Planchet Documentation

Kristian Boda

May 30, 2020

CONTENTS

CHAPTER ONE

INTRODUCTION

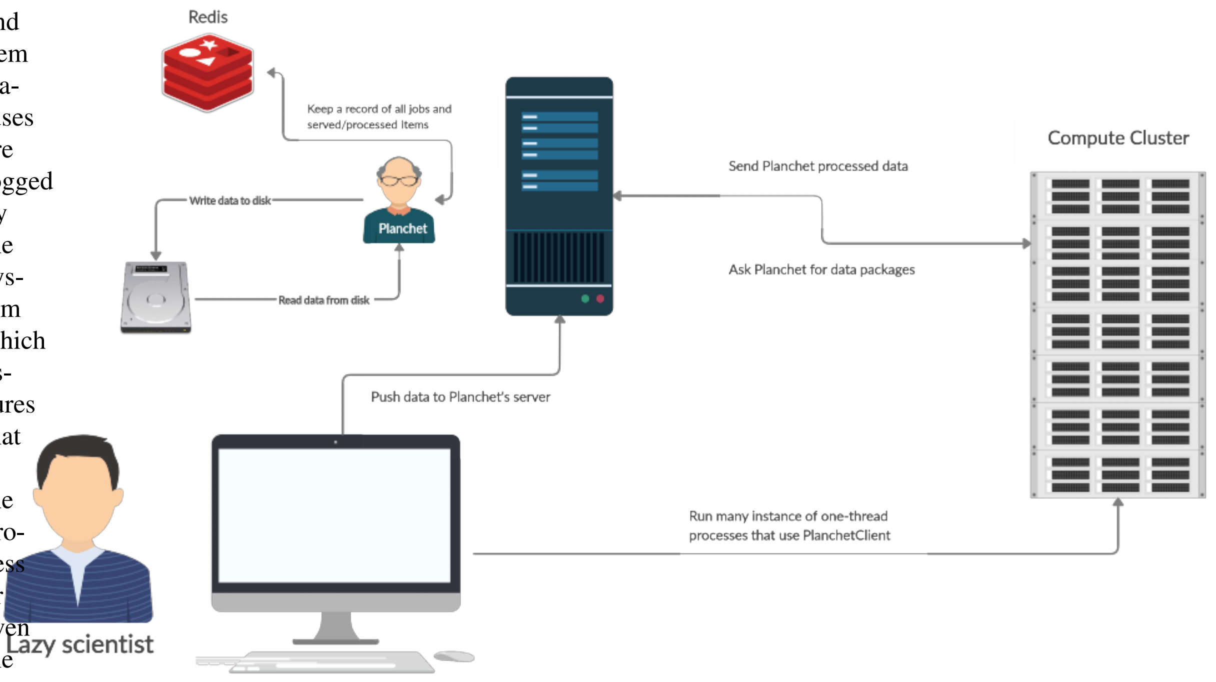
Planchet (pronounced /pl/) is a data package manager suited for processing large arrays of data items. It supports natively reading and writing into CSV and JSONL data files and serving their content over a FastAPI service to clients that process the data. It is a tool for scientists and hackers, not production.

1.1 How it works

Planchet attempts to solve the problem of controlled processing of large amounts of data in a simple and slightly naive way. It controls the reading and writing of the data and leaves the processing to the user. The usage takes roughly the following shape:

- You start by creating a job which specifies where to read, where to write, and what classes to use for that.
- Then, you ask for *n* data items, which your process works through locally.
- When your processing is done, you send the items back to the service to be written to disk.

All jobs and item statuses are logged by the system which assures that if the process or even the service


Redis
Keep a record of all jobs and served/processed Items
Write data to disk
Planchet
Read data from disk
Send Planchet processed data
Ask Planchet for data packages
Compute Cluster
Push data to Planchet's server
Run many instance of one-thread processes that use PlanchetClient
Lazy scientist

is
interrupted, only in-flight data processing will be lost.

1.2 Constraints

Single thread: Planchet is running in a single thread to avoid the mess of multiple processes writing in the same file. Until this is fixed (may be never) you should be careful with your batch sizes – keep them big enough to avoid overwhelming the service with requests but not too big so that you avoid request timeouts.

Independence: this tool was built in mind with independence of the data points. What that means is that if you are processing a CSV, you don't need line 1 to process line 2. Technically, even in that case you could implement something using Planchet, but the logic on the client side will not be pretty .

CHAPTER

TWO

QUICKSTART

This guide will take you through all the steps to set up a Planchet instance, and run a simple NER processing over sample text using a spaCy worker script.

2.1 On the server

On the server we need to install Planchet and download some news headlines data in an accessible directory. Then we copy over the data 1000 times to make it large.

```
git clone https://github.com/savkov/planchet.git
cd planchet
mkdir data
wget https://raw.githubusercontent.com/explosion/prodigy-recipes/master/example-
↪datasets/news_headlines.jsonl -O data/news_headlines.jsonl
python -c "news=open('data/news_headlines.jsonl').read();open('data/news_headlines.
↪jsonl', 'w').write(''.join([news for _ in range(200)]))"
export PLANCHET_REDIS_PWD=my-super-secret-password-%$^@
make install
make install-redis
make run
```

Note that the service will run at 0.0.0.0:5005 on your host machine. If you want to use a different host or port, use the make parameters:

```
make run HOST=my.host.com PORT=6000
```

Note: this guide will **not** work if you run a docker instance. If you do want to do that, you will need to alter the script as indicated in the comments below.

2.2 On the client

On the client side we need to install the Planchet client and spaCy.

```
pip install planchet spacy tqdm
python -m spacy download en_core_web_sm
export PLANCHET_REDIS_PWD=<your-redis-password>
```

Then we write the following script in a file called `spacy_ner.py` making sure you fill in the placeholders.

```
from planchet import PlanchetClient
import spacy
from tqdm import tqdm

nlp = spacy.load("en_core_web_sm")

PLANCHET_HOST = '0.0.0.0'  # <--- CHANGE IF NEEDED
PLANCHET_PORT = 5005

url = f'http://{PLANCHET_HOST}:{PLANCHET_PORT}'
client = PlanchetClient(url)

job_name = 'spacy-ner-job'
metadata = { # NOTE: this assumes planchet has access to this path
    'input_file_path': './data/news_headlines.jsonl',  # <--- change to /data/[...]␣
↪if using docker
    'output_file_path': './data/entities.jsonl'  # <--- change to /data/[...] if␣
↪using docker
}

# make sure you don't use the clean_start option here
client.start_job(job_name, metadata, 'JsonlReader', writer_name='JsonlWriter')

# make sure the number of items is large enough to avoid blocking the server
n_items = 100
headlines = client.get(job_name, n_items)

while headlines:
    ents = []
    print('Processing headlines batch...')
    for id_, item in tqdm(headlines):
        item['ents'] = [ent.text for ent in nlp(item['text']).ents]
        ents.append((id_, item))
    client.send(job_name, ents)
    headlines = client.get(job_name, n_items)
```

Finally, we want to do some parallel processing with 8 processes. We can start each process manually or we can use the *parallel* tool to start them all.

```
seq -w 0 8 | parallel python spacy_ner.py {}
```

CHAPTER

THREE

INSTALLATION

Planchet works in two components: a *service* and a *client*. The service is the core that does all the work managing the data, while the client is a light wrapper around the requests library that makes accessing the service safer and more convenient.

3.1 Service Side

You can use this repo and start straight away like this:

```
git clone git@github.com:savkov/planchet.git
export PLANCHET_REDIS_PWD=my-super-secret-password-%$^@
make install
make run-redis
make run
```

If you want to run Planchet on a different port, you can use the *uvicorn* command but note that you **MUST** use only one worker.

```
uvicorn app:app --reload --host 0.0.0.0 --port 5005 --workers 1
```

You can also run docker-compose from the git repo:

```
git clone git@github.com:savkov/planchet.git
export PLANCHET_REDIS_PWD=my-super-secret-password-%$^@
docker-compose up
```

3.2 Client Side

Install the client from PyPi using:

```
pip install planchet
```

CHAPTER

FOUR

USAGE

Here we discuss all necessary elements of setting up a Planchet processing job. If you are looking for a complete example, look at Quickstart. If you are in a hurry, skip to the *Client section*. If you want a quick overview of what is possible, you can take a look at the *PlanchetClient* object or at the Swagger API page typically under http://localhost:5005/docs.

4.1 Jobs

The job is essentially a big of metadata that controls the data management on the server side. It is created though a separate request before starting to do processing. To set up a regular job you will need a name, and to specify the reading and writing methods through the parameters. There are, however, some other types of jobs you may want to run, like an error logging job or a reading job, or a repair job. Here's a list of all parameters relevant for setting up a job:

- *name*: the name of the job. If the job already exists, your initial request will fail.
- *reader_name*: the name of the reader class, e.g. `CsvReader`.
- *writer_name*: the name of the writer class, e.g. `CsvWriter`.
- *metadata*: the metadata passed on the reader and the writer classes (see more *below*).
- *clean_start*: restarts the job if it exists.
- *mode*: I/O mode for the job. Possible values: `read`, `write`, `read-write`.
- *cont*: if true, resets the iterator of the reader and allows to serve all incomplete items again.
- *token*: if used, the job will require an authentication token

Now that we know what the parameters of a job are, let's consider the scenarios we mentioned above.

Reading job: if you are interested in reading data from Planchet and not storing the results, you can set the `mode` parameter to `read` which will remove the requirement of specifying a writer.

Error logging job: if you just want to dump things through Planchet, like your errors, you can set the `mode` parameter to `write` and disable the reader.

Repair job: you will likely have a case where you interrupt a process or you crash the system but you want to continue your processing. Typically, Planchet will just give you the next items available and simply ignore the ones that it served but never received. Making this smart has a lot of complications so instead we handle it by running a repair job at the end using the `cont` parameter. It essentially resets the iterator and wipes all non-complete items from the log. Then it goes and reads them again while skipping all the complete items.

4.2 The data

You can currently process anything you want that can be read and written to a CSV or a JSONL file. The *constraints* to this are basically independence of the data points and considerate sizes of the items and the workers pool.

You could process data from/to other types of format if you build your own reader and writer as discussed in the *advanced section*.

In practical terms, the data is served in two formats based on whether it comes from a CSV or JSONL file:

```
items = client.get(job_name, n_items)
for id_, item in items:
    print(item)
    # prints a list if reading from a CSV file
    # prints a dictionary or a list if reading from a JSONL file
```

4.3 Readers & writers

Planchet currently supports CSV through the `CsvReader` and `CsvWriter` classes, and JSONL through the `JsonlReader` and `JsonlWriter` classes. You need to specify one of each pair as the name of your reader and writer in order to confuiigure a job. You will also need to provide a shared metadata file for the reader and the writer, which is essentially a configuration.

Currently, the following parameters are used:

- *input_file_path*: path to the data file for the job (both formats)
- *output_file_path*: path to the output file for the job (both formats)
- *chunk_size*: size of the chunk to be read by the CSV reading iterator; you probably don't need to worry about this one.
- *overwrite*: if true, existing files are overwritten; if false existing files are appended.

Example

```
{
  "input_file_path": "/path/to/file",
  "output_file_path": "/path/to/output",
  "chunk_size": 100,
  "overwrite": False
}
```

4.4 The endpoints

scramble: starts a job. Requires `name`, `reader_name`, `writer_name`, and `metadata` parameters. Can be further parametrised by `cont` to make a repair job and `mode` to control whether it will be a read-only, write-only or read and write job.

/serve: serves a batch of items from a job (`job_name`). The number of items depends on the `batch_size`.

/receive: receives a batch of items from a job (`job_name`) sent through the `items` parameter.

/mark_errors: marks items from job `job_name` spacified in `ids` as errors.

/delete: deletes `job_name` and all items associated with it. Does not clean the output file.

/clean: deletes all items assiciated with `job_name`.

/report: returns the status of `job_name` and numbers of completed items and currently in flight.

/health_check: checks if the service is healthy.

4.5 The client

It is possible to use Planchet by directly querying the API endpoints, but it is much more convenient to use the *PlanchetClient* object. This section will briefly show how to create and execute a regular job and how a to change it to a repair job using the client. For a full description of all methods, refer to the *source documentation*.

Example

```
from planchet import PlanchetClient
from tqdm import tqdm

PLANCHET_HOST = '0.0.0.0'  # <--- CHANGE IF NEEDED
PLANCHET_PORT = 5005   #  <-- CHANGE IF NEEDED

url = f'http://{PLANCHET_HOST}:{PLANCHET_PORT}'
client = PlanchetClient(url)

job_name = 'regular-job'
metadata = {
    'input_file_path': '/data/data.jsonl',
    'output_file_path': '/data/output.jsonl'
}

# make sure you don't use the clean_start option here
# to make this a REPAIR JOB, set --> cont=True
client.start_job(job_name, metadata, 'JsonlReader', writer_name='JsonlWriter')

# make sure the number of items is large enough to avoid blocking the server
n_items = 100
items = client.get(job_name, n_items)

while items:
    processed = []
    print('Processing item batch...')
    for id_, item in tqdm(items):
        item['hash'] = hash(item['text'])
        processed.append((id_, item))
    client.send(job_name, processed)
    items = client.get(job_name, n_items)
```

CHAPTER

FIVE

ADVANCED

Planchet generally makes things easier for you but sometimes it's really difficult to configure it correctly. This page will discuss some advanced topics that will help you understand how to do that correctly and hopefully anticipate some of the common failures.

5.1 The ledger / Redis

Planchet uses Redis as a ledger to log all jobs it manges as well as all the items from these jobs. The service uses a special instance of Redis that has persistence pre-configured (see here). Planchet can also be run with a regular Redis instance without that feature, of course.

The data stored in Redis takes the following shapes:

Job

```
key -> "JOB:<job_name>"
value -> "{'metadata': '...','reader_name': '...','writer_name': '...','mode': '...'}"
```

Item

```
key -> "<job_name>: <item_id>"
value -> 'SERVED' or 'RECEIVED' or 'ERROR'
```

Token

```
key -> "TOKEN:<job_name>"
value -> '<token>'
```

5.2 Requests and batching

The service is running in a single process and all reading and writing is done in a single thread. This presents some constraints on how the service can be used.

Batches: the batches need to be set carefully as a batch size that is too small would make the service block too easily if there is a large amount of workers. A batch size that is too large could result in the service taking too long to receive and write the bacth to disk, which would again block the service. Ideally, one should set the batch size to a reasonable size that would give the service enough processing time given the number of workers. This sounds like a very dark art, but unless you are using a worker pool of many tens or even hundreds of workers (or very large data items), you can just use the default value of 100 and not worry about it.

Rquests: we alleviate the possible wrong “guessing” of the batch size by simply retying the requests several times. This is built into the *client* and generally you don’t need to worry about it unless you feel you need to force a particular number of retries.

5.3 Data formats

Currently, Planchet supports reading and writing only in two data formats: JSONL and CSV. However, new readers and writers can be added in the *io module* if they have the following signatures:

```python
class MyReader:
    def __init__(self, meta_data: Dict):
        # The content of the metadata dictionary is not controlled.
        # You need to make sure that you pass the correct parameters to
        # your reader.

    def __call__(self, batch_size: int) -> List:
        # This method should return a list of items. The list should be
        # of length equal to the batch_size parameter.

class MyWriter:
    def __init__(self, meta_data: Dict):
        # The content of the metadata dictionary is not controlled.
        # You need to make sure that you pass the correct parameters to
        # your writer.

    def __call__(self, data: List):
        # This method takes a list of items and writes them to disk.
```

As you can see, the reading/writing is not really constrained in any way. In fact, you can easily implement your own classes that read and write from/to a database, for example (probably won’t work in docker unless you add the appropriate dependencies though).

5.4 Data directories

Planchet can access anything the user it run under can. This means that if you run it on the bare metal and point it to a file, it will find it. If you are using docker, however, you will need to mount a directory into the container so the path to that file will change. By default docker will mount `.data/` in the Planchet directory to `/data` in the container. Make sure that you get this right as the client will not complain in a very useful way.

5.5 Security

As stated a few times in this documentation: Planchet is not a production tool. The main reason for that is that it easily exposes the host to external jobs. The considerations are different in the different ways of running Planchet.

Docker: when you run the service using docker, you are essentially giving complete access to anything in the container to Planchet. Anything can be read and re-written using a job. The good thing is that you probably won’t have anything useful in that container .

Bare metal: when you simply run Planchet using the `make run` route, you will give Planchet (and its users) exactly the same access to the system as the user you are doing it with. You may want to set up a special user to protect your system (see this for inspiration), but you should also remember to include possible data sources and output destinations into its permissions.

5.6 Debugging

As a fairly young project, Planchet is not great at telling you want's wrong. You will probably run into some trouble at some point, so instead of feeling silly, go and read the logs. For docker you can use `docker logs -f <planchet-container>` to read the output of the system as requests are coming in. If you are running it on the bare metal, well it's probably where you're running it .

CHAPTER

SIX

SOURCE

6.1 API

6.2 planchet.core

6.3 planchet.client

6.4 planchet.io

CHAPTER

SEVEN

DOCUMENTATION FOR THE CODE

- modindex