Starlake DAG generation

starlake dag generation allows you to generate *dag*(s) that will run your *job*(s) on a *schedule* using **Apache Airflow** or any other scheduler.

It relies on:

- **starlake** command line tool
- dag **configuration**(s) and their references within the loads and tasks
- **template(**s) that may be customized
- **starlake-airflow** *orchestration framework* to **dynamically** generate the tasks that will be run
- managing **task dependencies** to execute **transforms** in the **correct order**

- Prerequisites
- Command
- Configuration
  - References
    - DAG configuration for loading data
    - DAG configuration for transforming data
  - Properties
    - Comment
    - Template
    - Filename
    - Options
      - Default Airflow pool
      - Starlake env vars
      - Pre-load strategy
        - NONE
        - IMPORTED
        - PENDING
      - blocked URL
        - ACK
      - Load dependencies
      - Additional options
        - Bash
        - Dataproc
        - Cloud run
- Templates
  - Path
  - Starlake templates
    - Data loading
    - Data transformation
  - Customize existing templates
    - Transform parameters
      - User defined macros
    - Dataproc cluster configuration
    - Spark configuration
- Dependencies
  - Inline
  - Data-aware scheduling

Prerequisites

Before using **Starlake** dag generation, ensure the following minimum versions are installed on your system:

- **starlake**: 1.0.1-SNAPSHOT or higher
- **python**: 3.8 or higher
- **Apache Airflow**: 2.4.0 or higher (2.6.0 or higher is recommanded with *cloud-run*)
- **starlake-airflow**: 0.1.2.1 or higher

Command

```
starlake dag-generate [options]
```

where *options* are:

parameter	cardinality	description
--outputDir <value>	optional	Path for saving the resulting DAG file(s) (*${SL_ROOT}/metadata/dags/generated* by default).
--clean	optional	Wether to clean the resulting DAG file(s) before or not (*false* by default)

--domains	optional	Wether to generate DAG file(s) to load schema(s) or not (*true* by default if *--tasks* option has not been specified)
--tasks	optional	Whether to generate DAG file(s) for tasks or not (*true* by default if *--domains* option has not been specified)
--tags <value>	optional	Whether to generate DAG file(s) for the specified *tags* only (no tags by default)

Configuration

All DAG configuration files are located in *${SL_ROOT}/metadata/dags* directory. The root element is **dag**.

References

We reference a DAG configuration by using the configuration file name without its extension.

DAG configuration for loading data

The configuration files to use for *loading* data can be defined

- at the **project** level, in the **application** file *${SL_ROOT}/metadata/application.sl.yml* under the *application.dagRef.load* property. In this case the same configuration file will be used as the default DAG configuration for all the tables in the project.

```
application:
  dagRef:
    load: load_cloud_run_domain
#...
```

- at the **domain** level, in the **domain** configuration file *${SL_ROOT}/metadata/load/{domain}/_config.sl.yml* under the *load.metadata.dagRef* property. In this case the configuration file will be used as the default DAG configuration for all the tables in the domain.

```
load:
  metadata:
    dagRef:load_dataproc_domain
#...
```

- at the **table** level, in the **table** configuration file *${SL_ROOT}/metadata/load/{domain}/{table}.sl.yml* under the *table.metadata.dagRef* property. In this case the configuration file will be used as the default DAG configuration for the table only.

```
table:
  metadata:
    dagRef:load_bash_domain
#...
```

DAG configuration for transforming data

The configuration files to use for *transforming* data can be defined

- at the **project** level, in the **application** file *${SL_ROOT}/metadata/application.sl.yml* under the *application.dagRef.transform* property. In this case the same configuration file will be used as the default DAG configuration for all the transformations in the project.

```
application:
  dagRef:
    transform: norm_cloud_run_domain
#...
```

- at the **transformation** level, in the **transformation** configuration file *${SL_ROOT}/metadata/transform/{domain}/{transformation}.sl.yml* under the *task.dagRef* property. In this case the configuration file will be used as the default DAG configuration for the transformation only.

```
task:
  dagRef: agr_cloud_run_domain
#...
```

Properties

A DAG configuration defines four properties:

```
dag:
  comment: "dag for transforming tables for domain {{domain}} with cloud run" # will appear as a description of
the dag
  template: "gne_scheduled_task_cloud_run.py.j2" # the dag template to use
  filename: "{{squad}}/{{squad}}_{{domain}}_norm_cloud_run.py" # the relative path to the outputDir specified
as a parameter of the `dag-generate` command where the generated dag file will be copied
  options:
    sl_env_var: "{\"SL_ROOT\": \"${root_path}\", \"SL_DATASETS\": \"${root_path}/datasets\", \"SL_TIMEZONE\": \"
Europe/Paris\"}"

 #...
```

Comment

A short **description** to describe the generated DAG.

Template

The **path** to the template that will generate the DAG(s), either:

- an **absolute** path
- a **relative path** name to the *${SL_ROOT}metadata/dags/template* directory
- a **relative path** name to the *src/main/templates/dags* starlake resource directory

Filename

The filename defines the **relative path** to the DAG(s) that will be generated. The specified path is relative to the *outputDir* option that was specified on the command line (or its default value if not specified).

The value of this property may include **special variables** that will have a direct **impact** on the **number of dags** that will be generated:

- **domain**: a *single DAG* for all tables within the *domain* affected by this configuration

```
dag:
  filename: "{{squad}}/{{squad}}_{{domain}}_norm_cloud_run.py" # one DAG per domain
 #...
```

- **table** : *as many dags as* there are *tables* in the domain affected by this configuration

```
dag:
  filename: "{{squad}}/{{domain}}/{{squad}}_{{domain}}_{{table}}_norm_cloud_run.py" # one DAG per table
 #...
```

Otherwise, a single DAG will be generated for all tables affected by this configuration.

Options

This property allows you to pass a certain number of options to the template in the form of a *dictionary*.

Some of these **options** are **common** to all templates.

Default Airflow pool

default_pool option defines the Airflow pool to use for all tasks executed within the DAG

```
dag:
  options:
    default_pool: "${squad}_default_pool"
 #...
```

Starlake env vars

sl_en_var defines starlake environment variables passed as an encoded json string

```
dag:
  options:
    sl_env_var: "{\"SL_ROOT\": \"${root_path}\", \"SL_DATASETS\": \"${root_path}/datasets\", \"SL_TIMEZONE\": \"
Europe/Paris\"}"
 #...
```

Pre-load strategy

pre_load_strategy defines the strategy that can be used to conditionaly load the tables of a domain within the DAG.

Four possible strategies:

NONE

The load of the domain will not be conditionned and no pre-load tasks will be executed (the default strategy).

blocked URL

IMPORTED

This strategy implies that at least one file is present in the **landing area** (${*SL_ROOT}/incoming/{domain}* by default, if option *incoming_path* has not been specified). If there is one or more files to load, the method *sl_import* will be called to import the domain before loading it, otherwise the loading of the domain will be skipped.

```
dag:
  options:
    pre_load_strategy: "imported"
 #...
```

blocked URL

PENDING

This strategy implies that at least one file is present in the **pending datasets area of the domain** (${*SL_ROOT}/datasets/pending/{domain}* by default if option *pending_path* has not been specified), otherwise the loading of the domain will be skipped.

```
dag:
  options:
    pre_load_strategy: "pending"
 #...
```

blocked URL

ACK

This strategy implies that an **ack file** is present at the specified path (${*SL_ROOT}/datasets/pending/{domain}/{{{{ds}}}}.ack* by default if option *global_ack_file_path* has not been specified), otherwise the loading of the domain will be skipped.

```
dag:
  options:
    pre_load_strategy: "ack"
 #...
```

blocked URL

Load dependencies

load_dependencies defines wether or not we want to generate **recursively** all the **dependencies** associated to **each task** for which the transformation DAG was generated (*False* by default).

```
dag:
  options:
    load_dependencies: True
 #...
```

blocked URL

Additional options

Depending on the template chosen, a specific **factory class** will be instantiated to dynamically generate the **Airflow tasks** that will execute the Starlake's import, load and transform commands.

Each factory class defines additional options.

Bash

ai.starlake.airflow.bash.**StarlakeAirflowBashJob** is a concrete factory class that generates **Airflow tasks** using *airflow.operators.bash.BashOperator*.

An additional **SL_STARLAKE_PATH** option is required to specify the **path** to the *Starlake* **executable**.

Example of a generated DAG using load/airflow_scheduled_table_bash.py.j2 template

blocked URL

Dataproc

ai.starlake.airflow.gcp.**StarlakeAirflowDataprocJob** is another concrete factory class that generates **Airflow tasks** to execute Starlake's commands by submitting **Dataproc job** to the configured **Dataproc cluster**.

It delegates to an instance of the *ai.starlake.airflow.gcp.StarlakeAirflowDataprocCluster* class the responsibility to :

- **create** the **Dataproc cluster** by instantiating *airflow.providers.google.cloud.operators.dataproc.DataprocCreateClusterOperator*
- **submit Dataproc job** to the latter by instantiating *airflow.providers.google.cloud.operators.dataproc.DataprocSubmitJobOperator*
- **delete** the **Dataproc cluster** by instantiating *airflow.providers.google.cloud.operators.dataproc.DataprocDeleteClusterOperator*

The **creation** of the **Dataproc cluster** can be performed by calling the *create_cluster* method of the *cluster* property or by calling the *pre_tasks* method of the StarlakeAirflowDataprocJob (the call to the *pre_load* method will, behind the scene, call the *pre_tasks* method and add the optional resulting task to the group of Airflow tasks).

The **deletion** of the **Dataproc cluster** can be performed by calling the *delete_cluster* method of the *cluster* property or by calling the *post_tasks* method of the StarlakeAirflowDataprocJob.

Bellow is the list of additional options used to configure the **Dataproc cluster:**

name	type	description
cluster_id	str	the optional unique id of the cluster that will participate in the definition of the Dataproc cluster name (if not specified)
dataproc_name	str	the optional dataproc name of the cluster that will participate in the definition of the Dataproc cluster name (if not specified)
dataproc_project_id	str	the optional dataproc project id (the project id on which the composer has been instantiated by default)
dataproc_region	str	the optional region (*europe-west1* by default)
dataproc_subnet	str	the optional subnet (the *default* subnet if not specified)
dataproc_service_account	str	the optional service account (*service-{self.project_id}@dataproc-accounts.iam.gserviceaccount.com* by default)
dataproc_image_version	str	the image version of the dataproc cluster (*2.2-debian1* by default)
dataproc_master_machine_type	str	the optional master machine type (*n1-standard-4* by default)
dataproc_master_disk_type	str	the optional master disk type (*pd-standard* by default)
dataproc_master_disk_size	int	the optional master disk size (*1024* by default)
dataproc_worker_machine_type	str	the optional worker machine type (*n1-standard-4* by default)
dataproc_worker_disk_type	str	the optional worker disk size (*pd-standard* by default)
dataproc_worker_disk_size	int	the optional worker disk size (*1024* by default)
dataproc_num_workers	int	the optional number of workers (*4* by default)

All of these options will be used by default if no **StarlakeAirflowDataprocClusterConfig** was defined when instantiating **StarlakeAirflowDataprocCluster** or if the latter was not defined when instantiating **StarlakeAirflowDataprocJob**.

Bellow is the list of additional options used to configure the **Dataproc job**:

name	type	description
spark_jar_list	str	the required list of spark jars to be used (using , as separator)
spark_bucket	str	the required bucket to use for spark and biqquery temporary storage
spark_job_main_class	str	the optional main class of the spark job (*ai.starlake.job.Main* by default)
spark_executor_memory	str	the optional amount of memory to use per executor process (*11g* by default)
spark_executor_cores	int	the optional number of cores to use on each executor (*4* by default)
spark_executor_instances	int	the optional number of executor instances (*1* by default)

Cloud run

ai.starlake.airflow.gcp.**StarlakeAirflowCloudRunJob** is another concrete factory class that generates **Airflow tasks** to execute Starlake's commands by launching **Cloud run job**.

Bellow is the list of additional options used to configure the **Cloud run job**:

name	type	description
cloud_run_project_id	str	the optional cloud run project id (the project id on which the composer has been instantiated by default)
cloud_run_job_name	str	the required name of the cloud run job
cloud_run_region	str	the optional region (*europe-west1* by default)
cloud_run_async	bool	the optional flag to run the cloud run job asynchronously (*True* by default)
retry_on_failure	bool	the optional flag to retry the cloud run job on failure (*False* by default)
retry_delay_in_seconds	int	the optional delay in seconds to wait before retrying the cloud run job (*10* by default)

If the execution has been parameterized to be **asynchronous**, an *airflow.sensors.bash.BashSensor* will be instantiated to **wait** for the **completion** of the **C loud run job** execution.

Templates

Path

The template property within the DAG configuration may be defined as:

- an **absolute** path name
- a **relative** path name to the *${SL_ROOT}/metadata/dags/templates/* directory
- a **relative** path name to the *src/main/resources/templates/dags/* starlake resource directory

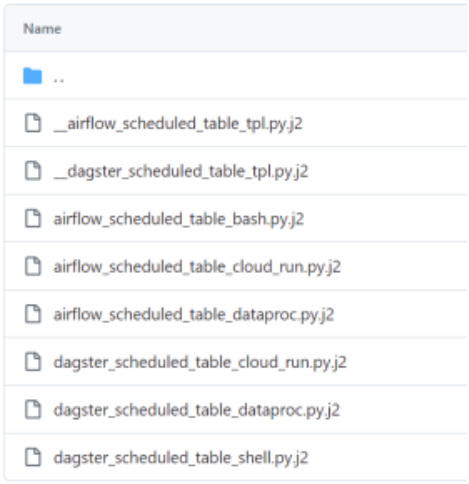
Starlake templates

Starlake templates are listed under the *src/main/resources/template/dags* **resource** directory.

There are **two types** of templates, those for **loading** data and others for **transforming data**.

Data loading

Starlake templates for data loading are listed under the **load** subdirectory.



__airflow_scheduled_table_tpl.py.j2 is the **abstract template** to generate Airflow DAGs for **data loading** which **requires** the instantiation of a **concrete factory class** that implements ai.starlake.airflow.**StarlakeAirflowJob**

Currently, there are **three concrete templates** for data loading.

All extend this abstract template by instantiating the corresponding concrete factory class using **include statements**.

- *airflow_scheduled_table_bash.py.j2*

```
# This template executes individual bash jobs and requires the following dag generation options set:
#
# - SL_STARLAKE_PATH: the path to the starlake executable [OPTIONAL]
# ...
{% include 'templates/dags/__starlake_airflow_bash_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_table_tpl.py.j2' %}
```

- *airflow_scheduled_table_cloud_run.py.j2*

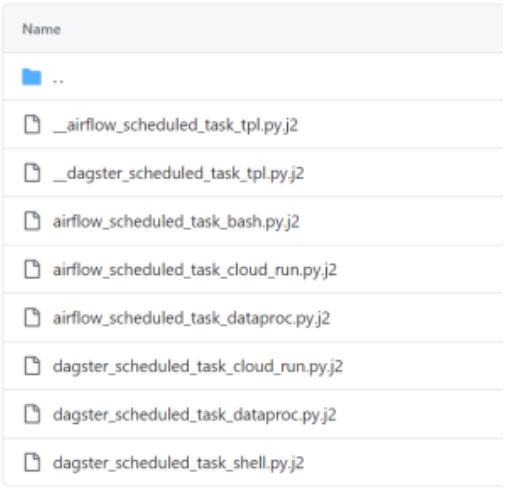
```
# This template executes individual cloud run jobs and requires the following dag generation options set:
#
# - cloud_run_project_id: the project id where the job is located (if not set, the project id of the composer
environment will be used) [OPTIONAL]
# - cloud_run_job_region: the region where the job is located (if not set, europe-west1 will be used) [OPTIONAL]
# - cloud_run_job_name: the name of the job to execute [REQUIRED]
# ...
{% include 'templates/dags/__starlake_airflow_cloud_run_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_table_tpl.py.j2' %}
```

- *airflow_scheduled_table_dataproc.py.j2*

```
# This template executes individual dataproc jobs and requires the following dag generation options set:
# - dataproc_name: the name of the dataproc cluster [OPTIONAL]
# - dataproc_project_id: the project id of the dataproc cluster (if not set, the project id of the composer
environment will be used) [OPTIONAL]
# - dataproc_region: the region of the dataproc cluster (if not set, europe-west1 will be used) [OPTIONAL]
# - dataproc_subnet: the subnetwork of the dataproc cluster (if not set, the default subnetwork will be used)
[OPTIONAL]
# - dataproc_service_account: the service account of the dataproc cluster (if not set, the default service
account will be used) [OPTIONAL]
# - dataproc_image_version: the image version of the dataproc cluster (if not set, 2.2-debian12 will be used)
[OPTIONAL]
# - dataproc_master_machine_type: the master machine type of the dataproc cluster (if not set, n1-standard-4
will be used) [OPTIONAL]
# - dataproc_master_disk_size: the master disk size of the dataproc cluster (if not set, 1024 will be used)
[OPTIONAL]
# - dataproc_master_disk_type: the master disk type of the dataproc cluster (if not set, pd-standard will be
used) [OPTIONAL]
# - dataproc_worker_machine_type: the worker machine type of the dataproc cluster (if not set, n1-standard-4
will be used) [OPTIONAL]
# - dataproc_worker_disk_size: the worker disk size of the dataproc cluster (if not set, 1024 will be used)
[OPTIONAL]
# - dataproc_worker_disk_type: the worker disk type of the dataproc cluster (if not set, pd-standard will be
used) [OPTIONAL]
# - dataproc_num_workers: the number of workers of the dataproc cluster (if not set, 4 will be used) [OPTIONAL]
# - spark_jar_list: the list of spark jars to be used [REQUIRED]
# - spark_bucket: the bucket to use for spark and biqquery temporary storage [REQUIRED]
# - spark_job_main_class: the main class of the spark job (if not set, the main class ai.starlake.job.Main will
be used) [OPTIONAL]
# - spark_executor_memory: the amount of memory to use per executor process (if not set, 11g will be used)
[OPTIONAL]
# - spark_executor_cores: the number of cores to use on each executor (if not set, 4 will be used) [OPTIONAL]
# - spark_executor_instances: the number of executor instances (if not set, 3 will be used) [OPTIONAL]
# ...
{% include 'templates/dags/__starlake_airflow_dataproc_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_table_tpl.py.j2' %}
```

Data transformation

Starlake templates for data transformation are listed under the **transform** subdirectory.



__airflow_scheduled_task_tpl.py.j2 is the **abstract template** to generate Airflow DAGs for **data transformation** which **requires**, in the same way, the instantiation of a **concrete factory class** that implements ai.starlake.airflow.**StarlakeAirflowJob**

Currently, there are **three concrete templates** for data transformation.

All extend this abstract template by instantiating the corresponding concrete factory class using **include statements**.

- *airflow_scheduled_task_bash.py.j2*

```
# ...
{% include 'templates/dags/__starlake_airflow_bash_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_task_tpl.py.j2' %}
```

- *airflow_scheduled_task_cloud_run.py.j2*

```
# ...
{% include 'templates/dags/__starlake_airflow_cloud_run_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_table_tpl.py.j2' %}
```

- *airflow_scheduled_task_dataproc.py.j2*

```
# ...
{% include 'templates/dags/__starlake_airflow_dataproc_job.py.j2' %}
{% include 'templates/dags/load/__airflow_scheduled_table_tpl.py.j2' %}
```

Customize existing templates

Although the options are useful for customizing the generated DAGs, there are situations where we need to be able to **dynamically** apply some of them **at runtime**.

Transform parameters

Often data transformation requires **parameterized SQL queries** whose parameters should be **evaluated at runtime**.

```
-- ...
step1 as(
  SELECT * FROM step0
  WHERE DAT_EXTRACTION >= '{{date_param_min}}' and DAT_EXTRACTION <= '{{date_param_max}}'
)
-- ...
```

User defined macros

All Starlake DAG templates for data transformation offer the ability of **injecting parameter values** via the optional definition of a **dictionary**-like **Python variable** named **jobs** where each key represents the **name of a transformation** and its value the **parameters** to be passed to the transformation. Each **entry** of this dictionary will be **added to** the **options** of the corresponding DAG.

```
#optional variable jobs as a dict of all parameters to apply by job
#eg jobs = {"task1 domain.task1 name": {"options": "task1 transform options"}, "task2 domain.task2 name":
{"options": "task2 transform options"}}
sl_job = StarlakeAirflowCloudRunJob(options=dict(options, **sys.modules[__name__].__dict__.get('jobs', {})))
```

```
# ai.starlake.job.IStarlakeJob
#...
    def sl_transform(self, task_id: str, transform_name: str, transform_options: str=None, spark_config:
StarlakeSparkConfig=None, **kwargs) -> T:
        """Transform job.
        Generate the scheduler task that will run the starlake `transform` command.

        Args:
            task_id (str): The optional task id.
            transform_name (str): The transform to run.
            transform_options (str): The optional transform options to use.
            spark_config (StarlakeSparkConfig): The optional spark configuration to use.

        Returns:
            T: The scheduler task.
        """
        task_id = f"{transform_name}" if not task_id else task_id
        arguments = ["transform", "--name", transform_name]
        transform_options = transform_options if transform_options else self.__class__.get_context_var
(transform_name, {}, self.options).get("options", "")
        if transform_options:
            arguments.extend(["--options", transform_options])
        return self.sl_job(task_id=task_id, arguments=arguments, spark_config=spark_config, **kwargs)
#...
```

Moreover, because the **SQL parameters** may be closely related to **Airflow context variable**(s), their **evaluation** may rely on some **Airflow user defined macros**.

All Starlake DAG templates for data transformation offer also the ability to specify User defined macros through the optional definition of a **dictionary**-like **P ython variable** named **user_defined_macros**

```
#...
# [START instantiate_dag]
with DAG(dag_id=os.path.basename(__file__).replace(".py", "").replace(".pyc", "").lower(),
         schedule_interval=None if cron == "None" else cron,
         schedule=schedule,
         default_args=sys.modules[__name__].__dict__.get('default_dag_args', DEFAULT_DAG_ARGS),
         catchup=False,
         user_defined_macros=sys.modules[__name__].__dict__.get('user_defined_macros', None),
         user_defined_filters=sys.modules[__name__].__dict__.get('user_defined_filters', None),
         tags=set([tag.upper() for tag in tags]),
         description=description) as dag:
#...
```

Because those variables have to be defined in the **same module** as that of the **generated DAG** (*options=dict(options, ******sys.modules[__name__].__dict__.get('jobs', {})****), user_defined_macros=****sys.modules[__name__].__dict__.get('user_defined_macros', None)***), we need to create a **customize d DAG template** that will allow us to specify those variables.

The new DAG template should **extend** the **existing one**(s), **including** our **specific code**

```
# metadata/dags/templates/__gne_jobs.py.j2
from gne.shared.services.job_params import get_days_interval,get_month_periode_depending_on_start_day_params

user_defined_macros = {
    "days_interval": get_days_interval,
    "month_periode_depending_on_start_day": get_month_periode_depending_on_start_day_params
}

#...
```

```
#gne_scheduled_task_cloud_run.py.j2 customized DAG template
{% include 'dags/templates/__gne_jobs.py.j2' %} # relative to the project metadata folder
{% include 'templates/dags/transform/airflow_scheduled_task_cloud_run.py.j2' %} # relative to src/main
/resources resource directory
```

In addition, those variables may be specified using **terraform variables** ...

```
# metadata/dags/templates/__gne_jobs.py.j2
#...

import json

jobs = json.loads("""${jobs}""")
```

variables.tf

```
variable "jobs" {
  type = list(object({
    domain  = string
    name    = string
    options = string
  }))
  default = []
}
```

main.tf

```
locals {
  jobs = tomap({
    for job in var.jobs :
      "${job.domain}.${job.name}" => {options=job.options}
  })

#...

resource "google_storage_bucket_object" "composer_storage_objects" {
  for_each = local.composer_storage_objects
  name     = each.value
  content  = templatefile(
    "${path.module}/${each.value}",
    merge(local.composer_storage_variables, {jobs=jsonencode(local.jobs)}, {clusters=jsonencode(var.clusters)})
  )
  bucket   = var.composer_bucket
}
```

map/vars_map.tfvars

```
jobs = [
   {
      domain  = "FLUX_INT"
      name    = "int_toge_a_aud_norm"
      options = "{{ days_interval(var.value.get('GNE_INT_TOGE_A_AUD_BASE_DATE', data_interval_end | ds), var.value.get
('GNE_INT_TOGE_A_AUD_DELTA', '30')) }}"
   },

...
   {
      domain  = "FLUX_INT"
      name   = "int_agr_toge_mensuel"
      options = "{{ month_periode_depending_on_start_day(var.value.get('GNE_INT_AGR_BASE_DATE', data_interval_end | ds), var.value.get
('GNE_INT_AGR_TOGE_START_DAY', '1')) }}"
   }
]
```

Finally, we will have to define a specific **DAG configuration** that will make use of our **customized DAG template**

```
---
dag:
  comment: "agregation dag for domain {{domain}} with cloud run" # will appear as a description of the dag
  template: "gne_scheduled_task_cloud_run.py.j2" # the dag template to use
  filename: "{{squad}}/{{squad}}_{{domain}}_agr_cloud_run.py" # the relative path to the outputDir specified as
a parameter of the `dag-generate` command where the generated dag file will be copied
  options:
    sl_env_var: "{\"SL_ROOT\": \"${root_path}\", \"SL_DATASETS\": \"${root_path}/datasets\", \"SL_TIMEZONE\": \"
Europe/Paris\"}"
    cloud_run_project_id: "${project_id}"
    cloud_run_job_name: "${job_name}-transform" # cloud run job name for auto jobs
    cloud_run_job_region: "${region}"
    cloud_run_async: False # whether or not to use asynchronous cloud run job execution
#    retry_on_failure: True # when asynchronous job execution has been selected, it specifies whether or not we
want to use a bash sensor with automatic retry for a specific exit code (implies airflow v2.6+)
    tags: "${squad} {{domain}} {{domain}}_CLOUD_RUN" # tags that will be added to the dag
    load_dependencies: False # whether or not to add all dependencies as airflow tasks within the resulting dag
    default_pool: "${squad}_default_pool" # pool to use for all tasks defined within the dag
```

Dataproc cluster configuration

All Starlake **DAG templates for dataproc** offer the ability to **customize** the **configuration** of the **dataproc cluster** through the implementation of optional **Python functions** that will return instances of either *StarlakeAirflowDataprocMasterConfig* or *StarlakeAirflowDataprocWorkerConfig* given the name of the config to apply, which, by default, will be evaluated to the name of the dag (if the option **cluster_config_name** has not been specified).

```
#__starlake_airflow_dataproc_job.py.j2
#...

#optional get_dataproc_master_config function that returns an instance of StarlakeAirflowDataprocMasterConfig
per dag name
dataproc_master_config = getattr(sys.modules[__name__], "get_dataproc_master_config",
default_dataproc_master_config)

#optional get_dataproc_worker_config function that returns an instance of StarlakeAirflowDataprocWorkerConfig
per dag name
dataproc_worker_config = getattr(sys.modules[__name__], "get_dataproc_worker_config",
default_dataproc_worker_config)

#optional get_dataproc_secondary_worker_config function that returns an instance of
StarlakeAirflowDataprocWorkerConfig per dag name
dataproc_secondary_worker_config = getattr(sys.modules[__name__], "get_dataproc_secondary_worker_config",
lambda dag_name: None)

cluster_config_name = StarlakeAirflowOptions.get_context_var("cluster_config_name", os.path.basename(__file__).
replace(".py", "").replace(".pyc", "").lower(), options)

#optional variable jobs as a dict of all options to apply by job
#eg jobs = {"task1 domain.task1 name": {"options": "task1 transform options"}, "task2 domain.task2 name":
{"options": "task2 transform options"}}
sl_job = StarlakeAirflowDataprocJob(
    cluster = StarlakeAirflowDataprocCluster(
        cluster_config=StarlakeAirflowDataprocClusterConfig(
            cluster_id=sys.modules[__name__].__dict__.get('cluster_id', cluster_config_name),
            dataproc_name=sys.modules[__name__].__dict__.get('dataproc_name', None),
            master_config = dataproc_master_config(cluster_config_name, **sys.modules[__name__].__dict__.get
('dataproc_master_properties', {})),
            worker_config = dataproc_worker_config(cluster_config_name, **sys.modules[__name__].__dict__.get
('dataproc_worker_properties', {})),
            secondary_worker_config = dataproc_secondary_worker_config(cluster_config_name),
            idle_delete_ttl=sys.modules[__name__].__dict__.get('dataproc_idle_delete_ttl', None),
            single_node=sys.modules[__name__].__dict__.get('dataproc_single_node', None),
            options=options,
            **sys.modules[__name__].__dict__.get('dataproc_cluster_properties', {})
        ),
        pool=sys.modules[__name__].__dict__.get('pool', None),
        options=options
    ),
    options=dict(options, **sys.modules[__name__].__dict__.get('jobs', {}))
)
```

Again, because those functions should be implemented in the **same module** as that of the **generated DAG** (*dataproc_master_config* = ***getattr(sys.modules[__name__], "get_dataproc_master_config", default_dataproc_master_config)***, ...), we need to create a **customized DAG template** that will allow us to implement those methods.

A **good practice** will be to inject those configurations via the use of **Terraform variables**.

```
#__gne_dataproc.py.j2 custom code

import json

from ai.starlake.job.airflow import AirflowStarlakeOptions
from ai.starlake.job.airflow.gcp import StarlakeDataprocWorkerConfig

clusters:dict = json.loads("""${clusters}""") # Terraform variable

# ...

def get_dataproc_worker_config(cluster_config_name: str, **kwargs):
    # lookup a specific configuration given the name of the cluster configuration
    worker_config = AirflowStarlakeOptions.get_context_var(cluster_config_name.upper().replace('-', '_'),
clusters.get(cluster_config_name, None), options, deserialize_json=True)
    if worker_config:
        return StarlakeDataprocWorkerConfig(
            num_instances=int(worker_config.get('numWorkers', 0)),
            machine_type=worker_config.get('workerType', None),
            disk_type=None,
            disk_size=None,
            options=options,
            **kwargs
        )
    else:
        return None

# additional dataproc cluster properties
dataproc_cluster_properties = {
    "spark:spark.driver.maxResultSize": "15360m",
    "spark:spark.driver.memory": "30720m",
}
```

```
# gne_scheduled_task_dataproc.py.j2 our customized DAG template for data transformation using dataproc

{% include 'dags/templates/__gne_jobs.py.j2' %} # specific code to inject jobs parameters
{% include 'dags/templates/__gne_dataproc.py.j2' %} # specific code to customize the configuration of our
dataproc cluster
{% include 'templates/dags/transform/scheduled_task_dataproc.py.j2' %} # the base Starlake DAG template that
needs to be extended
```

```
---
# norm_dataproc_domain.sl.yml our DAG configuration using our customized DAG template

dag:
  comment: "dag for transforming tables for domain {{domain}} with dataproc" # will appear as a description of
the dag
  template: "gne_scheduled_task_dataproc.py.j2" # the dag template to use
  filename: "{{squad}}/{{squad}}_{{domain}}_norm_dataproc.py" # the relative path to the outputDir specified as
a parameter of the `dag-generate` command where the generated dag file will be copied
  options:
    sl_env_var: "{\"SL_ROOT\": \"${root_path}\", \"SL_DATASETS\": \"${root_path}/datasets\", \"SL_TIMEZONE\": \"
Europe/Paris\"}"

    dataproc_name: "${dataproc_name}"
    dataproc_project_id: "${project_id}"
    dataproc_region: "${region}"
    dataproc_subnet: "${subnet}"
    dataproc_service_account: "${dataproc_service_account}"
    dataproc_image_version: "${dataproc_image_version}"
    dataproc_master_machine_type: "${dataproc_master_machine_type}"
    dataproc_worker_machine_type: "${dataproc_worker_machine_type}"
    dataproc_num_workers: "${dataproc_num_workers}"
    cluster_config_name: "${squad}-{{domain|lower|replace('_', '-')}}-norms"# the name of the cluster
configuration that will be looked up
    spark_config_name: "${squad}-{{domain|lower|replace('_', '-')}}-norms"
    spark_jar_list: "gs://${artefacts_bucket}/${main_jar}" #gs://${artefacts_bucket}/org.yaml/snakeyaml/2.2/jars
/snakeyaml-2.2.jar gs://spark-lib/bigquery/spark-3.5-bigquery-0.35.1.jar gs://${artefacts_bucket}/com.google.
cloud.spark/spark-bigquery-with-dependencies_2.12/${spark_bq_version}/spark-bigquery-with-dependencies_2.
12-${spark_bq_version}.jar
    spark_bucket: "${datastore_bucket}"

    tags: "${squad} {{domain}} {{domain}}_DATAPROC" # tags that will be added to the dag
    load_dependencies: False # whether or not to add all dependencies as airflow tasks within the resulting dag
    default_pool: "${squad}_default_pool" # pool to use for all tasks defined within the dag
```

variables.tf

```
variable "clusters" {
  type = map(object({
    workerType            = string
    numWorkers            = string
    sparkExecutorInstances = string
    numVcpu               = string
    memAlloc              = string
  }))
  default = {}
}
```

main.tf

```
resource "google_storage_bucket_object" "composer_storage_objects" {
  for_each = local.composer_storage_objects
  name     = each.value
  content  = templatefile(
    "${path.module}/${each.value}",
    merge(local.composer_storage_variables, {jobs=jsonencode(local.jobs)}, {clusters=jsonencode(var.clusters)})
  )
  bucket   = var.composer_bucket
  depends_on = [
    google_storage_bucket_object.metadata_storage_objects,
    null_resource.generate_int_dags
  ]
}
```

map/vars_map.tfvars

```
clusters = {
  gne-flux-int-ingest = {
    workerType            = "n1-standard-8"
    numWorkers            = "3"
    sparkExecutorInstances = "2"
    numVcpu               = "5"
    memAlloc              = "23g"
  },
  gne-flux-int-norms = {
    workerType            = ""
    numWorkers            = "0"
    sparkExecutorInstances = "0"
    numVcpu               = "0"
    memAlloc              = ""
  },
  gne-flux-int-agregat = {
    workerType            = "n1-standard-16"
    numWorkers            = "4"
    sparkExecutorInstances = "3"
    numVcpu               = "5"
    memAlloc              = "17g"
  },
  gne-flux-int-takeoff = {
    workerType            = ""
    numWorkers            = "0"
    sparkExecutorInstances = "0"
    numVcpu               = "0"
    memAlloc              = ""
  }
}
```

Spark configuration

As for the configuration of the dataproc cluster, it is possible to **customize** the **spark configuration** thanks to the optional implementation of a **Python function** named **get_spark_config** that will return an instance of *StarlakeSparkConfig* given the name of a spark configuration to apply, which by default is the name of the transformation (if the option **spark_config_name** has not been defined).

```
# __common_airflow.py.j2

spark_config = getattr(sys.modules[__name__], "get_spark_config", default_spark_config)

# __airflow_scheduled_task_tpl.py.j2

    def create_task(airflow_task_id: str, task_name: str, task_type: str):
        spark_config_name=StarlakeAirflowOptions.get_context_var('spark_config_name', task_name.lower(),
options)
        if (task_type == 'task'):
            return sl_job.sl_transform(
                task_id=airflow_task_id,
                transform_name=task_name,
                spark_config=spark_config(spark_config_name, **sys.modules[__name__].__dict__.get
('spark_properties', {}))
            )
```

Again, because this function should be implemented in the **same module** as that of the **generated DAG**, we need to create a **customized DAG template** that will allow us to implement this method, and a **good practice** will be to inject those configurations via the use of **Terraform variables**.

```
#__gne_dataproc.py.j2 custom code

import json

from ai.starlake.job import StarlakeSparkConfig
from ai.starlake.job.airflow import AirflowStarlakeOptions

clusters:dict = json.loads("""${clusters}""") # Terraform variable

def get_spark_config(spark_config_name: str, **kwargs):
    # use of the Terraform variable to lookup the spark configuration
    spark_config = AirflowStarlakeOptions.get_context_var(spark_config_name.upper().replace('-', '_'), clusters.
get(spark_config_name, None), options, deserialize_json=True)
    if spark_config:
        return StarlakeSparkConfig(
            memory=spark_config.get('memAlloc', None),
            cores=int(spark_config.get('numVcpu', 0)),
            instances=int(spark_config.get('sparkExecutorInstances', 0)),
            cls_options=AirflowStarlakeOptions(),
            options=options,
            **kwargs
        )
    else:
        return None

#...
```

Dependencies

For any transformation, *Starlake* is able to calculate all its dependencies towards other tasks or loads thanks to the **analysis** of **SQL queries**.

As seen previously, the **load_dependencies option** defines whether or not we wish to recursively **generate all the dependencies** associated with each task for which the transformation DAG must be generated (False by default). If we choose to not generate those dependencies, the corresponding DAG will be scheduled using the Airflow's **data-aware scheduling mechanism**.

All **dependencies** for data transformation are **available** in the generated DAG via the **Python** dictionary **variable task_deps**.

```
task_deps=json.loads("""[ {
  "data" : {
    "name" : "Customers.HighValueCustomers",
    "typ" : "task",
    "parent" : "Customers.CustomerLifeTimeValue",
    "parentTyp" : "task",
    "parentRef" : "CustomerLifetimeValue",
    "sink" : "Customers.HighValueCustomers"
  },
  "children" : [ {
    "data" : {
      "name" : "Customers.CustomerLifeTimeValue",
      "typ" : "task",
      "parent" : "starbake.Customers",
      "parentTyp" : "table",
      "parentRef" : "starbake.Customers",
      "sink" : "Customers.CustomerLifeTimeValue"
    },
    "children" : [ {
      "data" : {
        "name" : "starbake.Customers",
        "typ" : "table",
        "parentTyp" : "unknown"
      },
      "task" : false
    }, {
      "data" : {
        "name" : "starbake.Orders",
        "typ" : "table",
        "parentTyp" : "unknown"
      },
      "task" : false
    } ],
    "task" : true
  } ],
  "task" : true
} ]""")
```

Inline

In this strategy (**load_dependencies** = *True*), all the dependencies related to the transformation will be generated.

blocked URL

Data-aware scheduling

In this strategy (**load_dependencies** = *False*), the default strategy, a **schedule** will be created to check if the **dependencies** are **met** via the use of Airflow **Datasets**.

```
#...
schedule = None

datasets: Set[str] = []

_extra_dataset: Union[dict, None] = sys.modules[__name__].__dict__.get('extra_dataset', None)

_extra_dataset_parameters = '?' + '&'.join(list(f'{k}={v}' for (k,v) in _extra_dataset.items())) if
_extra_dataset else ''

# if you choose to not load the dependencies, a schedule will be created to check if the dependencies are met
def _load_datasets(task: dict):
    if 'children' in task:
        for child in task['children']:
            datasets.append(keep_ascii_only(child['data']['name']).lower())
            _load_datasets(child)

if load_dependencies.lower() != 'true':
    for task in task_deps:
        _load_datasets(task)
    schedule = list(map(lambda dataset: Dataset(dataset + _extra_dataset_parameters), datasets))

#...

with DAG(dag_id=os.path.basename(__file__).replace(".py", "").replace(".pyc", "").lower(),
         schedule_interval=None if cron == "None" else cron,
         schedule=schedule,
         default_args=sys.modules[__name__].__dict__.get('default_dag_args', DEFAULT_DAG_ARGS),
         catchup=False,
         user_defined_macros=sys.modules[__name__].__dict__.get('user_defined_macros', None),
         user_defined_filters=sys.modules[__name__].__dict__.get('user_defined_filters', None),
         tags=set([tag.upper() for tag in tags]),
         description=description) as dag:
#...
```

blocked URL

Those required Datasets are updated for each load and task that have been executed.

The *ai.starlake.airflow.StarlakeAirflowJob* class is responsible for **recording** the *outlets* related to the execution of each starlake command.

```
def __init__(
    self,
    pre_load_strategy: Union[StarlakePreLoadStrategy, str, None],
    options: dict=None,
    **kwargs) -> None:
    #...
    self.outlets: List[Dataset] = kwargs.get('outlets', [])

def sl_import(self, task_id: str, domain: str, **kwargs) -> BaseOperator:
    #...
    dataset = Dataset(keep_ascii_only(domain).lower())
    self.outlets += kwargs.get('outlets', []) + [dataset]
    #...

def sl_load(
    self,
    task_id: str,
    domain: str,
    table: str,
    spark_config: StarlakeSparkConfig=None,
    **kwargs) -> BaseOperator:
    #...
    dataset = Dataset(keep_ascii_only(f'{domain}.{table}').lower())
    self.outlets += kwargs.get('outlets', []) + [dataset]
    #...

def sl_transform(
    self,
    task_id: str,
    transform_name: str,
    transform_options: str=None,
    spark_config: StarlakeSparkConfig=None,
    **kwargs) -> BaseOperator:
    #...
    dataset = Dataset(keep_ascii_only(transform_name).lower())
    self.outlets += kwargs.get('outlets', []) + [dataset]
    #...
```

All the *outlets* that have been recorded are available in the **outlets** property of the Starlake concrete factory class instance and are used at the very last step of the corresponding DAG to update the Datasets.

```
    end = sl_job.dummy_op(task_id="end", outlets=[Dataset(keep_ascii_only(dag.dag_id))]+list(map(lambda x:
Dataset(x.uri + _extra_dataset_parameters), sl_job.outlets)))
```

In conjonction with the Starlake dag generation, the *outlets* property can be used to **schedule effortless** DAGs that will run the **transform** commands.