outflow

Release 0.7.0

Grégoire Duvauchelle

Aug 30, 2022

CONTENTS:

Outflow is a framework that helps you build and run task workflows.

The API is as simple as possible while still giving the user full control over the definition and execution of the workflows.

Feature highlight :

- Simple but powerful API
- Support for **parallelized and distributed execution**
- Centralized **command line interface** for your pipeline commands
- Integrated **database** access, sqlalchemy models and alembic migrations
- Executions and exceptions logging for **tracability**
- Strict type and input/output checking for a **robust** pipeline

CHAPTER

ONE

OVERVIEW

1.1 Introduction

1.1.1 What is outflow

Outflow is a framework that helps you write and run pipelines.

Outflow is a reimplementation of the core of Poppy, the pipeline framework originally developed by the RPW Operation Center at LESIA for their data reduction pipeline.

The objective of outflow is to add task and workflow parallelization capabilities to the Poppy framework. With outflow, you can parallelize your workflow either on a single machine, or on a computing cluster managed by Slurm.

1.1.2 Installation

Outflow is on PyPI! You can easily install the latest stable version with `pip install outflow`. Since Outflow is still in development so you might want to install the development versions, clone and install our git repository with `pip install -e .`

System requirements

- Python 3.7 and up
- Any linux distribution or Windows

CHAPTER

TWO

TUTORIAL

2.1 Create a pipeline with Outflow, part 1: Pipeline and plugins

In this tutorial, we will create a pipeline from scratch with Outflow, and walk through the main features of the framework.

First, you should install Outflow using *pip install outflow* to get the latest release from pypi. Then, you can check that outflow is correctly installed with :

```
$ python -m outflow --version
```

If Outflow is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named outflow".

Note: If you have any question or suggestions about the tutorial or Outflow in general, please come over to our Discord server

2.1.1 Creating a pipeline

To run properly, a pipeline built with Outflow needs some configuration files. You can use the following commands to generate the pipeline directory structure :

```
$ python -m outflow management create pipeline tuto_pipeline
```

You will get a directory called *tuto_pipeline/* in the current directory containing multiple files :

```
tuto_pipeline
├── config.yml
├── plugins/
├── manage.py
├── requirements.txt
└── settings.py
```

- `config.yml` : Contains configurations about your pipeline. This can vary from one pipeline execution to another and you can have several configuration files and choose it in your command line.
- `plugins` : This is where we will create our plugins. Plugins in this directory are automatically put in the python path by manage.py. However plugins can live anywhere as long as they are available in the python path (ie pip installed).
- `settings.py` : This file is specific to your pipeline and should be versioned. This contains among other things a list of the plugins used by your pipeline. See settings for full specification.

- `requirements.txt` : Contains the list of python dependencies
- `manage.py` : The entry point of the pipeline.

2.1.2 Create a plugin

A pipeline is not much without some tasks to execute. In this tutorial, we will use the example of a data reduction pipeline, so our tasks will be computations on some data.

With Outflow, tasks are defined in **plugins**. A plugin is a dedicated python package containing commands, tasks and models.

- Tasks are the building blocks of the pipeline, they have inputs, outputs, and a function to execute.
- Commands are used to describe a graph of tasks dependencies, as well as a cli entrypoint and its arguments.
- A model is a python class describing a database table. (optional)

We will see in the next tutorial chapters what those are in details.

For now, outflow plugins must use PEP 420 packages. This allows to have multiple plugins under the same namespace.

To create a new plugin, type the following command :

```
$ cd tuto_pipeline
$ python -m outflow management create plugin tuto.data_reduction --plugin_dir plugins/
↪data_reduction
```

This creates all the needed files containing an example of a basic command.

Then in the tuto_pipeline/settings.py file, add your new plugin to the plugin list

```
PLUGINS = [
    'outflow.management',
    'tuto.data_reduction',
]
```

You can test your newly created plugin by calling the command generated in the commands.py:

```
$ python manage.py data_reduction
```

You should see the following output on the command line:

```
* tuto.data_reduction.commands - commands.py:49 - INFO - Hello from data_reduction
```

If you do, congratulations! We now have everything we need to get you started with Outflow.

In the next chapter, we will add new tasks and commands to this pipeline template.

2.2 Create a pipeline with Outflow, part 2: Tasks and commands

The main feature of the Outflow framework is to manage tasks and their execution. In this chapter, we will see how to create tasks, how to execute them in a given order and piping data between them.

2.2.1 Getting started with tasks

To help you getting started, the plugin generated by the command you ran in the previous chapter contains two example tasks and a command.

First, go to the tasks.py file in the plugin directory. You will find two tasks: *GenData* that outputs a value, and *PrintData* that prints this value using the outflow logger.

Let's look at one of the example tasks, GenData :

```
from outflow.core.tasks import Task

@as_task
def GenData():
    raw_data = 42
    return {"raw_data": raw_data}
```

As you can see, you can easily create Outflow tasks with a regular python function that you decorate with the `@as_task` decorator. The decorator will turn this function into a Task subclass called GenData, that is why its name is written in CamelCase.

Tasks must return a dictionary, with keys matching the inputs of the next tasks, here the key is *raw_data*, like the input of the task PrintData below.

Outflow is here to help you build robust pipelines, for this reason tasks need to define input targets and output targets. Before running your workflow, Outflow uses these targets to check if the input and output of your tasks match each other.

Outflow wants to be robust but not at the cost of the user experience, that is why in the simpler cases like this one, outflow will try to guess the inputs and outputs of your task. More about inputs and outputs of tasks later in this chapter.

2.2.2 Commands

In Outflow, commands are entry points to your workflows.

Let's go to *commands.py* and look at the example command. This command is called *ComputeData* and executes the workflow:

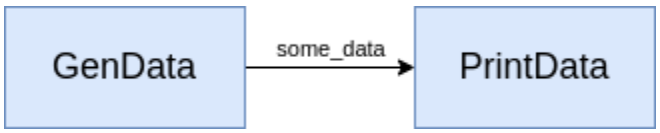


Fig. 1: Schema of the workflow executed by this command

```
from outflow.core.commands import Command, RootCommand
from .tasks import GenData, PrintData

```

(continues on next page)

(continued from previous page)

```
@RootCommand.subcommand(db_untracked=True)
class ComputeData(Command):
    def setup_tasks(self):

        # setup the workflow
        GenData() | PrintData()
```

You can see that in order to setup your workflow, you should use the operator | between the tasks that you want to pipe. This operator is overloaded for the class Task, it will create the dependency between tasks, and at runtime will send the result of the task on the left to the task on the right of the operator.

Warning: You should remember that inside the setup_tasks() method, nothing is executed yet. In this method, you build the dependency graph between your tasks. It is only after the execution of this setup method that Outflow will run each task in the right order.

You should be able to run this command with :

```
$ python manage.py compute_data
```

All the registered commands are by default callable with their snake case name. The parameter *name* of the *@RootCommand.subcommand()* decorator is available to override the cli name of the command.

2.2.3 Modify the workflow

We will edit the workflow by adding a new task between *GenData* and *PrintData*:

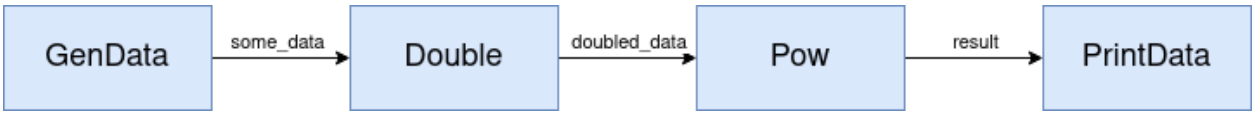


Fig. 2: Schema of the new workflow

Add a new task

Start by adding two tasks to the *tasks.py* file:

```
import time
from outflow.core.tasks import as_task


@as_task
def Transform(raw_data):
    logger.info(f"Transforming data {raw_data=}")
    # Simulate a big computation
    time.sleep(2)
    result = raw_data * 2

    # return the result for the next task
    return {"transformed_data": result}
```

(continues on next page)

(continued from previous page)

```
@as_task
def Compute(transformed_data):
    logger.info(f"Running computation for {transformed_data=}")
    time.sleep(2)
    result = transformed_data / 2

    return {"computation_result": result}
```

and edit the arguments of the task *PrintData* to account for the new output name.

```
@as_task
def PrintData(computation_result):
    logger.info(f"Result of the computation: {computation_result}")
```

Edit a command

Then edit the command by inserting our new tasks in the workflow definition:

```
from .tasks import GenData, PrintData, Transform, Compute

@RootCommand.subcommand(db_untracked=True)
class ComputeData(Command):
    def setup_tasks(self):

        # setup the workflow
        GenData() | Transform() | Compute() | PrintData()
```

Note: If you don't feel comfortable with the pipe operator, you can also use the bitwise right shift operator : >> . They are equivalent for tasks.

You can now run the command again and see that the output is different:

```
$ python manage.py compute_data
2020-11-17 18:19:58,369 - tuto.data_reduction.tasks - tasks.py:15 - INFO - Result of the␣
↪computation: 168
```

2.2.4 Task targets

In Outflow, Targets are simple objects that have a name and a type. The type can either be a native type like `int` or a type from the typing module. These targets objects are used by Outflow to represent the input and outputs of each tasks.

You will most likely not need to manipulate these Targets directly, as Outflow performs automatic target detection:

- For input targets, automatic target detection is done by looking at the arguments of the decorated function.
- For output targets, this is done by looking at the dictionary after the return statement. Try to replace `return {"raw_data": raw_data}` with `return raw_data` and you will see that if you try to run this workflow, it will raise an error like this : `Could not automatically determine outputs of task gen_data`.

Specify targets type

At runtime, Outflow will not only check if the name of targets match each other, but will also check the type of the target matches the type of the objects inside the dictionary returned by the task. If not specified by user, Outflow will consider targets to be of type `typing.Any`, meaning it will always work.

However, it is strongly recommended to type your tasks input and outputs. Imagine you make a mistake or a library has an unexpected behaviour and you don't return the right object, your commands could fail in the middle of a task : inside the task GenData, try to change the value of `raw_data` from the int `42` to the string `"42"`. Your command will run but will not print the expected value.

To type your input targets, you should use the usual python type hinting with the colon character. Let's annotate task Compute :

```
from outflow.core.tasks import Task

@as_task  #                 ↓ add this type hint annotation
def Compute(transformed_data: int):
    ...
```

Now, your task Compute will only run if the type of `raw_data` is an int. Try running the command again and it will now fail with an error like : `your type of argument "raw_data" must be int; got str instead`

To type task outputs, you should put a typed dictionary in the return annotation of the function. Let's annotate the task GenData :

```
@as_task  #   ↓ add this return type annotation
def GenData() -> {"raw_data": int}:
    raw_data = "42" # still keep the wrong type here
    return {"raw_data": raw_data}
```

Now if you try to call the command once again, the execution will fail even earlier, when GenData returns : `type of dict item "raw_data" for return_value must be int; got str instead`. You can now put the right value back in raw_data : `raw_data = 42`.

This runtime type checking is really useful and will save you time in the long run.

You have seen how to define input and output targets along with their type. Using `as_task` decorator and annotations is the recommended way to define tasks and their targets.

If you do not feel comfortable with this syntax, or that you task need special behaviour, it is always possible to subclass Task manually and use the `setup_targets` method. Check the commented tasks at the bottom of the tasks.py file for an example of tasks using the explicit syntax. There is also the page tasks in the user documentation that goes over this.

Tasks with only one output

For a simple task like GenData that returns a dictionary with only one key, Outflow allows you to return only the object directly and not a dictionary, like so `return raw_data`. .. code-block:: python

@as_task def GenData() -> {"raw_data": int}:

raw_data = 42 return raw_data # return the object directly # return {"raw_data": raw_data}

But remember, this shortcut is possible **only** if you correctly specified a typed dictionary with only one key in the return annotation of your task.

2.2.5 Adding task parameters

This command is not very interesting because it will output the same value every time. To make it a little more interesting, you can add a task parameter that will decide by how much the task multiplies our generated value.

A task parameter is a special kind of task input. As opposed to regular task inputs that comes from the upstream tasks, a parameter comes from the pipeline configuration file.

Edit the *Compute* task and add a parameter :

```
from outflow.core.types import Parameter

@as_task
def Compute(transformed_data: int, multiplier: Parameter(int)) -> {"computation_result":␣
↪int}:
    time.sleep(4)
    # Compute the product of "raw_data" and the multiplier
    result = transformed_data * multiplier
    return result
```

You define a parameter by using the custom type outflow.core.types.Parameter in the type hint annotation of your task input.

Then go to the config.yml file of your pipeline to add a value for `multiplier`. In this file, you will find a commented section called `parameters`. This is the section where outflow look for parameters defined as above.

Uncomment the section. Then replace `my_task` with `compute` (here we use the automatic lowercase name of the task). Finally replace `my_param1: param_value1` with `multiplier: 3`

```
...

parameters:
  compute:
    multiplier: 3

...
```

Finally, let's try again the command, now with a task that has a parameter.

```
$ python manage.py compute_data
tuto.data_reduction.tasks - tasks.py:15 - INFO - Result of the computation: 28.0
```

Parameter are useful to keep track of values that can change between pipeline executions, as they are recorded into the outflow internal database (not yet set up at this point of the tutorial). This allows to check with which parameter was a given command ran in the past.

This is also useful for the workflow cache that we will see in the next section.

Note: class Task has a method `add_parameter()` that you can use in the `setup_targets()` method if you want to use the explicit syntax.

2.2.6 Conclusion

Congratulations, you finished the outflow tutorial part 2! In the next chapter, we will see how to use workflows.

2.3 Create a pipeline with Outflow, part 3: Workflow and caching

In the previous chapter, we used the database as a manual caching mechanism. Outflow supports automatic output caching using the Workflow object.

Until now, we talked about workflows as being a sequence of task. However Outflow has a Workflow class that has a bunch of useful additional features compared to a simple sequence of task. Mainly, Workflows helps reusing common sequences of tasks in multiple commands. Workflows also supports caching their outputs, so that it is not ran if the inputs and tasks parameters did not change.

2.3.1 Workflows

Now let's turn the task Compute into a workflow object :

```
from outflow.core.workflow import as_workflow
from .tasks import GenData, PrintData, Compute


# create a workflow
@as_workflow
def process_data():
    Transform() | Compute()


@RootCommand.subcommand(db_untracked=True)
class ComputeData(Command):
    def setup_tasks(self):
        # setup the main workflow by using the workflow defined outside command
        GenData() | process_data() | PrintData()
```

Workflows behave like tasks, in the way that they have input and output targets, and can be pipe to tasks or other workflows. The input targets of a workflow are automatically copied from the first task of the workflow, and the output copied from the last task of the workflow.

Set up like this, the command should behave exactly as before (try it). One difference is that now you can reuse `process_data` workflow in multiple commands if needed.

2.3.2 Workflow caching

Workflows have a built in cache mechanism that caches the outputs of the workflows, and avoid running it again if it detects that it is called with exactly the same input values, and the task inside this workflow has the same Parameter values.

Workflow cached outputs are store on the disk. By default, the location is `settings.TEMP_DIR / "workflow_cache"`. You can edit this location either in the config.yml with the `workflow_cache_dir` key, or edit the TEMP_DIR value in your settings.py file.

To activate workflow caching, you only have to add `cache=True` in the Workflow decorator :

```
@as_workflow(cache=True)
def process_data():
    Transform() | Compute()
```

Now if you run the command twice in a row, you should see by looking a the logs that only the PrintData task is executed the second time.

To check that the cache system is working as intended, you can try to either change the return value of task GenData, or change the value of the parameter `multiplier` of task Compute. In both cases the workflow should detect that an input changed and execute again.

Overwrite cache

If you want to explicitly ignore cached results, you can use the *–overwrite-cache* argument in your cli. Outflow will not try to detect changes in your code, so you will have to use this flag if you change your code without changing anything else.

2.4 Create a pipeline with Outflow, part 4: Parallelize your workflows

Now is time to dive into the most interesting feature of outflow: parallelization of workflows on multiple cores, and distribution on a slurm computing cluster.

Note: You can follow this section until the last part if you do not have access to slurm. You can also check developers documentation to start a docker to simulate a slurm cluster.

2.4.1 The MapWorkflow

We will modify our workflow so that GenData return a list of integer, and using the MapWorkflow we will execute multiple times compute_workflow, once on each element of the list.

There are many types of parallel workflows, and Outflow currently supports map-reduce and branching. In this tutorial, we will use the map-reduce construction to create the following parallel workflow:

TODO update diagram

Fig. 3: Schema of the parallel workflow

We will configure and add an instance of outflow.library.tasks.MapWorkflow to our command.

Note: The class outflow.library.tasks.MapWorkflow is a generic class that will instantiate the right MapWorkflow depending on the backend. This means you can execute your workflow containing a MapWorkflow with any supported backend (default, multiprocessing, or slurm backends).

2.4.2 Edit the tasks

First, let's modify our task *GenData* so it returns a list instead of one integer:

```
@as_task
def GenData() -> {"raw_data_array": list}:  # <-- edit the return annotation
    raw_data_array = [42, 43, 44, 45]
    return {"raw_data_array": raw_data_array}
```

2.4.3 Add MapWorkflow to the command workflow

To write a *map* with outflow, we will use an instance of outflow.library.workflow.MapWorkflow. Outflow comes with a library of common tasks and workflows that you don't need to write, MapWorkflow is one of them.

Import *MapWorkflow* in *commands.py* and make ComputeWorkflow a MapWorkflow :

```
from outflow.library.workflows import MapWorkflow

# Change the decorator and disable cache for now
@MapWorkflow.as_workflow(cache=False)
def process_data():
    Transform() | Compute()
```

We will reactivate the cache at the end of this section.

Let's try this new command just like before:

```
$ python manage.py compute_data
```

You should see that outflow is running 4 instance of our task Compute sequentially:

```
tuto.data_reduction.tasks - tasks.py:51 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:51 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:51 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:51 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:27 - INFO - Result of the mapped computation: [[{
↪'computation_result': 126}], [{'computation_result': 129}], [{'computation_result':␣
↪132}], [{'computation_result': 135}]]
```

Since GenData has only one output and Transform has only one output, MapWorkflow knows it has to split the output of GenData and each item is the input of each executed workflow. If GenData had more than one output, or Transform more than one input, we would have to specify how to split the inputs of the MapWorkflow using the *map_on* argument of the decorator. It takes a dictionnary, key is the name of the input sequence, and the value is the name of the target input of the first task of the workflow that will receive each item of the sequence.

2.4.4 The parallel backend

With this configuration, Outflow executes each iteration of the MapWorkflow sequentially. This requires no special configuration and this can be useful for simple pipelines.

If you have bigger datasets, you can run these workflows in parallel on your local machine, using the *parallel* backend.

Let's edit our `GenData` task to generate more data and see that it computes linearly faster with the number of available cpus:

```
@as_task
def GenData() -> {"raw_data_array": list}:
    raw_data_array = [i for i in range(6)]
    return raw_data_array
```

Call our usual command:

```
$ python manage.py compute_data --backend parallel
```

This commands runs 6 workflows in parallel, each lasting 4 seconds. Depending on the number of cpus on your machine, this workflow will not take the same time to execute, but in any case it should be faster than with the sequential backend.

2.4.5 Distribute workflows on a slurm cluster

Note: An sqlite database cannot keep up with all the access from the different nodes running outflow tasks. It is strongly recommended to connect to a PostgreSQL database before continuing. See section database for how to connect to a postgres database.

If you have even bigger datasets, and access to a slurm computing cluster, Outflow can run its workflow using the slurm backend.

If you have access to a slurm cluster, simply run the command with the argument `--backend slurm`.

2.4.6 Allocate multiple cpus for multiprocessing computations for each workflow

Let's say that we can not only parallelize our pipeline on data but each computation can itself be shared between multiple cpu. We can simulate this by dividing the `time.sleep()` in our *Compute* task by the number of CPUs available:

```
# in task Compute
...
except NoResultFound:
    # Result not in the database: compute the value like before
    logger.info("Result not found in database, computing result and inserting")

    # add this line to access number of CPUs available
    from multiprocessing import cpu_count()

    # simulate a multiprocess computation
    computation_time = 5/cpu_count()
    logger.info(f"{cpu_count()} CPUs available, computation will last {computation_time}␣
↪seconds")
    time.sleep(computation_time)
```

(continues on next page)

(continued from previous page)

```
    computation_result = raw_data * multiplier
```

Then, edit MapWorkflow parameter to allow multiple CPUs per mapped workflows:

```
# in setup_task() of ComputeData command

            # ↓ add this parameter
@MapWorkflow.as_workflow(cpus_per_task=5, output_name="computation_result") as map_task:
def
    Compute()
```

You are now ready to run the usual command (with another multiplier again) and see that it will execute faster than the time it takes to say "Outflow is awesome".

```
$ python manage.py compute_data --multiplier 15
```

When using slurm, you can specify sbatch directives directly to the MapWorkflow arguments. See mapworkflow for details on MapWorkflow usage.

Congratulations, you arrived at the end of the Outflow tutorial! You should now know enough about the framework to create your own pipeline.

2.5 Create a pipeline with Outflow, part 5: Models and database

In this chapter, we will learn how to use the database by creating models for our plugin, generating migration, and insert and query the database using our model.

2.5.1 Database creation

The generated config.yml file already contains the information to a local sqlite database called `outflow.db` that will live in the root directory of the pipeline.

Outflow will use this database to store the tasks and their executions. Our plugin models will also create tables in this database.

2.5.2 Models

Outflow models are sqlalchemy models integrated with the framework. You may already know the concept of models from Django or Flask. If not, the basic concept and use case will be explained here. See the sqlalchemy documentation for more information.

What is a model

Models are python classes that represent your database layout. Either for querying, or even to create it (using migrations). Each model represent a database table, its constraints and relationships. Let's uncomment the example model in *model.py* in the directory models of our plugin :

```
from sqlalchemy import Column, INTEGER
from outflow.core.db import Model


class MyTable(Model):
    id_my_table = Column(INTEGER(), primary_key=True)
    my_column = Column(INTEGER(), nullable=False, unique=True)
```

As you can see, there is not much difference with a standard sqlalchemy model.

- The baseclass for all your models must be `outflow.core.db.Model class`.
- The table name in the database is defaults to the class name in snake case (i.e. "MyTable" -> "my_table"). To override the table name, set the __tablename__ class attribute.

Each model has a number of class variables, each of these model *fields* represent a database table *column*.

Each field is represented by an instance of a sqlalchemy.Column class – e.g., This tells sqlalchemy what type of data each field holds.

The name of each Column instance (e.g. my_column) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

Writing a model for our plugin

In our data_reduction plugin, we have a task that makes some computation on the generated data and a command line argument. Let's assume this computation is very big and takes time. We could use the database to cache the result of the computation, so that a pipeline execution with the same input data won't need to compute the result again.

To achieve this, let's modify the example model to match our needs :

```
from sqlalchemy import Column, INTEGER, UniqueConstraint
from outflow.core.db import Model

class ComputationResult(Model):
    id_computation_result = Column(INTEGER(), primary_key=True)
    input_value = Column(INTEGER(), nullable=False)
    multiplier = Column(INTEGER(), nullable=False)
    result = Column(INTEGER(), nullable=False)

    __table_args__ = (UniqueConstraint("input_value", "multiplier"),)
```

If you wish, rename the file *model.py* into *computation_result.py*

Create the database layout from models

We now have a model that could record the computation result. However, our database is still empty: no data of course but also not even tables.

The next step is to generate migrations for our plugin.

Migrations ?

Migrations are python scripts that execute SQL queries that modify the database layout. They are very useful to keep your data intact when you want to modify your database layout. Migrations should be added to your versioning system.

Outflow integrates alembic to automatically generate migrations of your plugins. Alembic compares the layout of the database with the models, and will generate the migration that will modify the database to match the current models.

Generate the migration

Before generating new migrations, you have to apply the initial Outflow migrations that creates tables needed for the internals of the framework:

```
python manage.py management db upgrade heads
```

To generate the migration that creates our table computation_result, type the following command :

```
python manage.py management db make_migrations --plugin tuto.data_reduction --message=
↪"Add table 'computation_result'"
```

This will create a new file in the directory *models/versions/default* of your plugins named something like *ddaf105ff7b5_add_table_computation_result.py*. Let's open it and take a look:

```
"""Add table 'computation_result'

Revision ID: a0d8cd78f167
Revises:
Create Date: 2020-11-18 16:21:37.862087

"""
from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = "a0d8cd78f167"
down_revision = None
branch_labels = ("tuto.data_reduction",)
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table(
        "computation_result",
        sa.Column("id_computation_result", sa.INTEGER(), nullable=False),
```

(continues on next page)

(continued from previous page)

```
        sa.Column("input_value", sa.INTEGER(), nullable=False),
        sa.Column("multiplier", sa.INTEGER(), nullable=False),
        sa.Column("result", sa.INTEGER(), nullable=False),
        sa.PrimaryKeyConstraint("id_computation_result"),
        sa.UniqueConstraint("input_value", "multiplier"),
    )
    op.grant_permissions("computation_result")
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_table("computation_result")
    # ### end Alembic commands ###
```

You can see that alembic generated a python script with two functions: upgrade and downgrade. As you probably guessed, they correspond to the modifications needed to go to one revision to the other (either up or down). Alembic's name for migration is *revision*.

Alembic is an amazing tool but it is not perfect. There are some changes that it cannot detect, because it is not always possible to guess what the user wants from two different models (for example it cannot distinguish between renaming a table or deleting it and creating an new one). For this reason, you should always check the migrations generated by alembic and edit it if needed.

Apply the migration

To apply the generated migration, call the following command :

```
$ python manage.py management db upgrade heads
```

heads is a shortcut to the last migration of the database.

2.5.3 A word about the pipeline context

To access the database, you will need to use the pipeline context.

The pipeline context is an object that contains:

- the settings and configuration of your pipeline (see settings and configuration)
- the command line arguments
- a reference to the database session

You can *import* the pipeline context anywhere, but you can only *access* it within the pipeline scope : i.e. inside the code of your tasks and inside the setup_tasks() method of the command.

To access the database inside the task, we will use the `context.session` object.

2.5.4 Access our database table

Before trying to access our models, we need to edit our command. You may have noticed the argument *db_untracked=True* in the command definition. This arguments allows to run commands without any databases. Now that we have set up a database, go to *commands.py* and remove this argument:

```
# remove the argument  ↓
@RootCommand.subcommand()
class ComputeData(Command):
    def setup_tasks(self):
        ...
```

Now that our database is in sync with our model, we can start inserting and querying the newly created table. We will edit the task `Compute` to check if the computation was already done before, and if not do it and insert it in the database.

As you may remember from the last chapter, the database session is in the pipeline context :

```
import time

from outflow.core.logging import logger
from outflow.core.tasks import Task
from outflow.core.pipeline import context
from sqlalchemy.orm.exc import NoResultFound

from .models.computation_result import ComputationResult


@as_task
def Compute(raw_data: int) -> {"computation_result": int}:
    # get the session of the default database
    session = context.session

    multiplier = context.args.multiplier

    # Check if the result of the computation is already in the database
    try:
        # query the database with our model ComputationResult
        computation_result_obj = session.query(ComputationResult) \
            .filter_by(input_value=raw_data, multiplier=multiplier).one()

        logger.info("Result found in database")

        # get the result from the model object (ie from the row in the table)
        computation_result = computation_result_obj.result

    except NoResultFound:
        # Result not in the database: compute the value like before
        logger.info("Result not found in database, computing result and inserting")

        # simulate a big computation
        time.sleep(3)
        computation_result = raw_data * multiplier

        # create an object ComputationResult
```

(continues on next page)

(continued from previous page)

```
        computation_result_obj = ComputationResult(
            input_value=raw_data,
            multiplier=multiplier,
            result=computation_result
        )

        # and insert it in the database
        session.add(computation_result_obj)
        session.commit()

    # return the result for the next task
    return {"computation_result": computation_result}
```

Now, run our command `python manage.py compute_data --multiplier 2` multiple times and you will see that the second time and afterward, the execution will be much faster since we already have the result cached in the database.

This is the end of part 3 of the tutorial. In this chapter, you have learned how to configure access to the database, create models and migrations, and finally how to query the database using your models.

The next chapter is about the most exciting feature of outflow: parallelization of workflows on multiple cores, and on multiple cluster nodes!

2.6 Create a pipeline with Outflow, part 6: Products

2.7 Create a pipeline with Outflow, part 5: Testing

We've built a basic pipeline, and we'll now create some automated tests for it.

2.7.1 Introducing automated testing

What are automated tests?

Tests are routines that check the operation of your code.

Testing operates at different levels. Some tests might apply to a tiny detail (does a particular task method return values as expected?) while others examine the overall operation of the software (does a workflow (a sequence of task) produce the desired result?). That's no different from the kind of testing you did earlier in Models and database section, using the shell command to examine the behavior of a task in the pipeline context.

What's different in automated tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your plugin, you can check that your code still works as you originally intended, without having to perform time consuming manual testing.

Why you need to create tests

So why create tests, and why now?

Tests will save you time

Up to a certain point, *checking that it seems to work* will be a satisfactory test. In a more sophisticated plugin, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the plugin's behavior. Checking that it still *seems to work* could mean running through your code's functionality with twenty different variations of your test data to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - even if you hadn't even realized it had gone wrong.

Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will refuse to look at it because it lacks tests; without tests, they won't trust it.

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

Tests help teams work together

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that colleagues don't inadvertently break your code (and that you don't break theirs without knowing).

2.7.2 Basic testing strategies

There are many ways to approach writing tests.

Some programmers follow a discipline called test-driven development; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development formalizes the problem in a Python test case.

More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it's never too late to get started.

Sometimes it's difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it's fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

So let's do that right away.

2.7.3 Tutorial code modifications

For this part of the tutorial, we have modified the code in order to be able to test parallelized and non-parallelize code.

Tasks written at the end of Part 3 are renamed into `ComputeOneData`, `GenOneData` and `PrintOneData`. Functions at the end of part 4 are `ComputeMoreData`, `GenMoreData` and `PrintMoredata`.

Commands are also renamed in to `compute_one_data` and `compute_more_data` :

```
$ python manage.py compute_one_data --multiplier 3
tuto.data_reduction.tasks - tasks.py:91 - INFO - Result of the computation: 126
$ python manage.py compute_more_data --multiplier 3
tuto.data_reduction.tasks - tasks.py:45 - INFO - Result found in database
tuto.data_reduction.tasks - tasks.py:55 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:55 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:55 - INFO - Result not found in database, computing␣
↪result and inserting
tuto.data_reduction.tasks - tasks.py:96 - INFO - Result of the mapped computation: [[{
↪'computation_result': 126}], [{'computation_result': 129}], [{'computation_result':␣
↪132}], [{'computation_result': 135}]]
```

This way, we will learn how to test tasks and commands, with and without parallelized workflow.

2.7.4 Pytest add-ons and configuration

pytest-cov

This will display which percentage of your code your tests cover. The higher it is, the better it is. This cannot be the only indicator to take into account : you can make tests that will execute every line of code, but if you do not make wise `assert`, it will be useless. But this can highlight that some parts have been forgotten into you tests. `pip install pytest-cov` if needed.

The configuration file is `.coveragerc`. You should create this file with

```
[run]
omit = plugins/data_reduction/tuto/data_reduction/tests/*
```

This will exclude tests from the coverage (I wonder why it is not the case by default).

Output will be written in a directory named `htmlcov`.

pytest-postgresql

This package will provide fixtures needed to perform tests with a real running Postgresql database. This will be used for command tests. It should be installed with `outflow`.

pytest-sugar

Pretty-printing for the tests output. It is not mandatory but I like it .

pytest.ini

```
[pytest]
# This will set the log level to display. Here INFO
log_cli = True
log_cli_level = INFO
log_level = INFO
# This will be used by pytest-postgresql to connect to the database
postgresql_user = flo
postgresql_password = xxx
postgresql_host = localhost
```

2.7.5 Testing simple tasks

We identify a bug

`GenOneData` and `GenMoreData` are written to return respectively `42` and `[42, 43, 44, 45]`. Suppose that was erroneous. They should return `40` and `[40, 50, 60, 70]`.

Create a test to expose the bug

Conventional places for an application's tests is in the plugin's `tests.py` file or in the `tests/` directory; the testing system will automatically find tests in any file whose name begins with `test`.

Put the following in the `tests/test_1_gen.py` file in the `data_reduction` application:

```
from outflow.core.test import TaskTestCase

class TestDataReductionGenTasks(TaskTestCase):

    def test_gen_one(self):
        from tuto.data_reduction.tasks import GenOneData

        # --- initialize the task ---
        self.task = GenOneData()
        self.config = {}
```

(continues on next page)

(continued from previous page)

```
        # --- run the task ---
        result = self.run_task()

        # --- make assertions ---
        # test the result
        assert isinstance(result, dict)
        assert 'some_data' in result
        assert result == {'some_data': 40}
```

This will test that:

- the result is a dictionary
- it has the key `some_data`
- the value of `some_data` is 40

Here we have created a `outflow.core.test.TaskTestCase` subclass. The task is run without arguments.

Running tests

In the terminal, we can run our test in the pipeline root directory using:

```
$ pytest
```

and you'll see something like:

```
$ pytest
collecting ...

_____ TestDataReductionGenTasks.test_gen_one _____
plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.py:17: in test_gen_one
    assert result == {'some_data': 40}
E   AssertionError: assert {'some_data': 42} == {'some_data': 40}
E     Differing items:
E     {'some_data': 42} != {'some_data': 40}
E     Full diff:
E     - {'some_data': 40}
E     ?                ^
E     + {'some_data': 42}
E     ?                ^

 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks.test_gen_one    100%
===== short test summary info =====
FAILED plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks::test_gen_one -
AssertionError: assert {'some_data': 42} == {'some_data': 40}

Results (0.78s):
       1 failed
         - plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.py:5␣
↪TestDataReductionGenTasks.test_gen_one
```

What happened is this:

- `pytest` looked for tests in the `data_reduction` plugin
- it found a class whose name starts with "Test"
- it looked for test methods - ones whose names begin with `test`
- just before executing `test_gen_one`, TaskTestCase creates a pipeline context
- … and using the `assert` statement, it checked the tested feature

The test informs us which test failed and even the line on which the failure occurred.

You can make as many `assert` in you test as you want. If one fails, all the test will be reported as failed, and the following assertions inside this test will not be evaluated.

Fixing the bug

Replace 42 by 40 in the `tasks.py` file. Run the test again :

```
$ pytest --tb=short plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks::test_gen_one
collecting ...
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks.test_gen_one   100%

Results (0.66s):
       1 passed
```

Repeat for `GenMoreData`

Do the same for testing `GenMoreData` : create a test `test_gen_more` that will raise that `GenMoreData` does not return the expected values.

```
$ pytest
collecting ...
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks.test_gen_one ✓      50%
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.
↪py::TestDataReductionGenTasks.test_gen_more ✓    100%

Results (0.65s):
       2 passed
```

2.7.6 Testing complex tasks

The tasks `ComputeOneData` and `ComputeMoreData` are complex tasks because they involve a python code and an interaction with a database. As we do not want to test the database connection, we will mock the database responses.

Testing ComputeOneData

Create a new file named `tests/test_2_compute_one.py`. The numbering is not mandatory but tests are executed in the alphabetic order of the file names. I like to test things in a logical order : it allows to break tests early if one fails and if I know that the other ones will fail in a same manner.

The complete code will be shown at the end of the section

We will mock the session linked to the database with a test decorator :

```
@mock.patch('outflow.core.db.database.Databases.session', new_callable=mock.PropertyMock)
def test_compute_one_already_in_db(self, mock_session):
       (...)
```

Every request to the database will return silently as if everything is ok.

Testing when result is already in database

To simulate the query (in `tasks.py`):

```
computation_result_obj = session.query(ComputationResult) \
   .filter_by(input_value=some_data, multiplier=multiplier).one()
```

we will declare the `ComputationResult` that we are waiting for:

```
db_result = ComputationResult(
   input_value = data,
   multiplier = mult,
   result = data * mult
)
```

and tell that this will be the returned value of `session.query.filter_by.one`:

```
mock_session\
   .return_value.query\
   .return_value.filter_by\
   .return_value.one.return_value = db_result
```

As the task is running outside the command, some context has to be defined :

```
from outflow.core.pipeline import context, config

context.force_dry_run = False
context.db_untracked = False
context._models = []
config["databases"] = mock.MagicMock()
context.args = Namespace(multiplier = mult, dry_run = False, db_untracked = False)
```

Like before, setup and run the task :

```
self.task = ComputeOne()
result = self.run_task(some_data = data)
```

We can then make the needed assertions:

```
assert isinstance(result, dict)
assert 'computation_result' in result
assert result == {'computation_result': data * mult}
```

As the query to the database returned a `ComputationResult` this means that the result was already in database. Then no insertion should have been made. To ensure this was the case, we can examine the call traceback:

```
# filtering "add" calls with a ComputationResult object as parameter
call_add_computation_result = [
    call.args[0]
    for call in mock_session.return_value.add.call_args_list
    if isinstance(call.args[0], ComputationResult)]
```

`call_args_list` returns all the `add` calls that have been made to the session. We filter the results because outflow uses also the session to log each job into the tables `public.task`. We are only interested in the `add` into the table `computation_result`.

For this test, no `ComputationResult` should be added in database. Then :

```
assert len(call_add_computation_result) == 0
```

Testing when the result is not in database

In this case, the query should not return a `ComputationResult` object but instead raise a `NoResultFound` exception :

```
from sqlalchemy.orm.exc import NoResultFound
mock_session\
    .return_value.query\
    .return_value.filter_by\
    .return_value.one.side_effect = NoResultFound
```

Note that we use the method `side_effect` instead of `return_value` for a function.

The same assertions can be made for this test, but for the `len(call_add_computation_result)` the expected value will be different as the `ComputationResult` should have been inserted in database :

```
assert len(call_add_computation_result) == 1
```

At this stage, `pytest` should output :

```
$ pytest
collecting ...
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.py ✓✓            50%
 plugins/data_reduction/tuto/data_reduction/tests/test_2_compute_one.py ✓✓   100%

Results (4.30s):
       4 passed
```

The output may differ depending on your `pytest.ini` (here `log_cli = False`).

Complete code

```
import pytest
from unittest import mock
from random import sample, randrange
from argparse import Namespace

from outflow.core.test import TaskTestCase
from outflow.core.pipeline import context, config

from tuto.data_reduction.tasks import ComputeOne
from tuto.data_reduction.models.computation_result import ComputationResult

class TestDataReductionComputeTasks(TaskTestCase):

    @pytest.fixture(autouse=True)
    def setup_context(self, with_pipeline_context_manager):
        context.force_dry_run = False
        context.db_untracked = False
        context._models = []
        config["databases"] = mock.MagicMock()

    @mock.patch('outflow.core.db.database.Databases.session', new_callable=mock.
↪PropertyMock)
    def test_compute_one_already_in_db(self, mock_session):

        # --- define test data ---
        data = randrange(100)
        mult = randrange(10)
        db_result = ComputationResult(
            input_value = data,
            multiplier = mult,
            result = data * mult
        )

        # --- define mock session queries
        mock_session\
            .return_value.query\
            .return_value.filter_by\
            .return_value.one.return_value = db_result

        # -- define args
        context.args = Namespace(multiplier = mult, dry_run = False, db_untracked =␣
↪False)
        self.task = ComputeOne()

        # --- run the task ---
        result = self.run_task(some_data = data)

        # filtering "add" calls with a ComputationResult object as parameter
        call_add_computation_result = [
            call.args[0]
```

(continues on next page)

(continued from previous page)

```
            for call in mock_session.return_value.add.call_args_list
            if isinstance(call.args[0], ComputationResult)]

        # --- make assertions ---
        assert isinstance(result, dict)
        assert 'computation_result' in result
        assert result == {'computation_result': data * mult}
        assert len(call_add_computation_result) == 0

    # @mock.patch('tuto.data_reduction.tasks.context')
    @mock.patch('outflow.core.db.database.Databases.session', new_callable=mock.
↪PropertyMock)
    def test_compute_one_not_in_db(self, mock_session):

        # --- initialize the task ---
        from sqlalchemy.orm.exc import NoResultFound

        # --- define test data ---
        data = randrange(100)
        mult = randrange(10)
        db_result = ComputationResult(
            input_value = data,
            multiplier = mult,
            result = data * mult
        )

        # --- define mock session queries
        mock_session\
            .return_value.query\
            .return_value.filter_by\
            .return_value.one.side_effect = NoResultFound

        # -- define args
        context.args = Namespace(multiplier = mult, dry_run = False, db_untracked =␣
↪False)
        self.task = ComputeOne()

        # --- run the task ---
        result = self.run_task(some_data = data)

        # filtering "add" calls with a ComputationResult object as parameter
        call_add_computation_result = [
            call.args[0]
            for call in mock_session.return_value.add.call_args_list
            if isinstance(call.args[0], ComputationResult)]

        # --- make assertions ---
        assert isinstance(result, dict)
        assert 'computation_result' in result
        assert result == {'computation_result': data * mult}
        assert len(call_add_computation_result) == 1
```

Testing ComputeMoreData

Testing `ComputeMoreData` will be similar to `ComputeOneData`. We have to tell to the mocked session the array of values that will be returned by `session.query.filter_by.one`.

Testing when all the data are already in database

Define the values to be returned :

```
db_result_list = [ComputationResult(
   input_value = data_array[i],
   multiplier = mult,
   result = data_array[i] * mult
) for i in range(nb_data) ]
```

where `data_array` is the array that will be given to the task, and `nb_data` the length of this array.

Give them to the mocked session :

```
mock_session\
   .return_value.query\
   .return_value.filter_by\
   .return_value.one.side_effect = db_result_list
```

For testing the values returned by the task, we cannot make a simple comparison because the different tasks distributed on different CPUs may not return in the same order.

We have then to test:

- the length of the result array is `nb_data`
- each expected returned value is in `result`

```
assert len(result['map_computation_result']) == nb_data

for i in range(nb_data):
   data = data_array[i]
   assert [{'computation_result': data * mult}] in result['map_computation_result']
```

Finally, don't forget to test that no `add` calls were made to insert `ComputationResult` in database :

```
assert len(call_add_computation_result) == 0
```

Testing when some data are not in database

For this case, just replace one `ComputationResult` by a `NoResultFound`. For example :

```
from random import randrange
item_not_in_db = randrange(nb_data)
db_result_list[i] = NoResultFound
```

And in this case, there should be 1 `add` call to insert `ComputationResult` in database :

```
assert len(call_add_computation_result) == 1
```

Finally, pytest will return:

```
$ pytest
collecting ...
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.py ✓✓              33%
 plugins/data_reduction/tuto/data_reduction/tests/test_2_compute_one.py ✓✓      67%
 plugins/data_reduction/tuto/data_reduction/tests/test_3_compute_more.py ✓✓    100%

Results (12.72s):
       6 passed
```

Complete code

```
import pytest
from random import sample, randrange
from argparse import Namespace
from unittest import mock

from outflow.core.test import TaskTestCase
from outflow.library.workflows import MapWorkflow

from outflow.core.pipeline import context, config

from tuto.data_reduction.models.computation_result import ComputationResult
from tuto.data_reduction.tasks import ComputeMore


class TestDataReductionComputeMoreTasks(TaskTestCase):

    @pytest.fixture(autouse=True)
    def setup_context(self, with_pipeline_context_manager):
        context.force_dry_run = False
        context.db_untracked = False
        context._models = []
        config["databases"] = mock.MagicMock()
        config["backend"] = "ray"
        config["ray"] = {"cluster_type": "local"}

    @mock.patch('outflow.core.db.database.Databases.session', new_callable=mock.
↪PropertyMock)
    def test_compute_more(self, mock_session):
        """
        Testing when all data are already in database
        Request to DB returns a ComputationResult
        """

        # --- define test data ---
        nb_data = randrange(1, 5)
        data_array = sample(range(100), k=nb_data)
```

(continues on next page)

(continued from previous page)

```
        mult = randrange(10)

        # --- define the data returned by the request to the DB ---
        db_result_list = [ComputationResult(
            input_value=data_array[i],
            multiplier=mult,
            result=data_array[i] * mult
        ) for i in range(nb_data)]

        # --- define mock session queries
        mock_session
            .return_value.query
            .return_value.filter_by
            .return_value.one.side_effect = db_result_list

        # -- define args
        context.args = Namespace(multiplier=mult, dry_run=False, db_untracked=False)

        # --- initialize the task ---
        mapped_computation = MapWorkflow(ComputeMore(), output_name="map_computation_
↪result")
        self.task = mapped_computation

        # --- run the task ---
        result = self.run_task(some_data_array=data_array)

        # --- make assertions ---

        # filtering "add" calls with a ComputationResult object as parameter
        call_add_computation_result = [
            call.args[0]
            for call in mock_session.return_value.add.call_args_list
            if isinstance(call.args[0], ComputationResult)]

        # all results are already be in base, since we mock a result in base
        assert len(call_add_computation_result) == 0

        # check the type of result
        assert isinstance(result, dict)
        assert 'map_computation_result' in result

        # we should get as many results as input data
        assert len(result['map_computation_result']) == nb_data

        # verifying that every expected result is returned
        for i in range(nb_data):
            data = data_array[i]
            assert [{'computation_result': data * mult}] in result['map_computation_
↪result']

    @mock.patch('outflow.core.db.database.Databases.session', new_callable=mock.
↪PropertyMock)
```

(continues on next page)

(continued from previous page)

```
    def test_compute_more_with_results_not_in_base(self, mock_session):
        """
        Testing when some data are not already in database
        Request to DB returns a ComputationResult or a NoResultFound exception
        """

        # --- define test data ---
        nb_data = randrange(2, 5)
        nb_data_not_in_db = randrange(1, nb_data)
        data_not_in_db = sample(range(nb_data), k=nb_data_not_in_db)
        data_array = sample(range(100), k=nb_data)
        mult = randrange(10)
        db_result_list = []

        # --- define the data returned by the request to the DB ---
        from sqlalchemy.orm.exc import NoResultFound
        for i in range(nb_data):
            if i in data_not_in_db:
                db_result_list.append(NoResultFound)
            else:
                db_result_list.append(ComputationResult(
                    input_value=data_array[i],
                    multiplier=mult,
                    result=data_array[i] * mult
                ))

        # --- define mock session queries
        mock_session
            .return_value.query
            .return_value.filter_by
            .return_value.one.side_effect = db_result_list

        # -- define args
        context.force_dry_run = False
        context.db_untracked = False
        context._models = []
        config["databases"] = mock.MagicMock()
        config["backend"] = "ray"
        context.args = Namespace(multiplier=mult, dry_run=False, db_untracked=False)

        # --- initialize the task ---
        mapped_computation = MapWorkflow(ComputeMore(), output_name="map_computation_
↪result")
        self.task = mapped_computation

        # --- run the task ---
        result = self.run_task(some_data_array=data_array)

        # --- make assertions ---

        # filtering "add" calls with a ComputationResult object as parameter
        call_add_computation_result = [
```

(continues on next page)

(continued from previous page)

```
            call.args[0]
            for call in mock_session.return_value.add.call_args_list
            if isinstance(call.args[0], ComputationResult)]

        # nb_data_not_in_db should be added since they were not found
        assert len(call_add_computation_result) == nb_data_not_in_db

        # check the type of result
        assert isinstance(result, dict)
        assert 'map_computation_result' in result

        # we should get as many results as input data
        assert len(result['map_computation_result']) == nb_data

        # verifying that every expected result is returned
        for i in range(nb_data):
            data = data_array[i]
            assert [{'computation_result': data * mult}] in result['map_computation_
↪result']
```

2.7.7 Testing commands

The aim of testing command is to verify that tasks are executed one after the other as defined. The cases tested at the tasks level do not need to be tested again.

Regular commands

In the first part of the tutorial, we defined a command `data_reduction` which only prints a "Hello world". There is no many things to test for this command, but it will give the skeleton for testing a command :

```
class TestDataReductionNoCmd(CommandTestCase):
    def test_data_reduction(self):

        # --- initialize the command ---
        from tuto.data_reduction.commands import DataReduction
        self.root_command_class = DataReduction
        arg_list = []

        # --- run the command ---
        return_code, result = self.run_command(arg_list)

        # --- make assertions ---
        assert return_code == 0
        assert result == [{"None": None}]
```

Commands involving an interaction with a Postgresql database

Outflow comes with a `PostgresCommandTestCase` class that will setup a fresh database for us before each test and that will drop it after use.

Create a new file test `test_4_commands.py`.

```
from outflow.core.test.test_cases import (PostgresCommandTestCase,
                                          postgresql_fixture)

class TestDataReductionCmd(PostgresCommandTestCase):

    PLUGINS = ['outflow.management', 'tuto.data_reduction']
```

In order to be able to use the database, migrations has to be applied before each test (since a new database is created each time).

We will use a fixture to do this automatically (that will prevent duplicated code).

```
    # Automatically run a db upgrade heads before each test
    @pytest.fixture(autouse=True)
    def setup_db_upgrade(self, with_pipeline_context_manager):
        # -- commands to be executed
        db_upgrade = ['management', 'db', 'upgrade', 'heads', '-ll', 'INFO']
        self.run_command(db_upgrade, force_dry_run = False)
```

Testing the migration

```
# --- test if the upgrade is ok
def test_db_upgrade(self):
   with self.pipeline_db_session() as session:
      # The table computation_result has be to created
      try:
         c = session.query(ComputationResult).count()
         # and it should be empty
         assert c == 0
      except Exception as e:
         assert False, e
```

To verify that this test is relevant, comment the line `self.run_command(db_upgrade)` in the `setup_db_upgrade()` function.

`pytest` will then fail with :

```
FAILED plugins/data_reduction/tuto/data_reduction/tests/test_4_commands.
↪py::TestDataReductionCmd::test_db_upgrade -
AssertionError: ProgrammingError('(psycopg2.errors.UndefinedTable) relation "computation_
↪result" does not exist
```

This way, you are sure that if the migration fails for any reason, you will be notified. Do not forget to comment out the line `self.run_command(db_upgrade)` afterwards.

Testing `compute_one_data`

Testing this command is easy. Just define the command:

```
command = [
    'compute_one_data',
    '--multiplier',
    f'{multiplier}',
    '-ll',
    'INFO',
]
```

Run the command:

```
return_code, result = self.run_command(command, force_dry_run = False)
```

And test we get the expected results:

```
assert return_code == 0
assert result[0]['computation_result'] == 40 * multiplier
```

Testing `compute_more_data`

For this command, the Ray (TODO change to parallel backend) backend has to be activated. Before running the command, configuration has to be updated :

```
custom_config = {
    "backend": "ray",
    "ray": {"cluster_type": "local"}
}
config.update(custom_config)
```

The other parts of the tests remain the same : assert the `return_code` and that each expected value is present in the result array :

```
assert return_code == 0
for i in range(40, 71, 10):
   res = [{'computation_result': i * multiplier}]
   assert res in result[0]['map_computation_result']
```

Finally, `pytest` should return:

```
$ pytest
collecting ...
 plugins/data_reduction/tuto/data_reduction/tests/test_1_gen.py ✓✓            20%
 plugins/data_reduction/tuto/data_reduction/tests/test_2_compute_one.py ✓✓    40%
 plugins/data_reduction/tuto/data_reduction/tests/test_3_compute_more.py ✓✓   60%
 plugins/data_reduction/tuto/data_reduction/tests/test_4_commands.py ✓✓✓✓     100%

Results (22.31s):
      10 passed
```

Congratulations !

2.7.8 Test coverage

In addition you can check the coverage of your tests running `pytest --cov=tuto.data_reduction --cov-report html`. This will generate an HTML report in the `htmlcov/` directory.

With this tutorial, you should be able to reach 100% of coverage.

| Module | statements | missing | excluded | coverage | | ——————————————————————————————- | ———- | ——- | ——— | ——— | | Total | 96 | 0 | 0 | 100% | | `plugins/data_reduction/tuto/data_reduction/__init__.py` | 0 | 0 | 0 | 100% | | `plugins/data_reduction/tuto/data_reduction/commands.py` | 30 | 0 | 0 | 100% | | `plugins/data_reduction/tuto/data_reduction/models/__init__.py` | 7 | 0 | 0 | 100% | | `plugins/data_reduction/tuto/data_reduction/models/computation_result.py` | 8 | 0 | 0 | 100% | | `plugins/data_reduction/tuto/data_reduction/tasks.py` | 51 | 0 | 0 | 100% |

CHAPTER

THREE

USER GUIDE

3.1 How to install Outflow

3.1.1 Install Python

System requirements

- Python 3.7 and up
- Any linux distribution or Windows

It is recommended to have a virtual environment per pipeline. For example :

```
$ python3 -m venv ~/.virtualenvs/my_pipeline
$ source ~/.virtualenvs/my_pipeline/bin/activate
```

3.1.2 Install the Outflow code

Outflow is release on PyPi, so inside your virtual environment, you can just use:

```
(my_pipeline) $ pip install outflow
```

3.2 Tasks

Tasks are the smallest building blocks of your pipelines.

3.2.1 Create a task from a function

The `as_task` decorator allows you to convert a function into a Task subclass, that you can use in your workflow definition just like regular tasks defined above. You can use this decorator both with and without parenthesis.

```
from outflow.core.tasks import as_task

@as_task
def MyTask(a, b):
    return {"result": a+b}
```

3.2.2 Create a task by subclassing Task

```
from outflow.core.tasks import Task

class MyTask(Task):
    def run(self, a: int, b: int) -> {"result": int}:
        return {"result": a+b}
```

3.2.3 Create a Task template

If you need a special behaviour for each of your task, you can use the `as_task` method of your subclass on your tasks. Here is an example with tasks that will always log their name before running.

```
from outflow.core.tasks import Task
from outflow.core.logging import logger


class LogTask(Task):
    def __call__(self, *args, **kwargs):
        logger.info(f"Running task {self.name}")
        return super().__call__(*args, **kwargs)

@LogTask.as_task
def MyTask():
    pass
```

3.2.4 Running a task directly

You might want to manually run a task, for example if you are in the outflow shell. In this case, you can call a task instance manually, and passing all needed task inputs as keyword arguments of this call.

Inside a python shell

```
>>> @as_task
... def Add(a:int, b:int) -> {"result": int}:
...     return a+b
>>> add = Add()
>>> add(2,4)
{"result": 6}
```

3.2.5 Targets

Targets represent the input and outputs of your tasks. There are 3 types of targets, inputs, outputs, and parameters.

Target typing

In all target definition syntax, the type is optional (but highly recommended) and defaults to typing.Any.

Parameters

Parameters are special inputs that comes from the pipeline configuration file. At the task execution, outflow will look for the name of the task in the `parameter` field of the configuration file. Note that by default, the task name is the snake_case name of the decorated function. A parameter must be present in the configuration file and cannot have a default value.

3.2.6 Task definition alternatives

Let's take the example of a task that compute the sum of two numbers a and b, and returns the result :

```
from outflow.core.tasks import Task

class MyTask(Task):
    def setup_targets(self):
        self.add_input("a", int)
        self.add_input("b", int)
        self.add_output("result", int)

    def run(self, a, b):
        return {"result": a+b}
```

Class Task have a method `setup_targets` that is automatically called by the framework. As shown in the above example, you can override this method to add the targets. The default method will try to guess your targets from the run function annotations.

You can have a hybrid approach, by calling `super().setup_targets()` and then manually set up additional targets.

Automatic inputs

```
from outflow.core.tasks import Task

class MyTask(Task):
    def run(self, a: int, b: int):
        return {"result": a+b}
```

Automatic output targets from annotations

Outflow will make output targets from the return annotations. This is the recommended way since you can type your outputs.

```
from outflow.core.tasks import Task

class MyTask(Task):
    def setup_targets(self):
        self.add_input("a", int)
        self.add_input("b", int)
        super().setup_targets() # use automatic output detection
        # self.add_output("result", int)

    def run(self, a, b) -> {"result": int}:
        return {"result": a+b}
```

Automatic output targets from return statement

If Outflow does not find a return annotation, it will try to check your `run` method for the return statement, and recover the dictionary to automatically add the output targets to the task. In this case the type of each target will be typing.Any.

```
from outflow.core.tasks import Task

class MyTask(Task):
    def setup_targets(self):
        self.add_input("a", int)
        self.add_input("b", int)
        super().setup_targets() # use automatic output detection from return statement
        # self.add_output("result", int)

    def run(self, a, b):
        return {"result": a+b}
```

Implicit return

If your task has **only one** output, and the type of this output is not a dictionary, it is possible to return the value directly from the `run` method.

```
from outflow.core.tasks import Task

class MyTask(Task):
    def setup_targets(self):
        self.add_input("a", int)
        self.add_input("b", int)
        self.add_output("result", int)

    def run(self, a, b):
        return a+b
        # return {"result": a+b}
```

3.3 Workflows

In Outflow, Workflows and tasks are both what we call Blocks. They both share the concepts of targets (inputs and outputs, parameters are for tasks only), and they both have a `run()` method.

Tasks `run()` is the function that you decorate (or that you specify in your subclass). For Workflows, the default `run()` behaviour is to run the sequence of tasks it contains. MapWorkflows are special workflows, their `run()` method is to run its containing task sequence as many times as their input they are mapped on.

3.3.1 Workflow definition

There are multiple ways to define workflows :

Using the as_workflow decorator

```
from outflow.core.workflow import as_workflow

@as_workflow
def process():
    Extract() | Transform() | Load()


@RootCommand.subcommand()
class MyCommand(Command):
    def setup_tasks(self):
        process() # call the decorated function to instanciate and register the workflow
```

You can use the as_workflow decorator with or without parenthesis.

Using the explicit syntax

The `as_workflow` decorator is syntactic sugar for something you can do manually if you prefer. Here is the same workflow with the explicit syntax.

```

@RootCommand.subcommand()
class MyCommand(Command):
    def setup_tasks(self):
        process = Workflow()
        process.start() # start registering tasks
        # everything declared between these two calls with belong to the workflow
↪"process"
        Extract() | Transform() | Load()
        process.stop() # stop registering
```

External task in a workflow

If you have a task inside a workflow that needs the output of a task outside this workflow. For example, the Transform task need the result of a big computation that we want to run only once (for example if using a MapWorkflow).

With the decorator syntax :

```
@as_workflow
def process(big_computation):
    transform = Transform()

    big_computation | transform
    Extract() | transform | Load()


@RootCommand.subcommand()
class MyCommand(Command):
    def setup_tasks(self):
        big_computation = BigComputation()
        process(big_computation) # call the decorated function to instanciate and␣
↪register the workflow
```

With the explicit syntax :

```
@RootCommand.subcommand()
class MyCommand(Command):
    def setup_tasks(self):
        big_computation = BigComputation()

        process = Workflow()
        process.start() # start registering tasks
        # everything declared between these two calls with belong to the workflow
↪"process"
        transform = Transform()
        big_computation | transform
        Extract() | transform | Load()
        process.stop() # stop registering
```

3.3.2 Using decorator on special workflow classes

If you need special workflow behaviour, you can subclass Workflow and use its method `as_workflow`.

For example, a workflow class that will do a coin toss and run half the time :

```
class RandomWorkflow(Workflow):
    def _run(self, **inputs):
        import random

        if random.choice([True, False]):
            logger.info("Heads, I run")
            return super()._run(**inputs)
        else:
            logger.info("Tails, I sleep")
            return {output: None for output in self.outputs} # return None for all␣
↪output targets


# create a random workflow
@RandomWorkflow.as_workflow(cache=True, output_name="computation_result")
def compute_workflow():
    Extract() | Transform() | Load()
```

3.4 Commands

3.4.1 Create a command

To create a first level command, import `from outflow.core.commands import Command, RootCommand` and subclass Command and decorate with `@RootCommand.subcommand()`:

```
from outflow.core.commands import Command, RootCommand

@RootCommand.subcommand()
class MyCommand(Command):

    def setup_tasks(self):
        # instantiate tasks and setup the workflow
```

This command will be available at :

```
$ python manage.py my_command
```

3.4.2 Create subcommands

If you want to create subcommands of a common command (think `git clone git commit`), first create a non-invokable top level command:

```
@RootCommand.subcommand(invokable=False)
class Git(Command):
    pass
```

This command will print its help if called directly.

Then, create a subcommand of the previous one.

```
@Git.subcommand(invokable=False)
class Clone(Command):
    def setup_tasks(self):
        ...
```

3.4.3 Built-in commands

Outflow ships with a bunch of useful commands, available through the `management` command :

```
$ python -m outflow management ...
or
$ python manage.py management ...
```

(ShellCommand)=

shell

`python manage.py management shell`

This command will execute an `IPythonTask`. This is useful for development and debugging, because you are inside an outflow pipeline so you have acces to everything you would in a pipeline execution : the pipeline context, the database session, the config and settings. You can also import tasks and execute them like so (if you provide them with the expected inputs)

```
In [1]: from namespace.plugin.tasks import FirstTask, SecondTask
In [2]: first_task = FirstTask() # instanciate the task
In [3]: my_input = 42
In [4]: first_task_result = first_task(input1=my_input) # call the task with the inputs␣
↪as kwargs
In [5]: second_task = SecondTask()
In [6]: second_task(first_task_result) # tasks return dictionaries so you can call the␣
↪next task of the workflow directly with the result of the previous task
```

display_config

`python manage.py management display_config`

Prints the path of the configuration file.

3.5 Database

3.5.1 The outflow database

Outflow has a bunch of tables, used for internal bookkeeping but that can be useful for users. Among other, outflow records the pipeline execution (called runs) and every task inside them, their state (pending, success, failed or skipped), the exception if one occurred, and the configuration file used for a given run.

You can query these tables using the usual syntax :

```
from outflow.management.models.configuration import Configuration
from outflow.core.pipeline import context
context.session.query(Configuration).all()
```

This can be useful for example to check if a given input file has already been processed using the current configuration.

You can also check the state of the different runs with the more friendly interface : the outflow dashboard.

3.5.2 Connect to a postgres database

For pipeline developers

You will need to create a database and two roles (postgres lingo for a user), an admin role that will be owner of the database and have all right on both the layout (the tables structure) and the data, and a user role that only has rights on the data. This allows you to apply migrations and modify the database layout using the admin role, and your pipeline users to acces and fill the database with data. They will not be able to accidentally drop a table and all its data by wrongly applying a downgrade migration for example.

If you manage your own PostgreSQL instance with a local installation for example, here are example commands to create the roles and a database. Ask your IT to do this for you on a real PostgreSQL instance and provide you with the credentials if you can.

```
CREATE USER "pipeuser" PASSWORD 'userpwd' NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT;
CREATE USER "pipeadmin" PASSWORD 'adminpwd' NOSUPERUSER CREATEDB NOCREATEROLE INHERIT;
CREATE DATABASE outflow_tuto WITH OWNER pipeadmin;
```

For everyone

To connect to a postgres database, edit your config.yml file and replace the sqlite section with this one:

```
#  default:
#    dialect: sqlite
#    path: backend.db

databases:
```

(continues on next page)

(continued from previous page)

```
  default:
    dialect: postgresql
    address: address:port
    #    admin: admin_username:admin_password
    user: user_username:user_password
    database: database_name
```

Replace the address and database fields with adequate ones for you.

If you are a pipeline developer, you need to fill the `admin` field to apply the migrations. If you are a pipeline user, you only need the user field.

It is also possible to use the same credential in both admin and user fields, this postgres account should have admin right on the database.

Using yaml includes to define database credentials outside the config.yml

If you are not comfortable putting your database credentials inside the config.yml file (if you want to commit the config.yml file to a vcs for example), it is possible to use the `include` feature described in the [Configuration](Outflow configuration) page of the documentation. Your configuration files may look like this:

```
# Inside the config.yml
databases: !include databases.yml

# Inside the databases.yml
default:
  dialect: postgresql
  address: address:port
  # admin: admin_username:admin_password
  user: user_username:user_password
  database: database_name
```

3.5.3 Setup connection to multiple databases

Starting with 0.12 Flask-SQLAlchemy can easily connect to multiple databases. To achieve that it preconfigures SQLAlchemy to support multiple "binds".

What are binds? In SQLAlchemy speak a bind is something that can execute SQL statements and is usually a connection or engine. In Flask-SQLAlchemy binds are always engines that are created for you automatically behind the scenes. Each of these engines is then associated with a short key (the bind key). This key is then used at model declaration time to associate a model with a specific engine.

If no bind key is specified for a model the default connection is used instead (as configured in the pipeline configuration file).

NOTE

The table names must be unique across all binds.

Example Configuration

The following configuration declares two database connections. The default one as well as one other :

```
databases:
  default:
    dialect: postgresql
    address: localhost:5432
    admin: admin:pwd
    user: user:pwd
    database: outflow

  another_db:
    dialect: postgresql
    address: localhost:5432
    admin: admin:pwd
    user: user:pwd
    database: another_db
```

Inside a model file:

```
```

Referring to Binds

In your model you specify the bind to use with the **bind_key** attribute:

```
class User(Model):
    __bind_key__ = 'another_db'
    id = Column(Integer, primary_key=True)
    username = Column(String(80), unique=True)

class TableWithoutBind(Model):
    id = Column(Integer, primary_key=True)
    some_field = Column(Integer)
```

Then to query this table there is nothing in particular to do. The sqlalchemy session in the pipeline context have access to all the tables:

```
context.session.query(User.username).all()
context.session.query(TableWithoutBind.some_field).all()
```

3.6 Backend

Your outflow pipelines can run on different backends. There are currently 3 backends in outflow:

- default
- parallel
- slurm

3.6.1 Default backend

The default backend executes tasks sequentially in one python process. Tasks return dictionaries and these are passed to the next tasks as parameters, so object never leave the python context. You can pass objects as big as you want between the tasks.

3.6.2 Parallel backend

The parallel backend executes tasks inside a multiprocessing pool. It is useful for MapWorkflow.

3.6.3 Slurm backend

The slurm backend executes MapWorkflows by submitting a slurm array to a slurm cluster. See SlurmMapWorkflow for details.

3.6.4 Specify a backend

There are multiple ways to define the backend that will be used:

- in the command line with the argument –backend
- per command inside their definition
- inside the config.yml file

Priority is in this order :

command line > command definition > config.yml

The recommended way is to leave `backend: default` inside the config.yml (or remove the key), and define the preferred backend for a given command using :

```
@RootCommand.subcommand(backend="parallel")
class MyCommand(Command):
    ...
```

3.7 Migrations

With Alembic and using SQLAlchemy as the underlying engine, Outflow provides a way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your relational database.

3.7.1 The Commands

There are several commands which you will use to interact with migrations and Outflow's handling of database schema:

- `management db upgrade --database <DB-LABEL>`, which is responsible for upgrading to a later migration version.
- `management db downgrade --database <DB-LABEL>`, which is responsible for reverting to a previous migration version.
- `management db make_migrations`, which is responsible for creating new migrations based on the changes you have made to your models.

You should think of migrations as a version control system for your database schema. `make_migrations` is responsible for packaging up your model changes into individual migration files - analogous to commits - and `upgrade`/`upgrade` is responsible for applying/reverting those to your database.

This will add a `versions` folder to the `models` directory of your plugin. The contents of this folder need to be added to version control along with your other source files.

WARNING: The migration script needs to be reviewed and edited, as Alembic currently does not detect every change you make to your models. In particular, Alembic is currently unable to detect table name changes, column name changes, or anonymously named constraints. A detailed summary of limitations can be found in the Alembic autogenerate documentation. Once finalized, the migration script also needs to be added to version control.

3.7.2 Backend Support

Migrations are supported on all backends that Outflow ships with :

- PostgreSQL
- SQLite

3.8 Outflow settings

An Outflow settings file is a list of uppercase variables that will be stored in the pipeline context. This document explains how settings work and which default settings are available.

3.8.1 The basics

A settings file is just a Python module with module-level variables.

Here is the default Outflow settings:

```
import os

from outflow.core.commands import RootCommand

ROOT_DIRECTORY = os.environ.get("PIPELINE_ROOT_DIRECTORY", None)

PLUGINS = [
    "outflow.management",
]

ROOT_COMMAND_CLASS = RootCommand
```

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors.
- It can assign settings dynamically using normal Python syntax. For example:

```
MY_SETTING = [str(i) for i in range(30)]
```

It can import values from other settings files.

3.8.2 Designating the settings

When you use Outflow, the default settings module is the one located at the root of the pipeline folder.

But, you can specify which settings you're using. Do this by using an environment variable, `OUTFLOW_SETTINGS_MODULE`, or using the CLI argument `--settings`.

In both cases, the value should be in Python path syntax, e.g. `export OUTFLOW_SETTINGS_MODULE=my_project.my_settings` or `--settings my_project.my_settings`. Note that the settings module should be on the Python import search path.

3.8.3 Default settings

An Outflow settings file doesn't have to define any settings if it doesn't need to. Each setting has a sensible default value. These defaults live in the module `outflow/core/pipeline/default_settings.py`.

Here's the algorithm Outflow uses in compiling settings:

- Load settings from `default_settings.py`.
- Load settings from the specified settings file, overriding the default settings as necessary.

Note that a settings file should not import from default_settings, because that's redundant.

3.8.4 Using settings in Python code

In your Outflow pipeline, use settings by importing the object `outflow.core.pipeline.settings`. Example:

```
from outflow.core.pipeline import settings

if len(settings.PLUGINS) > 3:
    # Do something
    pass
```

Note that `outflow.core.pipeline.settings` isn't a module – it's an object. So importing individual settings is not possible:

```
from outflow.core.pipeline.settings import PLUGINS  # This won't work.
```

Also note that your code should not import from either `default_settings` or your own settings file. `outflow.core.pipeline.settings` abstracts the concepts of default settings and pipeline instance specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

3.8.5 Altering settings at runtime

You shouldn't alter settings in your applications at runtime. For example, don't do this in a command:

```
from outflow.core.pipeline import settings

settings.PLUGINS = []   # Don't do this!
```

The only place you should assign to settings is in a settings file.

3.8.6 Available settings

```
PLUGINS : List of strings, each of then will be imported by outflow.

TEMP_DIR : Outflow will use this directory to save temporary files, like MapWorkflows␣
↪serialized input and outputs, or workflow caches. Defaults to '/tmp/outflow_tmp.␣
↪IMPORTANT : if using a shared computing cluster, /tmp is probably not shared between␣
↪nodes. Please set TEMP_DIR in your settings.py file to a directory that is shared␣
↪between nodes.

ROOT_DIRECTORY : Directory of the pipeline. Set by manage.py to be its current directory.

MAIN_DATABASE : Name of the default database, the one outflow will use for its internal␣
↪tables, and for any models without special binds.

BACKENDS : Mapping of each backend and their module path.

ROOT_COMMAND_CLASS : Class of the root command, override for special behaviour. Defaults␣
↪to RootCommand that does nothing particular.
```

3.8.7 Creating your own settings

There's nothing stopping you from creating your own settings, for your own Outflow pipeline, but follow these guidelines:

- Setting names must be all uppercase.
- Don't reinvent an already-existing setting.

For settings that are sequences, Outflow itself uses lists, but this is only a convention.

3.9 Outflow configuration

The configuration file contains information that is likely to change from one pipeline instance to another. It is

3.9.1 The basics

Outflow can handle json, toml and yaml file formats for the configuration file.

Here are a couple of example fields for the configuration file:

```
databases:
  default:
    dialect: postgresql
    admin: pipeadmin:adminpwd
    user: pipeuser:userpwd
    address: postgres
    database: pipeline_db

custom_field: my_value
```

3.9.2 Designating the configuration file

When you use Outflow, the default configuration file (`config.json`, `config.yml` or `config.toml`) is the one located at the root of the pipeline folder.

But, you can specify which file you're using. Do this by using an environment variable, `OUTFLOW_CONFIG_PATH`, or using the CLI argument `--config`.

In both cases, the value should be a filepath `export OUTFLOW_SETTINGS_MODULE=/path/to/my/config.yaml` or `--config /path/to/my/config.yaml`.

3.9.3 Default configuration

The default Outflow configuration only includes a default logging field with a basic configuration. This configuration lives in the file `outflow/core/logging/config.json`.

Here's the algorithm Outflow uses in compiling the configuration:

- Create an empty configuration and fill the logging field with the basic logging configuration.
- Load the configuration file from the specified filepath, overriding the default fields as necessary.

Note that if the user specify a logging field in its own configuration file, the whole default logging configuration will be overwritten.

3.9.4 Using the configuration in the code

In your Outflow pipeline, use config by importing the object `outflow.core.pipeline.config`. Example:

```
from outflow.core.pipeline import config

print(config.logging)
```

Note that `outflow.core.pipeline.config` isn't a module – it's an object. So importing individual config fields is not possible:

```
from outflow.core.pipeline.config import logging  # This won't work.
```

3.9.5 Altering the configuration at runtime

You shouldn't alter the configuration in your applications at runtime. For example, don't do this in a command:

```
from outflow.core.pipeline import config

config.logging = {}   # Don't do this!
```

[comment]: <> (## Available configuration fields) [comment]: <> (For a full list of available configuration fields, see the settings reference. (#TODO))

3.9.6 Creating your own configuration fields

There's nothing stopping you from creating your own configuration fields, for your own Outflow pipeline, but don't reinvent an already-existing field.

3.9.7 Include other yaml files from your config.yml

Outflow comes with pyyaml-include that allows you to include yaml files. You might have multiple configuration files with some identical sections, this allows you to write these sections only once.

3.10 Logging

Logger configuration is integrated into outflow to facilitate logging from the different plugins. By default a simple configuration and formatter is loaded, you can override them in your config.yml file.

3.10.1 Logging from tasks

The logger object in the logging module of outflow core is already set up with all the registered plugins:

```
#inside a tasks.py file
from outflow.core.logging import logger

@as_task
def MyTask():
    logger.info("Hello from MyTask")
```

result :

```
2021-09-30 14:17:43,459 - my_project.my_plugin - tasks.py:15 - INFO - Hello from MyTask
```

NOTE

Since the outflow logger looks at the calling module to route the logs, you tasks must be imported using the namespace convention and not with an absolute import:

```
#inside a commands.py file
#do this
from my_project.my_plugin import MyTask
#not this
from plugins.my_plugin.my_project.my_plugin.tasks import MyTask
```

3.10.2 Configuring your logger

The syntax of the logging section of the config.yml file is the same as a DictConfig from the logging module. In it, you can configure the level, handlers etc.. of the logs. (see python documentation about logging configuration for more details)

```
logging:
  loggers:
    matplotlib:
      level: WARNING
      handlers: ["console"]
    my_plugin:
      level: DEBUG
      handlers: ["console"]
    my_another_plugin:
      level: ERROR
      handlers: ["console"]
```

Log to a file

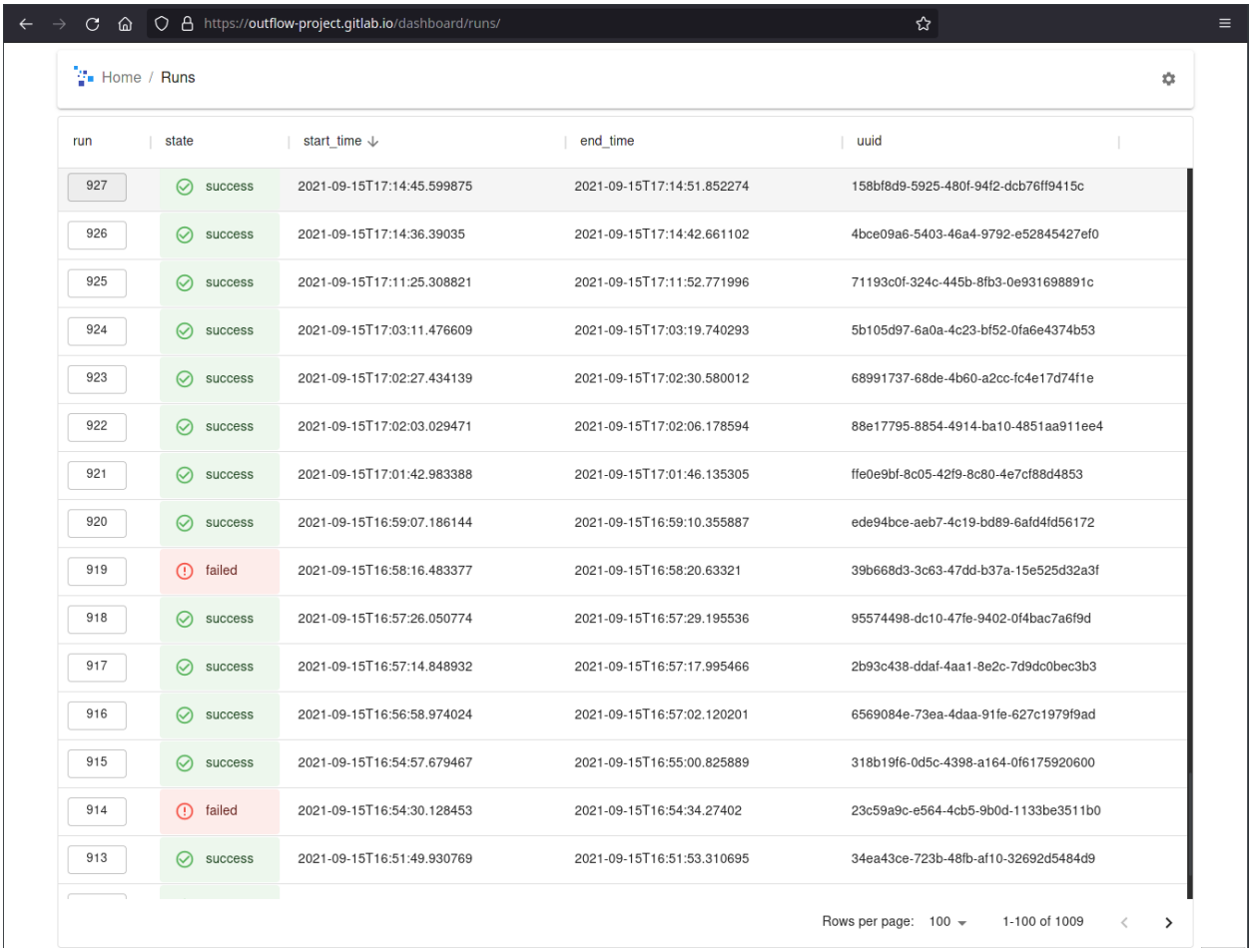
To log to a file, you can use the handler "file" in your configuration :

```
logging:
  loggers:
    "": # root logger, will log everything to a file
      level: DEBUG
      handlers: ["file"]
```

3.11 Dashboard

The outflow dashboard is a web interface where you can check the execution of your pipeline runs.

It is accessible at https://outflow-project.gitlab.io/dashboard/, but will probably be empty for now as it needs some setting up (see next session).



3.11.1 Set up Hasura

The dashboard uses hasura to query the run execution in database in real time. Hasura is compatible only with postgres databases, not sqlite.

You need to run an instance of hasura that can connect to you postgres database. You have two choices :

- run hasura in docker on a machine that can access the postgres server using `network_mode: host` inside the hasura docker-compose file.
- run both hasura and your postgres server using the same docker-compose.

Then:

- Go to the hasura console (usually http://localhost:8080/console)
- Click on the `data` tab.
- Click on Connect Database
- Fill your database url
- Click on `View Database`
- Click on `Track All` to track all tables
- Click on the other `Track All` button to track relationships

Now you can head over to the dashboard and you should see your pipeline executions.

By default, the dashboard queries hasura in at the default address and port of hasura on the localhost. If you are running hasura with the default address and port you have nothing to do, else click on the gear icon in the dashboard to set the hasura url.

3.11.2 View the task workflow

You can see the workflow of task executed by a run by clicking on the run in the left column.

3.12 Design Patterns

This section is a collection of useful patterns that can be implemented using the features of the framework.

Outflow is still young so if you find interesting way to use it, do not hesitate to share it with the community by starting a merge request with your addition to this section, or contact us on the discord server.

3.12.1 Use config or cli arguments inside the workflow definition

When outflow reads the code in the setup_task of your commands, it has already parsed both the configuration file and the command line arguments. That means you can access them and use the values to make your workflow configurable or conditional.

Example 1: use the cli arguments to make a task execute or not

Let's say you have a task called Debug that sets up a debug environment for your pipeline.

```
from outflow.core.pipeline import context
from outflow.core.commands import Command, RootCommand
from my_project.my_plugin.tasks import Debug, First, Second

@RootCommand.subcommand()
class MyCommand(Command):
    def add_arguments(self):
        self.add_argument("--debug", action="store_true", help="Set up debug environment
↪")

    def setup_tasks(self):
        first_task = First()
        second_task = Second()

        if context.args.debug:
            debug_task = Debug() # Remember that any instantiated task is executed
            debug_task | first_task

        first_task | second_task
```

Example 2: use the config.yml file to configure the resources of a MapWorkflow

You can define the sbatch directives inside the configuration and pass them to the MapWorkflow :

Inside the config.yml file:

```
my_map_resources:
  cpus_per_task: 5
  mem: 10GB
  partition: short
```

In your workflow definition.

```
from outflow.core.pipeline import config

@SlurmMapWorkflow.as_workflow(**config.my_map_resources)
def my_map():
    ...
```

This works because Outflow is responsible for importing your plugin code, and this will be done after loading the configuration file.

3.12.2 Define workflows and commands outside plugins

The code structure presented in the tutorial might not fit your pipeline design. If you prefer, it is possible to define workflows and commands inside the pipeline directory itself.

If your workflows use a combination of tasks from different plugins, it might be more fitting to define them in the pipeline directory rather than inside one of the plugins. This will also help with avoiding dependencies between plugins.

Implementation

Simply create a `my_commands.py` file inside your pipeline directory containing the definition of your command.

Then, you need to import this command module from the manage.py file, so Outflow is able to find and register it (this step is done automatically for plugins).

```
# inside the manage.py file

if __name__ == "__main__":
    pipeline_root_directory = Pipeline.get_parent_directory_posix_path(__file__)
    # add plugins to the python path
    # note: for cython like plugins, the compilation step is required and
    # plugin installation via pip is strongly encouraged
    plugins_dir = pathlib.Path(__file__).parent_workflow / "plugins"
    for plugin_path in plugins_dir.glob("*"):
        sys.path.append(plugin_path.resolve().as_posix())

    ###
    # <--- add either one of these two lines, if you use flake8 it will complain about␣
↪the first one with "imported but unused" so you might want to use the second one
    import my_commands

    importlib.import_module("my_commands")
```

(continues on next page)

(continued from previous page)

```
    ###

    with Pipeline(root_directory=pipeline_root_directory) as pipeline:
        result = pipeline.run()
```

That's all! You can now call this new command as usual with `python manage.py some_command`.

3.12.3 Nesting MapWorkflow and IterativeWorkflow

Case : you have a MapWorkflow inside an IterativeWorkflow, one of the tasks inside the MapWorkflow needs the output of the iterative workflow. In the example below, the Process task needs the output of the IterativeWorkflow, ie the output of ReduceProcessed. This value is an int for the example. In this case, you need to manually add an input to the MapWorkflow, because there is one level of workflow between the iterative workflow and the tasks that need the reinjected value. During execution, all blocks defined one level inside the IterativeWorkflow can have access to the reinjected value, but the MapWorkflow will only retrieve the value if set manually.

```
@as_task
def Process(reduced_data: Optional[int]):
    # do something with reduced_data
    pass

@as_task
def ReduceProcessed() -> {}:
    # return some int (for the example) that represent the combination of all processed␣
↪files
    pass

@MapWorkflow.as_workflow
def inner_map_workflow():
    Read() | Process()

def break_func(reduced_data: int):
    if reduced_data > threshold:
        return True
    else:
        return False

@IterativeWorkflow.as_workflow(max_iterations=5, break_func=break_func)
def my_iterative_workflow():
    inner_map = inner_map_workflow()
    inner_map | ReduceProcessed()

    # mandatory : manually add input to the map_workflow
    inner_map.add_input("reduced_data", type=Optional[int])

class MyCommand(Command):
    def setup_tasks(self):
        ListFiles() | my_iterative_workflow()
```

3.13 Example workflows

3.13.1

```
            > C
            |       |
A > B         |-> E
            |       |
            > D
```

```
from outflow import Command
from mypipeline.tasks import A, B, C, D, E

class MyCommand(Command):
    def setup_tasks():
        a = A()
        b = B()
        e = E()

        a | b | C() | e
        b | D() | e
```

Note that C and D will not be executed in parallel. This needs some work on the backend and will be possible in a future version of outflow.

3.14 Outflow standard task and workflow library

3.14.1 Tasks

Built-in tasks

IPythonTask

This task spawn an ipython shell, so that you have interactive access to everything that a task has access to, notably the pipeline context and the objects at this stage of the workflow.

The pipeline context is available with `from outflow.core.pipeline import context` as usual.

The objects returned by the previous task are in the variable `kwargs`.

Example use

Let's say you have a workflow like this one :

```
from outflow.core.tasks import Task
from outflow.core.logging import logger
from outflow.core.commands import Command, RootCommand
```

(continues on next page)

(continued from previous page)

```
@as_task
def TaskA() -> {"some_output": str}:
    value = "some_value"
    logger.info(f"Value in TaskA : {value}")
    return {"some_output": value}

@as_task
def TaskB(some_output):
    logger.info(f"Value in TaskB : {some_output}")


@RootCommand.subcommand()
class SomeCommand(Command):
    def setup_task(self):
        TaskA() | TaskB()
```

And you want to visualize or edit the values of your objects between the tasks A and B. Edit the workflow and add an InteractiveTask between the two:

```
...
from outflow.library.tasks import InteractiveTask

class SomeCommand(Command):
    def setup_task(self):
        A = TaskA()
        B = TaskB()
        intepreter = InteractiveTask()
        A | intepreter | B
```

Now if you run this command, you can view and edit the value inside “some_output” :

```
$ python3 manage.py some_command
outflow_tests.plugin_a.tasks - tasks.py:10 - INFO - Value in TaskA : some_value
|> You are inside an outflow interactive task. You can access the outputs of the␣
↪previous task through the dictionary 'kwargs'.
If you edit the values in 'kwargs', the changes will be passed to the next task.
To quit, do NOT use exit(), use EOF instead (usually CTRL+D)

kwargs
{'some_output': 'some_value'}
|> kwargs["some_output"] = "some_other_value"
|> ^D
now exiting InteractiveConsole...
outflow_tests.plugin_a.tasks - tasks.py:15 - INFO - Value in TaskB : some_other_value
```

IfThenElse

The IfThenElse construct split a workflow into two branches, and execute either of them depending on a user-defined condition. The input targets of the first task of both branches should be the output of the task before the branching.

`IfThenElse` is a function that takes a Callable (the condition) and return a tuple of three tasks:

- the first is the "if" in which you pipe the inputs
- the second is the "then", out of which you pipe the tasks to execute if the condition succeeds
- the third is the "else"

Either the "then" or the "else" its children will be executed, the other and its children will be skipped (ie return a Skipped object).

The condition must have this signature :

```
def condition(**kwargs):
    # kwargs contains all the inputs of the task piped to the "if"
    # return either True or False
```

Your condition can use the pipeline arguments, the database, or the values returned by the previous task.

It is possible to merge the two branches of an IfElse construct using the MergeTask.

Example usage

Branches containing tasks with targets

The first task of each branch should have the same input targets:

```
from outflow.library.tasks import IfThenElse
from outflow.core.tasks import Task
from outflow.core.logging import logger
from outflow.core.commands import Command, RootCommand
import random


@as_task
def A() -> {"choice": bool}:
    choice = random.choice([True, False])
    return {"choice": choice}

@as_task
def B(choice: bool) -> {"output": str}:
    logger.info("Executing B")
    return {"output": "b"}

@as_task
def C(choice: bool) -> {"output": str}:
    logger.info("Executing C")
    return {"output": "c"}

@RootCommand.subcommand()
```

(continues on next page)

(continued from previous page)

```
class SomeCommand(Command):
    @staticmethod
    def condition(**kwargs):
        return kwargs["choice"]

    def setup_tasks(self):
        a = A()
        b = B()
        c = C()

        if_t, then_t, else_t = IfThenElse(self.condition)

        a | if_t

        then_t | b
        else_t | c
```

(MergeTask)=

MergeTask

The MergeTask can be used to merge multiple branches of a workflow. These branches can be either the result of an IfThenElse construct or manual branching using the piping syntax.

The MergeTask returns the only non-Skipped result of the different branches.

Merge IfThenElse branches

```

@as_task
def A() -> {"out1": str}:
    return {"out1": "a"}


@as_task
def B(out1: str) -> {"out1": str, "out2": str}:
    return {"out1": out1, "out2": "b"}


@as_task
def C(out1: str) -> {"out1": str, "out2": str}:
    return {"out1": out1, "out2": "c"}

@as_task
def LastTask(out1: str, out2: str):
    logger.info(f"{out1=} , {out2=}")
    # will print either
    # out1 = a , out2 = b
    # or
    # out1 = a , out2 = c
```

(continues on next page)

(continued from previous page)

```
@RootCommand.subcommand()
class SomeConditionalCommand(Command):
    @staticmethod
    def condition(**kwargs):
        # some condition

    def setup_tasks(self):
        a = A()
        b = B()
        c = C()
        if_t, then_t, else_t = IfElse(self.condition)
        merge_task = MergeTask()
        last_task = LastTask()

        a | if_t

        then_workflow | b | merge_task
        else_workflow | c | merge_task

        merge_task | last_task
```

Merge manual branches

Merging manual branching works the same, but you have to manually return Skipped() from either one branch or the other.

`from outflow.core.types import Skipped`

3.14.2 MapWorkflow

MapWorkflows implements the pattern map-reduce in Outflow. Its purpose is to split an input sequence, and execute itself as many times as the length of the sequence. Each workflow execution will get one of these items.

There is 3 implementations of the MapWorkflow, one for each backend. It is recommended to use the generic `outflow.core.library.tasks.MapWorkflow` and let the backend choose the right one. Keyword arguments concerning slurm are ignored by other MapWorkflow implementations, so you can for example pass kwargs specific to a SlurmMapWorkflow to the MapWorkflow, and they will be ignored if ran with the default backend.

Usage

```
@as_task
def ListFiles() -> {"file_path_list": List[str]}:
    paths = os.listdir(data_directory)
    return {"file_path_list": paths}

@as_task
def Read("file_path": str) -> {"data": Any}:
    # read file
```

(continues on next page)

(continued from previous page)

```
    return {"data": data}

@MapWorkflow.as_workflow
def process():
    Read() | Transform() | Save()

class Process(Command):
    def setup_tasks(self):
        ListFiles() | process()
```

SlurmMapWorkflow

You can configure how your SlurmMapWorkflow will be distributed by passing arguments to the SlurmMapWorkflow.

```
@SlurmMapWorkflow.as_workflow(partition="short",
                              mem="10GO")
def process():
    Read() | Transform() | Save()
```

See simple-slurm github for the syntax, and sbatch documentation for available arguments.

There is only one (optional) argument specific to outflow : `simultaneous_tasks` which specifies the value after the % sign of the slurm array. This tells slurm how many jobs of the slurm array (ie how many mapped workflows) will be executed at the same time. This can be useful if there are limitations on the number of cpus per user on your slurm cluster.

The argument `cpu_per_tasks` is interesting if you have tasks that can take advantage of multiprocessing computations.

Use config.yml to specify SlurmMapWorkflow sbatch directives

You can easily store your SlurmMapWorkflows configurations in the config.yml file :

```
my_map_config:
  partition: batch
  cpus_per_task: 2
  simultaneous_tasks: 10
```

```
from outflow.core.pipeline import config

with SlurmMapWorkflow(**config["my_map_config"]) as map_task:
    MyComputation()
```

3.14.3 LoopWorkflow

The LoopTask allows you to repeat a workflow, either a given number of iterations, or indefinitely.

Usage

This will repeat the workflow `A | B` 3 times:

```
from outflow.library.tasks import LoopWorkflow

@LoopWorkflow.as_workflow(iterations=3)
def looped_workflow():
    TaskA() | TaskB()
```

`infinite=True` will repeat the workflow `A | B` until an exception is raised, or the processed killed, ctrl-c pressed, or exit_pipeline() called.

```
@LoopWorkflow.as_workflow(infinite=True)
def looped_workflow():
    TaskA() | TaskB()
```

Combining the LoopTask with a `Sleep` Task is useful for automatic processing:

```
@as_task
def Sleep(seconds=0):
    logger.info(f"Sleeping for {seconds} seconds")
    time.sleep(seconds)

@LoopWorkflow.as_workflow(infinite=True)
def infinite_processing():
    TaskA() | TaskB() | Sleep(seconds=60)
```

CHAPTER

FOUR

QUICK REFERENCE

4.1 Turn a function into a outflow task

Let's say you have this function :

```
def do_stuff():
    # some computation
```

To turn this function into a outflow task, you only need to add the `@as_task` decorator :

```
@as_task
def DoStuff():
    # some computation
```

The decorator `@as_task` turns a function into a class so it may be better to use the class naming convention here

4.2 Specifying the outputs of a task

Tasks usually return some data for the next task. You have two choices to define the outputs of your tasks :

4.2.1 Recommended: return annotation

```
@Target.output("returned_data")
@as_task
def DoStuff() -> {"returned_data": int}:
    ret = #some computation
    return {
        "returned_data": ret
    }
```

4.2.2 Alternative way: Target.output decorator

```
@Target.output("returned_data", type=int)
@as_task
def DoStuff():
    ret = #some computation
    return {
        "returned_data": ret
    }
```

CHAPTER

FIVE

API REFERENCE