11-792 Project Report

Nicholas Gekakis, Boyue Li

May 7, 2018

1 Overview

In this project, we built a distributed parallel configuration space exploration pipeline framework, which can exhaustively try all possible parameter combinations automatically. Users can easily configure multiple modules, create their own modules and run on multiple processes or even multiple machines.

2 Requirements

2.1 Easy to configure and deploy

The framework should be easy to configure and deploy.

2.2 Save and resume

The framework should be able to save intermediate results so that it can be interrupted and resume running at a later time.

2.3 Automatical parameter exploring

The framework should be able to automatically execute using all possible parameter combinations that have been configured by the pipeline developers.

2.4 Easy to develop users' modules

The framwork should support a easy way for users to develop their own modules.

2.5 Load balancing

The framework should be able to handle load balancing since different modules require different excution time.

3 Design

3.1 Overview

As shown in Fig. 3.1, a pipeline is constructed from several independent modules. A module reads data from the data server, processes data according to all possible parameters, then save results for each configuration to the data server.

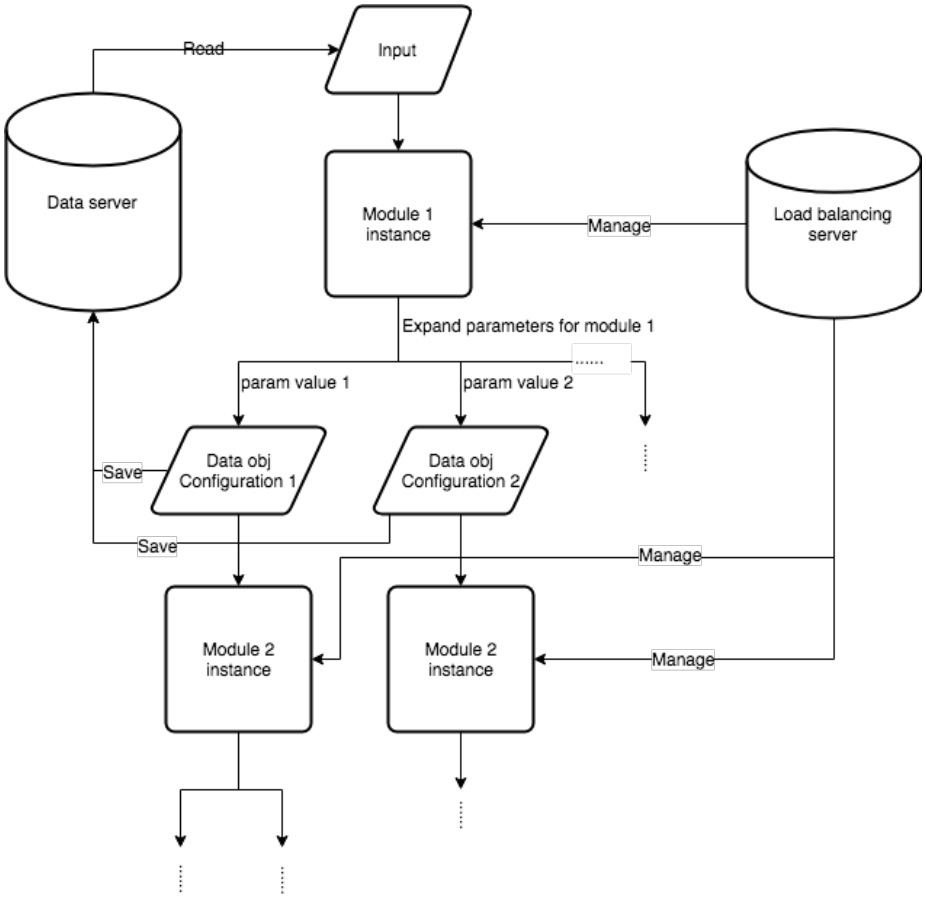

Figure 1: Control flowchart.

Every module runs on an independent process and communicates through RabbitMQ using the job class (defined in section 3.5) which contains parameters, current excuetion status and the path to input file. Fig. 3.1 describes the information flow. Load balancing server distributes jobs to different modules' instances. Once the job is finished, the instance send a confirmation to the load balancing server.

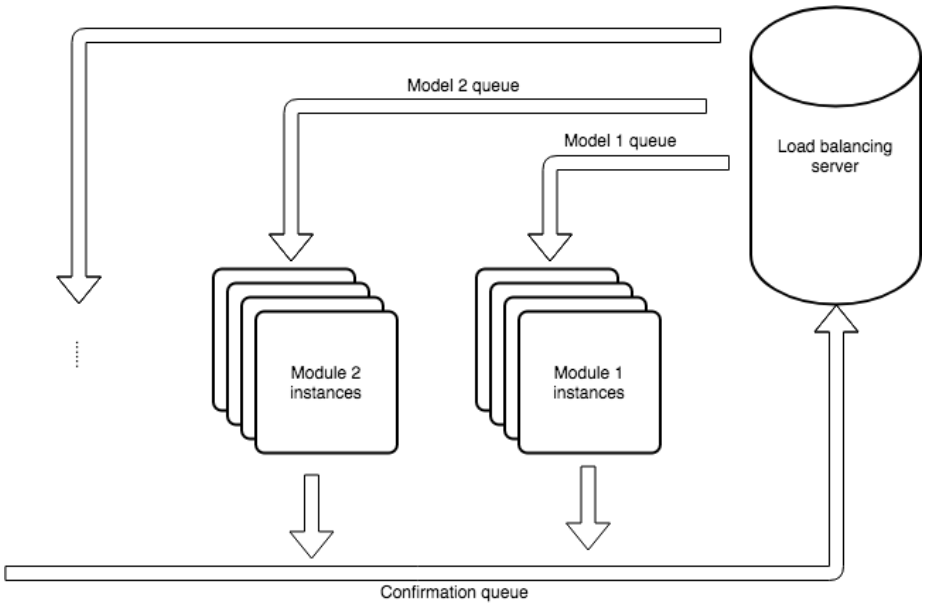

Figure 2: Information flowchart.

Users only need to specify the connections between modules and the parameters every module needs, the framework will automatically handle execution and configuration space exploration.

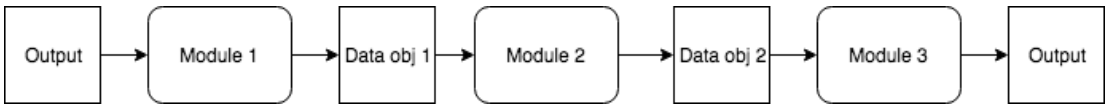

Figure 3: A sample pipeline

Figure 3.1 shows a sample pipeline which passes the input information object through some modules and produces the output information object.

We also provided a command line executable, to ease users' pain of coding.

3.2 Pipeline

The pipeline class manages modules and parameters. It reads configuration file, creates the pipeline and is the actual class for the load balancing server. The pipeline can't have branches or merges, and the order of the configuration file should be the same with the actual pipeline.

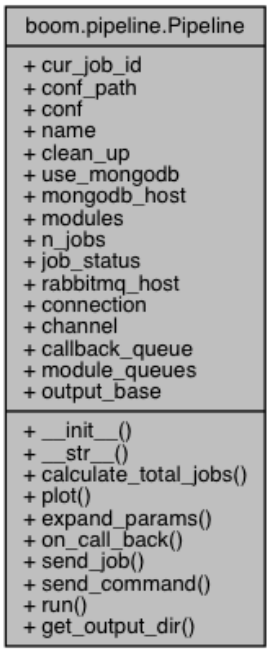

Figure 4: UML diagram for the pipeline class.

3.3 Module

A module is the basic computation unit which takes an input and produces an output. Every input and output is an job object defined in sec. 3.5.

A module needs to maintain the following fields:

- Name of the module.
- Number of instances.
- Configuration of the pipeline.

When running, the pipeline will send jobs to module instances.

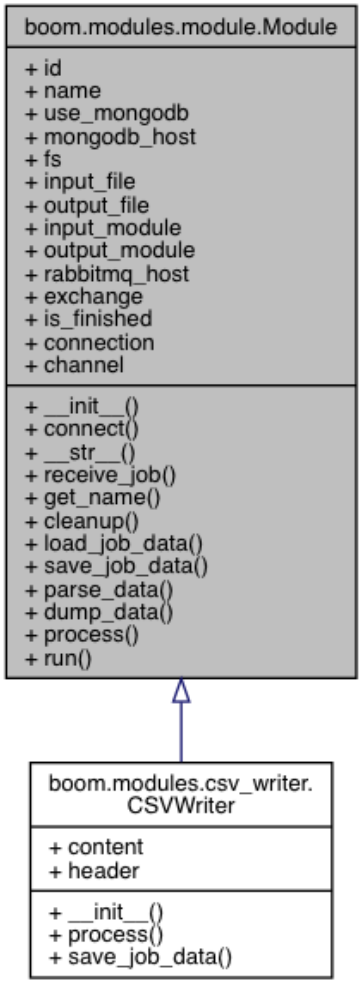

Figure 5: UML diagrams for the abstract module class and derived classes.

3.4 Parameter

The parameter class manages one parameter. It should handle all operations related the parameter, including updating the parameter value, set and reset the value. There are several types of params: integer, float and collection.

It also needs to save maintain the following fileds:

- Name of the parameter.
- Type of the parameter: int, float or collection.
- Interval of possible values or possible values of a collection.

- Step size of the parameter (for numerical params).

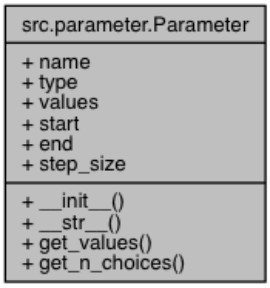

Figure 6: UML diagram for parameter class.

3.5 Job

The job class is the information object used between modules. It maintains the following fileds:

- Job id: the unique id of the job.
- Producer: the module that produced this information object.
- Consumer: the module that this information object to be passed to.
- Input uri: the uri to the data file.
- Output path: the path to store the resulting data file.
- Params: the params that the module should use to process the data object.
- Timestample: the timestamp when the data object was created.
- Processing time: the time taken to process the data object.

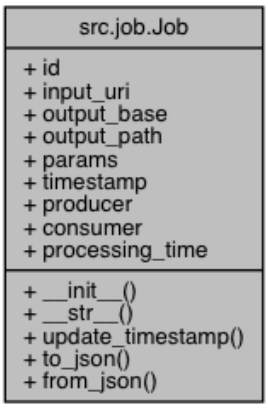

Figure 7: UML diagram for job class.

3.6 Configuration file

We use YAML files to configure the framework.

3.7 User defined modules

Users need to put their module classes in a file in the same folder with the configuration file. Then the framework will automatically load them.

3.8 Executable

We provide an executable that reads in the configuration file, and runs all possible configurations.

4 Example pipeline

This pipeline consists of a sample module, which does nothing but adds its own configuration and parameters to the data to show this pipeline is working.

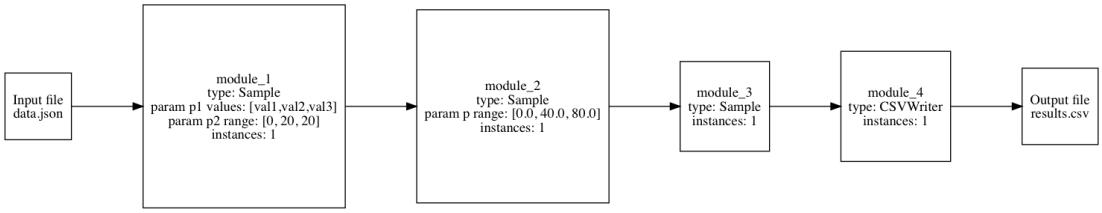

Figure 8: The structure for toy pipeline

5 Experiments

5.1 Dataset

The results were run on a subset (100 total questions) of the bioasq_train_formatted.json dataset.

5.2 Pipeline

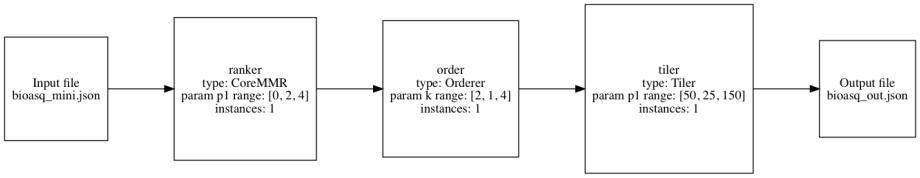

Figure 9: The structure for BioAsq pipeline

6 Implemented Modules

Table 1 listed all modules and their parameters we implemented. We have also improvemented the existing BioASQ code a lot. We made a multi-process CoreMMR module, that is more than 4x faster than the original module (also depending the configuration).

Module	Parameters
CoreMMR	α
Orderer	k
Tiler	word_limit
Rouge	

Table 1: Table of implemented modules.

6.1 Results

Acknowledgements

Thanks for Khyathi's help.