Parade Documentation

Release 0.1.20.3

He Bai

Apr 23, 2019

First steps

Version: 0.1.20.3

`Parade` is a simple and out-of-box toolkit to handle data work such as ETL, data analysis, BI reports, etc, and enable fast and flexible integration mechanism with applications. It can be used for a wide range of purposes, from composing & scheduling data workflow to providing unified web-APIs of data query.

This documentation contains everything you need to know about Parade.

CHAPTER 1

Installation

1.1 Requirements

- Python 3.4+
- Works on Linux, Windows, Mac OSX, BSD

1.2 Install with pip

The quick way:

```
> pip install parade
```

CHAPTER 2

Tutorial

After installation, a command line tool *parade* is placed in $PATH. Have a glance at the usage output:

```
> parade -h

usage: parade [-h] {search,init} ...

The CLI of parade engine.

positional arguments:
  {search,init}
    search       search a contrib component
    init         init a workspace to work with

optional arguments:
  -h, --help     show this help message and exit
```

Until now, you can do nearly nothing but to initialize a workspace to place your task and other stuff. We leave the search command later . . .

2.1 Initialize Workspace

In this tutorials, we'll compose a series of ETL tasks, compose them into a DAG workflow and schedule the flow with a third-party scheduler (e.g, *Azkaban*). Type following command to Initialize the workspace named *example*:

```
> parade init example

New Parade workspace 'example', using template directory 'site-packages/parade/
↪template/workspace', created in:
    /$CMD/example

You can start your first task with:
```

(continues on next page)

(continued from previous page)

```
    cd example
    parade gentask your_task -t etl
```

Enter the workspace directory, and re-check usage again:

```
> parade -h

usage: parade [-h] {feature,task,flow,init,server,convert,notebook} ...

The CLI of parade engine.

positional arguments:
  {feature,task,flow,init,server,convert,notebook}
    feature             feature-related sub commands
    task                task-related sub commands
    flow                flow-related sub commands
    init                init a workspace to work with
    server              start a parade api server
    convert             convert a source table to a target table
    notebook            start a jupyter notebook server

optional arguments:
  -h, --help            show this help message and exit
```

You can find much more sub-commands available now. We come to the details of these sub-commands later. At this moment we have a look at the directory structure.

```
> tree

.
├── __init__.py
├── __pycache__
├── example
│   ├── __init__.py
│   ├── ...
│   ├── contrib
│   │   ├── ...
│   │   ├── connection
│   │   │   ├── __init__.py
│   │   │   └── ...
│   │   ├── flowrunner
│   │   │   ├── __init__.py
│   │   │   └── ...
│   │   ├── flowstore
│   │   │   ├── __init__.py
│   │   │   └── ...
│   │   └── notify
│   │       ├── __init__.py
│   │       └── ...
│   └── task
│       ├── __init__.py
│       └── ...
├── example-default-1.0.yml
├── jupyter
│   ├── ...
│   └── ...
```

(continues on next page)

(continued from previous page)

```
└── parade.bootstrap.yml

29 directories, 19 files
```

At top there are a package named *example* (as we specified) and two yaml files. The package has two sub-packages:

- *contrib* contains user defined components, such as connection drivers, task dagstores, etc.
- *task* holds all the data tasks to execute or schedule.

The yaml file *parade.bootstrap.yml* is just a pointer to the configuration repo for this workspace. Its content is as follows:

```yaml
workspace:
  name: example
config:
  name: example
  driver: yaml
  profile: default
  version: 1.0
  uri: "{name}-{profile}-{version}.yml"
```

The first section contains some basic information about the workspace. In the second section, we use a configuration repo based on **default** YAML driver, which is also a yaml file with formatted name *{name}-{profile}-{version}.yml* (You can implement your own configuration repo and specify it as *config.driver* in *parade.bootstrap.yml*). Providing the configuration name, *example*, profile, *default*, and version, *1.0*, the file configuration repo file is *example-default-1.0.yml*.

```yaml
connection:
  # name of the connection
  rdb-conn:
    driver: rdb
    protocol: postgresql
    host: 127.0.0.1
    port: 5432
    user: nameit
    password: changeme
    db: yourdb
    uri: postgresql://nameit:changeme@127.0.0.1:5432/yourdb
  elastic-conn:
    driver: elastic
    protocol: http
    host: 127.0.0.1
    port: 9200
    user: elastic
    password: changeme
    db: example
    uri: http://elastic:changeme@127.0.0.1:9200/
checkpoint:
  connection: rdb-conn
flowstore:
  driver: 'azkaban'
  host: "http://127.0.0.1:8081"
  username: azkaban
  password: azkaban
  project: TestProject
  notifymail: "yourmail@yourdomain.com"
  cmd: "parade exec {task}"
```

The file defines some third-party data connections and DAG-workflow stores for our tasks. We have two connections here: one names *rdb-conn*, connecting to the postgresql database *yourdb* with driver *rdb*, the other names *elastic-conn*, is a document database based by a elasticsearch server.

In the flowstore section, we use the famous job scheduler of LinkedIn, Azkaban, to schedule our data workflow. You may already find that Parade can be easily integrated with other third-party components with different **drivers**. This is benefited from its easy & unified plugin based architecture, which we'll present later.

The layout of example workspace so far are:

- Core package holds our data tasks and some contributed components
- The top level contains configuration files

Parade expects you keeps your workspace nice and tidy. There's a place for everything, and everything is in its place.

2.2 Compose Tasks

In this tutorial, we will load the IMDB 5000 Movie Dataset published on Kaggle into our database, and process some further analysis on this dataset, then re-store the analysis result into another table. After that we try to build a DAG workflow with these tasks and schedule them as a whole.

2.2.1 Compose a ETL task

For sake of convenience, we've placed the dataset as a unzipped CSV file in our github repo, check it here. In the first task, we will load this dataset as a table into our postgresql/mysql database.

Use gentask subcommand of Parade to generate the skeleton for this ETL task.

```
> parade task create movie_data -t etl
```

Open the skeleton python file *example/task/movie_data.py*, you'll find lots of attributes as *@property* functions you can override to customize the task. We only reserve the required attributes to make the source tidy:

```python
# -*- coding:utf-8 -*-
from parade.core.task import ETLTask


class MovieData(ETLTask):

    @property
    def target_conn(self):
        """
        the target connection to write the result
        :return:
        """
        raise NotImplementedError("The target is required")

    def execute_internal(self, context, **kwargs):
        """
        the internal execution process to be implemented
        :param context:
        :param kwargs:
        :return:
        """
        raise NotImplementedError
```

The first *@property* function *target_conn* is used to specify the connection where we store our ETL data. We can simply set it to the connection key in our configuration, like *rdb-conn*.

The other function *execute_internal* is the core logic of this task. We simply load the raw csv dataset, do some projection & filter operation on it, and then store the result. Parade can handle Pandas Dataframe (and dict) by default. So we can use pandas to process the data and return the result dataframe directly. The edited task is:

```
# -*- coding:utf-8 -*-
from parade.core.task import ETLTask


class MovieData(ETLTask):

    @property
    def target_conn(self):
        """
        the target connection to write the result
        :return:
        """
        return 'rdb-conn'

    def execute_internal(self, context, **kwargs):
        """
        the internal execution process to be implemented
        :param context:
        :param kwargs:
        :return:
        """
        df = pd.read_csv('https://raw.githubusercontent.com/bailaohe/parade/master/
↪assets/movie_metadata.csv')

        # Process projection on the dataset to get our interested attributes
        df = df[['movie_title', 'genres', 'title_year', 'content_rating', 'budget',
↪'num_voted_users', 'imdb_score']]

        # Filter out records with *NAN* title_year and budget
        df = df[pd.notnull(df['title_year'])]
        df = df[df['budget'] > 0]

        # Extract the genres ROOT
        df['genres_root'] = df['genres'].apply(lambda g: g.split('|')[0])

        return df
```

2.2.2 Execute the ETL task

Use the following Parade command to execute the single task *movie_data*.

```
> parade task exec movie_data

2017-05-24 16:37:14,913 program_name DEBUG [11758] [exec.py:20]: prepare to execute␣
↪tasks ['movie_data']
2017-05-24 16:37:14,914 program_name INFO [11758] [exec.py:27]: single task movie_
↪data provided, ignore its dependencies
2017-05-24 16:37:35,634 program_name WARNING [11758] [rdb.py:72]: Detect columns with␣
↪float types ['title_year', 'budget', 'imdb_score'], you better check if this is␣
↪caused by NAN-integer column issue of pandas!
```

(continues on next page)

(continued from previous page)

```
2017-05-24 16:37:35,636 program_name WARNING [11758] [rdb.py:80]: Detect columns with␣
↪object types ['movie_title', 'genres', 'content_rating', 'genres_root'], which is␣
↪automatically converted to *VARCHAR(256)*, you can override this by specifying type␣
↪hints!
2017-05-24 16:37:38,374 program_name INFO [11758] [rdb.py:105]: Write to movie_data:␣
↪rows #0-#10000
```

When the execution is done, you can check your newly loaded data in the databased labeled by *rdb-conn*.

```
> select count(1) from movie_data;
+----------+
| count(1) |
+----------+
|     4543 |
+----------+
1 row in set (0.01 sec)
```

Relexed and comfortable, right? However, when you check the generated table, you may find something unsatisfactory.

```
> show create table movie_data;

CREATE TABLE `movie_data` (
  `movie_title` varchar(256) DEFAULT NULL,
  `genres` varchar(256) DEFAULT NULL,
  `title_year` double DEFAULT NULL,
  `content_rating` varchar(256) DEFAULT NULL,
  `budget` double DEFAULT NULL,
  `num_voted_users` bigint(20) DEFAULT NULL,
  `imdb_score` double DEFAULT NULL,
  `genres_root` varchar(256) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8

1 row in set (0.00 sec)
```

The columns *title_year* and *budget* should be of type Integer in original dataset, but stored as Double in the target table. This is because both of them has **NAN** value in the raw dataset. Although we've filtered NAN-value records out, Pandas automatically convert Integer columns with NAN-value into float. Check the outputs of execution above, Parade has warned on this issue. Moreover, all String-typed columns have been converted into *VARCHAR(256)* indistinctively.

To address this issue, we further customize the datatype of ETL target with *@property* function *target_typehints*. Parade provides a compact set of basic datatypes. We simply return a column-name to Parade-datatype dict in the function.

We can further customize the indexes on the ETL target with *@property* function *target_indexes*. The *rdb* connection driver can recognize this attribute and build indexes after writing the result. The final task source is as follows:

```
# -*- coding:utf-8 -*-

from parade.core.task import SingleSourceETLTask
from parade.type import stdtypes
import pandas as pd


class MovieData(SingleSourceETLTask):

    @property
    def target_conn(self):
```

(continues on next page)

(continued from previous page)

```
        """
        the target connection to write the result
        :return:
        """
        return 'stat'

    @property
    def target_typehints(self):
        """
        a dict of column_name => datatype, to customize the data type before write␣
↪target
        :return:
        """
        return {
            'movie_title': stdtypes.StringType(128),
            'genres': stdtypes.StringType(128),
            'genres_root': stdtypes.StringType(32),
            'content_rating': stdtypes.StringType(16),
            'title_year': stdtypes.IntegerType(),
            'budget': stdtypes.IntegerType(20),
        }

    @property
    def target_indexes(self):
        """
        a string or a string-tuple or a string/string-tuple list to specify the␣
↪indexes on the target table
        :return:
        """
        return ['movie_title', ('title_year', 'genres')]

    def execute_internal(self, context, **kwargs):
        """
        the internal execution process to be implemented
        :param context:
        :param kwargs:
        :return:
        """
        df = pd.read_csv('https://raw.githubusercontent.com/bailaohe/parade/master/
↪assets/movie_metadata.csv')

        # Process projection on the dataset to get our interested attributes
        df = df[['movie_title', 'genres', 'title_year', 'content_rating', 'budget',
↪'num_voted_users', 'imdb_score']]

        # Filter out records with *NAN* title_year and budget
        df = df[pd.notnull(df['title_year'])]
        df = df[df['budget'] > 0]

        # Extract the genres ROOT
        df['genres_root'] = df['genres'].apply(lambda g: g.split('|')[0])

        return df
```

Re-execute the task and check your database again:

```
mysql> show create table movie_data;

CREATE TABLE `movie_data` (
  `movie_title` varchar(128) DEFAULT NULL,
  `genres` varchar(128) DEFAULT NULL,
  `title_year` int(11) DEFAULT NULL,
  `content_rating` varchar(16) DEFAULT NULL,
  `budget` bigint(20) DEFAULT NULL,
  `num_voted_users` bigint(20) DEFAULT NULL,
  `imdb_score` double DEFAULT NULL,
  `genres_root` varchar(32) DEFAULT NULL,
  KEY `idx_movie_title` (`movie_title`),
  KEY `idx_title_year_genres` (`title_year`,`genres`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

1 row in set (0.02 sec)
```

2.2.3 Compose another Analysis task

Let's say that we need compose another task based on the ETL result of the first one. The task is used to analysis some distribution stats, such as top-rated (*imdb_score* >= *7*) count, excellent rate (*#top-rated / #total*), and average budget among the movie genres.

Since the dataset is already placed in a single database, we can process above analysis with one SQL statement:

```
SELECT
  genres_root,
  COUNT(1) DIV 1                              total_count,
  SUM(IF(imdb_score >= 7, 1, 0)) DIV 1        excellence_count,
  SUM(IF(imdb_score >= 7, 1, 0)) / count(1)  excellence_rate,
  AVG(budget) DIV 1                           avg_budget
FROM movie_data
GROUP BY genres_root
ORDER BY excellence_count DESC;
```

Parade provides another sub-task-type of *etl*, i.e., *setl* standing for *single source ETL*, to facilitate the implementation of ETL tasks can be accomplished by single query on source connection.

Run following command to generate the skeleton of this task:

```
> parade task create genres_distrib -t setl
```

Then we edit the source *example/task/genres_distrib.py* to contains following stuff:

```
# -*- coding:utf-8 -*-
from parade.core.task import SqlETLTask
from parade.type import stdtypes


class GenresDistrib(SqlETLTask):

    @property
    def target_conn(self):
        """
        the target connection to write the result
        :return:
```

(continues on next page)

(continued from previous page)

```
        """
        return 'stat'

    @property
    def target_typehints(self):
        """
        a dict of column_name => datatype, to customize the data type before write␣
↪target
        :return:
        """
        return {
            'genres_root': stdtypes.StringType(32),
            'avg_budget': stdtypes.IntegerType(20),
            'total_count': stdtypes.IntegerType(),
            'excellence_count': stdtypes.IntegerType(),
        }

    @property
    def source_conn(self):
        """
        the source connection to write the result
        :return:
        """
        return 'stat'

    @property
    def source(self):
        """
        the single source (table/query) to process etl
        :return:
        """
        return """
        SELECT
          genres_root,
          COUNT(1) DIV 1                            total_count,
          SUM(IF(imdb_score >= 7, 1, 0)) DIV 1      excellence_count,
          SUM(IF(imdb_score >= 7, 1, 0)) / count(1) excellence_rate,
          AVG(budget) DIV 1                         avg_budget
        FROM movie_data
        GROUP BY genres_root
        ORDER BY excellence_count DESC;
        """

    @property
    def deps(self):
        """
        a string-array to specified the dependant tasks has to be completed before␣
↪this one
        :return:
        """
        return ['movie_data']
```

You can see we just return the sql string from the *@property* function *source*. Along with that, the source specifies the source & target connection and typehints.

Finally, we add another *@property* function *deps* to customize the dependance of this task with an string-array. This task only requires a single task, i.e., *movie_data*.

After executing this task, we can check the analysis result in database:

```
> select * from genres_distrib;
+-------------+-------------+-----------------+----------------+------------+
| genres_root | total_count | excellence_count | excellence_rate | avg_budget |
+-------------+-------------+-----------------+----------------+------------+
| Drama       |         840 |             405 |         0.4821 |   24908465 |
| Action      |        1098 |             263 |         0.2395 |   64647649 |
| Comedy      |        1169 |             261 |         0.2233 |   31601107 |
| Adventure   |         433 |             165 |         0.3811 |   64342879 |
| Biography   |         233 |             161 |          0.691 |   24308608 |
| Crime       |         298 |             145 |         0.4866 |   36998053 |
| Documentary |          67 |              50 |         0.7463 |    5831930 |
| Horror      |         215 |              28 |         0.1302 |   11349153 |
| Animation   |          51 |              25 |         0.4902 |   50958431 |
| Fantasy     |          47 |              13 |         0.2766 |   14440319 |
| Mystery     |          32 |              13 |         0.4063 |   26230156 |
| Western     |          11 |               7 |         0.6364 |    3203181 |
| Sci-Fi      |          13 |               4 |         0.3077 |   17182307 |
| Thriller    |          16 |               3 |         0.1875 |    2959812 |
| Family      |          11 |               3 |         0.2727 |    6010909 |
| Romance     |           6 |               1 |         0.1667 |   20558833 |
| Film-Noir   |           1 |               1 |              1 |    1696377 |
| Musical     |           2 |               1 |            0.5 |    3189500 |
+-------------+-------------+-----------------+----------------+------------+
18 rows in set (0.01 sec)
```

From the result we can find that the genres with the most top-rates is *Drama*, but its average budget is much less than *Action* and *Adventure*.

2.2.4 Compose the third Archive task

Sometimes we may want to archive our data into some targets other than relational database, such as Elasticsearch. Parade's plugin-based architecture makes incorporation with contributed components, such as third-party connections, very easy. Let's try to archive the output of task *movie_data* into a elasticsearch server.

As mentioned above, you can implement your own connection driver in a python class (overriding the base *Connection* class) and place it into the package path *contrib/connection* in your workspace. Parade then can load your connection before task execution. We have provided some in-hand connections in our github repo parade-contrib. You can download the elasticsearch connection driver into your workspace by simply typing:

```
> parade feature install connection elastic
```

Now you will find a *elastic.py* file in *example/contrib/connection*. This connection require some configuration about the target elasticsearch server in *example-default-1.0.yml* as follows, which we have provided before:

```
elastic-conn:
  driver: elastic
  protocol: http
  host: 127.0.0.1
  port: 9200
  user: elastic
  password: changeme
  db: example
  uri: http://elastic:changeme@127.0.0.1:9200/
```

The following things are easy. We generate another setl-type task, archive_data, with dependence on *movie_data*. The task takes *elastic-conn* as target_conn, and return following sql statement from *source*:

```sql
SELECT
  movie_title, genres,
  title_year, content_rating,
  budget, num_voted_users, imdb_score
FROM movie_data
```

After execution, we can find the data is already stored in our elasticsearch:

```
> curl -XGET 'localhost:9200/example/_search' | json_pp

{
   "timed_out" : false,
   "hits" : {
      "max_score" : 1,
      "total" : 4543,
      "hits" : [
         {
            "_score" : 1,
            "_id" : "AVw9cIPdG8Vt64Emu4r_",
            "_source" : {
               "budget" : 250000000,
               "genres" : "Adventure|Family|Fantasy|Mystery",
               "imdb_score" : 7.5,
               "movie_title" : "Harry Potter and the Half-Blood Prince ",
               "num_voted_users" : 321795,
               "title_year" : 2009,
               "content_rating" : "PG"
            },
            "_type" : "archive_data",
            "_index" : "example"
         },
         ...
      ]
   },
   "took" : 14,
   "_shards" : {
      "successful" : 5,
      "total" : 5,
      "failed" : 0
   }
}
```

2.3 Build the DAG-workflow

Till now, we only execute every composed task one by one. You may find that we have specify the dependences between them in the task source, which make these tasks constitute a DAG-workflow, but Parade seems not recognitive to these attributes.

In fact, Parade can handle DAG very well in executing task batch. We can provide multiple tasks as arguments to sub-command *task exec*, Parade will build a DAG-workflow based on their inter-dependences and execute them in correct order.

```
> parade exec movie_data genres_distrib archive_data

2017-05-25 11:19:54,213 program_name DEBUG [5196] [exec.py:20]: prepare to execute␣
↪tasks ['movie_data', 'genres_distrib', 'archive_data']
2017-05-25 11:19:54,216 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[genres_distrib] ...
2017-05-25 11:19:54,216 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[movie_data] ...
2017-05-25 11:19:54,216 program_name DEBUG [5196] [engine.py:147]: task movie_data␣
↪start executing ...
2017-05-25 11:19:55,220 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[archive_data] ...
2017-05-25 11:19:56,224 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[genres_distrib] ...
2017-05-25 11:20:01,565 program_name DEBUG [5196] [engine.py:149]: task movie_data␣
↪Executed successfully
2017-05-25 11:20:01,565 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[genres_distrib] ...
2017-05-25 11:20:01,566 program_name DEBUG [5196] [engine.py:144]: all dependant␣
↪task(s) of task genres_distrib is done
2017-05-25 11:20:01,566 program_name DEBUG [5196] [engine.py:147]: task genres_
↪distrib start executing ...
2017-05-25 11:20:02,249 program_name DEBUG [5196] [engine.py:127]: pick up task␣
↪[archive_data] ...
2017-05-25 11:20:02,249 program_name DEBUG [5196] [engine.py:144]: all dependant␣
↪task(s) of task archive_data is done
2017-05-25 11:20:02,249 program_name DEBUG [5196] [engine.py:147]: task archive_data␣
↪start executing ...
2017-05-25 11:20:02,370 program_name DEBUG [5196] [engine.py:149]: task archive_data␣
↪Executed successfully
2017-05-25 11:20:02,779 program_name DEBUG [5196] [engine.py:149]: task genres_
↪distrib Executed successfully
```

As you have seen, Parade take dependences into consideration by default when executing multiple tasks. If you do not provide any task arguments, Parade will search and execute all tasks in current workspace as a single DAG. If you just want to execute multiple tasks rather than DAG, use option *–nodep* when running *task exec*.

At present, Parade can only handle DAG execution in single process in single host, which make us think Parade should support scheduling DAG execution to some third-party scheduler. In this example, we indicate that how we can submit our tasks as a workflow into LinkedIn's Azkaban Scheduler. Install this dagstore driver at first:

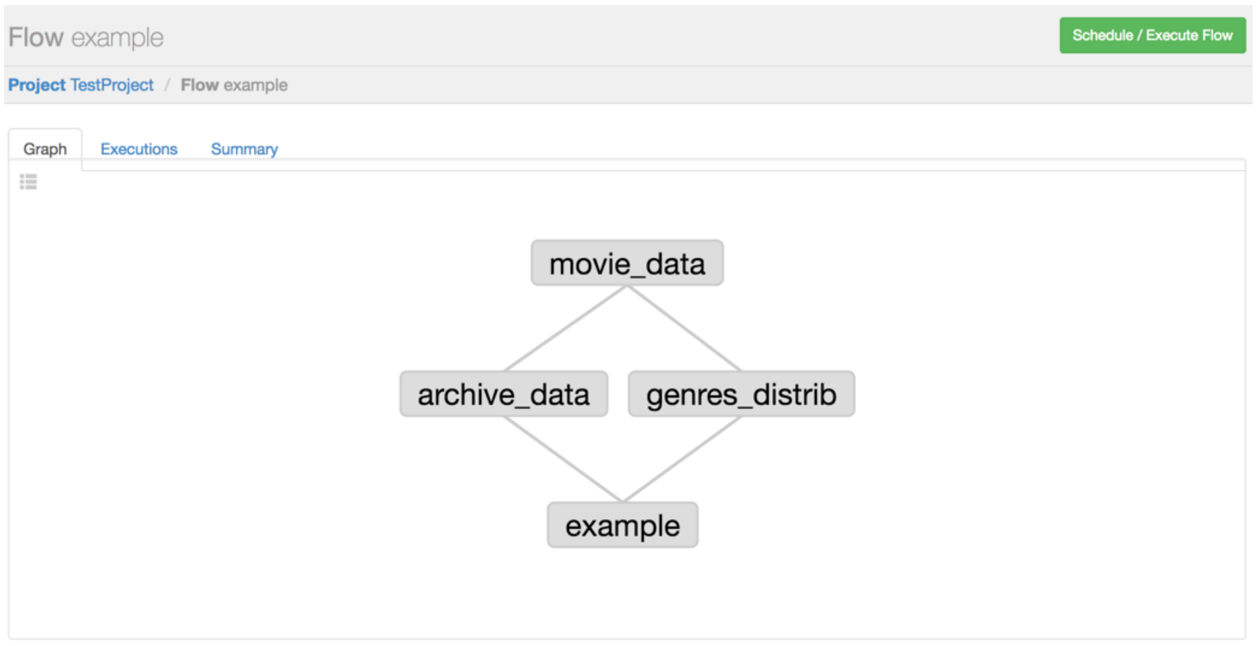
```
> parade feature install flowstore azkaban
```

Then add following azkaban configuration into example-default-1.0.yml:

```
flowstore:
  driver: 'azkaban'
  host: "http://127.0.0.1:8081"
  username: azkaban
  password: azkaban
  project: TestProject
  notifymail: "yourmail@yourdomain.com"
  cmd: "parade exec {task}"
```

Now we can use following command to submit the workflow to azkaban:

You'll find the DAG is already located in your azkaban server. And now you can schedule the execution with azkaban server.


Flow example
Schedule / Execute Flow
Project TestProject / Flow example
Graph
Executions
Summary
movie_data
archive_data
genres_distrib
example

CHAPTER 3

Indices and tables

- genindex
- modindex
- search