID have the same meaning as when adding a single attribute. However, the lists of values in the entries of the `values_dicionary` must in this case contain one element for each of the columns of the attributes in `attribute_names`, in the correct order. E.g.

```
avt.create_attribute("a", "Integer[2]")
avt.create_attribute("b", "Float[2]")
avt.set_attributes(["a", "b"], {"entity1": [1, 2, 1.1, 2.2]})
```

7.3.2 Loading the results of a batch computation

For performance reasons, the results of a batch computation can be directly loaded from a tab-separated file. This is done using the `load_computation` method of the AttributeValueTables instance:

```
avt.load_computation(computation_id, attributes, inputfile)
```

Thereby a temporary table is created, the data is loaded to the table and then merged with the original tables (the temporary table name suffix is `temporary` and can be set to a different value using the keyword argument `tmpsfx`).

The inputfile must contain a number of columns and datatypes compatible with the list of attributes, e.g.

```
avt.create_attribute("a", "Integer[2]")
avt.create_attribute("b", "Float[2]")
avt.load_computation(computation_id1, ["a", "b"], "results_file.tsv")

# where results_file.tsv contains, e.g.:
entity1   1   2   1.1    2.2
entity2   2   3   2.3    3.1
```

7.4 Deleting an attribute value

To remove the value of an attribute for some entities, the `unset_attribute` method of the AttributeValueTables instance is used:

```
avt.unset_attribute(attribute_name, [list_of_entity_ids])
```

7.5 Querying the values of an attribute

To query the values of an attributes for a list of entities, the `query_attribute` method of the AttributeValueTables instance is used:

```
avt.query_attribute(attribute_name, [list_of_entity_ids])
```

To query the values for all entities, do not pass a list of entities as second argument:

```
avt.query_attribute(attribute_name)
```

The return value is a dictionary, with an entry for each entity of the list for which a value of the attribute exists. The value of the dictionary entry is a tuple `(attribute_value, computation_id)` if computation IDs are supported (default) or just `attribute_value` otherwise. Thereby `attribute_value` is a scalar, if the attribute is scalar, and is a tuple for compound/arrray attributes.

7.6 Destroying an attribute

To destroy an attribute the following method of the `AttributeValueTables` instance is used:

```
avt.destroy_attribute(attribute_name)
```

7.7 Listing the attributes

A list of the attributes is provided by the following property of the `AttributeValueTables` instance:

```
avt.attribute_names
```

CHAPTER

EIGHT

SNACLI

SnaCLI is a library which simplifies writing scripts which can be both run directly from the command line (based on docopt), as well as be called in a Snakemake file.

8.1 Introduction

In the *Snakemake* workflow management system, tasks can call Python scripts and pass them input, output and log filenames as well as global configuration values and non-filename parameters.

The system for passing and accessing these arguments is different to that adopted by command line tools (for example using the *docopt* library).

SnaCLI allows to easily combine the two approaches (docopt and snakemake) for providing arguments to a script. Script based on SnaCLI can be invoked both from the command line and from snakemake. Inside the script, the same code can be used in both cases, without modifications.

8.2 Usage

The usage of the library is explained in the user manual.

CHAPTER

NINE

BACKGROUND: SNAKEMAKE EXTERNAL PYTHON SCRIPTS

Snakemake (https://snakemake.github.io/) is a popular Python-based workflow management system. Tasks in Snakemake are defined as rules in *Snakefiles*.

Each rule may define input files, output files and non-file parameters. Rule definitions in the Snakefiles can call an external command using the shell, directly contain code to be executed (e.g. Python code) or invoke an external script (in different programming languages, including Python), to which the rule parameters are passed.

The following is an example of calling an external Python script:

```
rule a:
  input:
    inputfile1="i1", inputfile2="i2"
  output:
    outputfile1="o1", outputfile2="o2"
  params:
    f="oo", bar=True
  script:
    script(path/to/example.py)
```

Inside the script `example.py`, the input, output and params are accessible from the attributes `input`, `output` and `params` of the global variable `snakemake`.

CHAPTER

TEN

BACKGROUND: CLI SCRIPTS BASED ON DOCOPT

Docopt (http://docopt.org/) is a command line interface description language with implementations in different languages, including Python.

The syntax of the script and the available options are described in a string, which is passed to the `docopt()` function. For example:

```
Usage:
  test_script.py <inputfile1> <outputfile1> [INPUTFILE2] [options]

Options:
  -2, --outputfile2 FNAME   Output file 2 (default: stdout)
  -f VALUE                  Option with a value
  --bar                     Boolean option
```

The return value of the function is a dictionary, which contains the values of options and positional arguments. In this example it would contain the keys `"<inputfile1>"`, `"INPUTFILE2"`, `"--outputfile1"`, `"-x"`, `"--bar"`.

CHAPTER

ELEVEN

SNACLI

SnaCLI allows to easily combine the two approaches (docopt and snakemake) for providing arguments to a script. Therefore the script can be invoked both from the command line and from snakemake.

This is obtained by employing the `snacli.args()` context manager. Lists are passed to `args()` describing how to obtain from the snakemake rule the values of options and positional arguments described in the docopt string.

For the examples given above:

```
with snacli.args(input=["<inputfile1>", "INPUTFILE2"],
                 output=["<outputfile1>", "--outputfile2"],
                 params=["--bar", "-f"]) as args:
  # code which does something with args, as if they would
  # come from docopt, e.g.
  print(args["<inputfile>"])
```

The value yielded from the context manager is then always equivalent to the value returned by `docopt()`, both in the case that the script is invoked from the command line and that it is called from a Snakefile. Thus, the same code can be used in both cases, without modifications.

11.1 Mapping of docopt keys to snakemake names

Although Snakefiles support both named filenames (e.g. `input: input1="file1"`) and unnamed filenames (e.g. `input: "file1"`) in the input and output keys, for SnaCLI to work, all filenames must be named.

The docopt keys contain formatting: `UPCASE` or `<angular>` for positional arguments, an initial `-` or `--` for options. These cannot be used directly as names in the input, output and params of snakemake rules. Thus the `snacli.args()` maps each docopt key value to a snakemake name value, by stripping the initial `-` or `--`, removing the angular brackets and making upcase-only keys lowcase. If further `-` are present, they are replaced by an underscore.

Examples of docopt keys and the corresponding snakemake name:

- `<inputfile1>`: `inputfile1`
- `INPUTFILE2`: `inputfile2`
- `--param1`: `param1`
- `-x`: `x`
- `--long-name`: `long_name`

11.1.1 Customized docopt key to snakemake name mapping

It is possible to manually override the mapping of docopt keys to snakemake names by using, in the lists passed to `snacli.args()`, a 2-tuple `(docopt_key, snakemake_name)` instead of a string. Eg. to map `--param-2` to `param2` instead of `param_2`:

```
# in the snakefile:
rule foo:
  params: param2: "value"
  script: "foo.py"

# in foo.py:
with snacli.args(params=["--param1", ("--param-2", "param2")]) as args:
  print (args["--param-2"])
  ...
```

11.2 Passing options to docopt

By default, the docstring of the script (`__doc__`) is passed to `docopt()` as string. Another string can be used instead, by passing it to the keyword argument `doc`, e.g.

```
with snacli.args(doc=somestring, input= ...
```

Besides the lists of input, output and params, the `snacli.args()` also accepts keyword arguments which are passed to `docopt()`, i.e. `argv`, `help`, `version`, `options_first` – see the docopt documentation for their meaning, e.g.

```
with snacli.args(version="1.0", input= ...
```

11.3 Using a script also as a non-interactive module

Sometimes a script shall be also used as non-interactive, i.e. imported as module in another Python module.

In case the script is imported as module, the value yielded by the `snacli.args()` context manager will be `None`. Thus to support inclusion as a module, an additional `if` condition must be added, e.g.:

```
with snacli.args(...) as args:
  if args:
    ...
```

11.4 Reusing argument definitions in multiple scripts

If the same arguments are used in multiple scripts, they can be collected in a separate module and re-used.

For example, say that the optional arguments `--input1` and `--param2` are used in multiple scripts. Then the docstring of a script could be set to:

```
  Usage:
    foo [options]
```

(continues on next page)

(continued from previous page)

```
  Options:
    --specific    Specific option for this script only
    {}
```

The definition string for the options and the mapping of docopt strings to snakemake could be provided in a module `bar`, which can be reused also in other scripts, e.g.:

```
optstr='''
    --input1          Input filename 1
    --param2 VALUE    Value of parameter 2
'''

optmap = {"input": ["--input1"], "params": ["--param2"]}
```

Then in the script, SnaCLI can be used as follows:

```
import bar
with snacli.args(bar.optmap, docvars=[bar.optstr]),
                 params = ["--specific"]) as args:
  ...
```

I.e. the common mapping is passed as positional argument to `snacli.args`, before any keyword argument, and the docvars keyword argument is used, with the arguments which shall be passed to `format()` called on the docopt string.

This can be generalized over multiple re-usable modules, e.g.:

```
...
  Options:
    {bar_opts}
    {foo_opts}
...

import foo
import bar
with snacli.args(foo.optmap, bar.optmap,
                 docvars={"bar_opts": bar.optstr,
                          "foo_opts": foo.optstr},
                 params = ["--specific"]) as args:
  ...
```

11.4.1 Multiple entries for the same key

It is possible to override the name mapping for a key passed with a positional argument, using one of the following positional arguments, or in the keyword arguments.

In the example above, if foo.optmap contains `{inputs = ["--specific"]}`, the later setting in the keyword argument `params` would be applied instead, i.e. `specific` would be taken from `params` and not from `inputs`.

Instead, using the same key in different lists of the same positional argument, or in different keyword arguments is an error, leading to unspecified behaviour.