

Summary

The relationship between science and technology is growing stronger every day, this is evident in the fact that scientists cannot perform their day to day scientific activities without using new technologies that will facilitate their work and save them time and effort. Data processing workflows in the business field sector are widely used, but scientific workflow in the field of science research is relatively new. Scientific workflows are important in the field of science because they help manage the huge amount of data that results from different experiments. These data are hard to manage, analyse and aggregate manually to get the required results. Both scientific and business workflow management systems share the existence of the graphical user interface

The scientific workflow interpreter based on the Yahoo! Pipes project aims to use the Yahoo! Pipes editor to create different workflows using the Yahoo! Pipes modules, and then run these workflows locally. This is needed in cases where scientists try to run the workflow in Yahoo! Pipes, but because of the large amount of data the workflow will become slow or might abort, so, by using the JSON data of the workflow, we have emulated what Yahoo! Pipes are performing but locally. Yahoo! Pipes is a visual programming tool that helps its users create workflows in a visual environment, Since it has many modules for creating the workflow, such as Sources (fetch CSV, fetch feed, Google base, Yahoo search), user input (Data input, text input, URL input), Operators (count, create RSS, filter, loop, split, union), we chose certain modules and then emulated what these workflows were doing locally. The modules were chosen according to the reasoning given in chapter 3.

The workflow created in Yahoo! Pipes can be represented in three formats (XML, JSON, and PHP). The JSON format has its advantages, it is easy to be read and written by people, and is easily parsed by machine. JSON data can be parsed using different languages like Java, Perl, Java scripts, and Python. We used Python because it supports the web features and data structures we need. The project is a combination of software development and investigation. The software development part is the program that we have developed to emulate the different modules of Yahoo! Pipes and used to run the workflows locally. The investigatory part is where we found that Python was very supportive in developing the program and emulating the modules, it has various libraries and built-in functions.

We developed a program that will take the workflow id number (each Yahoo! pipe has a unique id number), and from this id the program will parse the JSON data of the pipe and find what modules it consists of; it will sort the modules using the topological sort algorithm then it will run each of these modules and at the end produce the output of the workflow created in Yahoo! Pipes on the screen. We investigated how Python was supportive during the development of the program and in emulating the modules, as we will see in chapter 3.

We accomplished the following different aspects of the project:

- 1- We used Python successfully to parse the JSON data of the workflow and used this data to emulate the modules of the workflow.
- 2- We emulated the most popular modules in Yahoo! pipes. We emulated 24 modules that could be used to produce a number of workflows.
- 3- We implemented the 'topological sort' algorithm mentioned in [32] [35] [38] to sort the modules of the workflows in order to implement them.
- 4- We ran several workflows that were created in Yahoo! Pipes using scientific data URLs; we could run them locally using the program that we developed.

Acknowledgment

It is an honour for me to thank everybody who made this thesis possible.

I would like to thank my project supervisor Professor Peter Flach for his supervision, advice and encouragement during every stage of this project.

I would like to thank Mr Simon Price for his guidance and ideas that were helpful in the project.

I would like to thank Dr Steve Gregory for his great support during the master program.

To my husband Abdullah and my daughters Fajer and Jouri, I am really thankful for their patience during my absence, and my oldest daughter Nejoood, who was my companion during my studies.

To my parents, brother and sister who supported me during my studies; although they were far, they always encouraged me.

Table of Contents

Summary.....	iii
Acknowledgment.....	iv
1. Introduction	1
1.1 Aims and Objectives	1
1.2 Dissertation Structure.....	1
2. Project Background.....	3
2.1 Scientific Workflow Management Systems	3
2.1.1 Scientific Workflows	3
2.1.2 Practical Scientific Workflow Management Systems (SWFMS)	4
2.1.2.1 Taverna	4
2.1.2.2 Kepler.....	7
2.1.2.3 Triana.....	8
2.2 Visual Programming and Yahoo! Pipes	10
2.2.1 Visual Programming Languages	10
2.2.2 Overview of Yahoo! Pipes	11
2.2.3 Creating a Yahoo! Pipe	13
2.2.4 Yahoo!Pipes structure	15
2.3 Summary.....	16
3. Scientific Workflow Interpreter Design and Implementation.....	17
3.1 The Scientific Workflow Interpreter	17
3.2 System General Overview.....	17
3.3 Topological Sorting Algorithm	19
3.4 Modules chosen	20
3.5 JSON Data Structure of Workflows	22
3.6 Scientific Workflow Interpreter Implementation	25
3.6.1 Python Language and Implementation	25
3.6.2 The Scientific Workflow Interpreter Main Modules	26
3.6.3 Interpreting Single Module	31
3.6.3.1 Text Input Module.....	31
3.6.3.2 Filter Module.....	32
3.6.4 Interpreting Control Flow Module	34
3.7 Summary.....	36
4. Scientific Workflows for Testing and Evaluation	37

4.1 Scientific workflow of earthquake data example1	37
4.2 Scientific workflow of earthquake data example2	39
4.3 Scientific workflow of genome data example1	41
4.4 Scientific workflow of genome data example2	43
4.5 Unit testing and workflow output.....	45
4.6 Evaluation	47
4.7 Summary.....	48
5. Conclusions and Future Work.....	49
5.1 Conclusions.....	49
5.2 Extending and improving the interpreter.....	49
References	51
Appendix: Source Code	54

1. Introduction

The relationship between science and technology is becoming stronger every day, this is evident in the fact that scientists cannot perform their day to day scientific activities without using new technologies that will facilitate their work and save them time and effort. Data processing workflows in the business field sector are widely used; there are many business workflow management systems such as: BPEL (Business Process Execution Language), Process Maker and Mentor lite [15], but scientific workflow in the field of science research is relatively new.

Scientific workflows are important in the field of science because they help manage the large amount of data that results from different experiments. These data are hard to manage, analyse and aggregate manually to get the required results. There also many scientific workflow management systems in the field of science such as: Taverna, Kepler and Triana (these will be explained later in chapter 2). Although the business and scientific workflow management systems differ in the fields they are implemented in, they all are the same in having a graphical interface which is user friendly and doesn't require any programming background to use. In this chapter we will describe the aims and objectives of the project and then describe the structure of dissertation.

1.1 Aims and Objectives

The aim of this project is to use Python as a programming language to write a program that parses the JSON data of the workflow created in Yahoo! Pipes, and use this data to emulate the modules each workflow consists of. The user will first create and edit the workflow in Yahoo! Pipes and then use the program we developed to run the workflow locally on his computer. Since Yahoo! Pipes has many modules for creating the workflows, we will choose certain modules and then emulate what these modules are performing. JSON data (this is the structure of the workflow created in Yahoo! Pipes) can be represented using different languages like Java, Perl, Java scripts, and Python. We used Python because it supports different data structures and provides the different libraries needed during the implementation process.

To meet the project's proposed aims, these are the objectives:

- 1- Parse the JSON data of any workflow created in Yahoo! Pipes: the JSON data is the main structure of any workflow and will be used mainly in the implementation process.
- 2- Sort the workflow modules in order to be able to run the workflow; the modules of the workflow must be sorted so as to know the order of implementation.
- 3- Emulate a significant number of modules that will allow the users to create different workflows in Yahoo! Pipes and run them locally using our program.
- 4- Create a number of workflows using scientific data in Yahoo! Pipes and run them using our program to test them; we used data from an earthquake website, and biological data of different species genes and proteins.

1.2 Dissertation Structure

The dissertation consists of five chapters. Chapter one is the introduction, aims and objectives of the project. Chapter two is the theoretical background of the project, it gives an overview of scientific workflows and how they are important. It gives an overview of other scientific workflow management systems used in the scientific field. It explains Yahoo! Pipes and how it is used to create workflows, it also gives a general overview of the JSON data of workflows.

Chapter three is about the design and implementation of the project. It gives a general overview of the program and explains how we choose the modules of Yahoo! Pipes that we implemented. The topological sorting algorithm is described in this chapter. It describes in detail the JSON data structure of a workflow and the JSON objects we used to emulate the modules. This chapter also describes the advantages of Python language and the libraries and functions of Python that we used during the implementation. It gives the general algorithm of the main program we developed. We also give an example of implementing a single module and implementing a workflow control module.

Chapter four is about the workflows we created using scientific data to test our system, it describes what scientific data we used to create different workflows in Yahoo! Pipes and then we run them locally using our program instead of running them in Yahoo! Pipes. The results of running these workflows are described in this chapter, it also shows how the aims and objectives of the project were achieved. Chapter five will summarise the findings and conclusions of the project, it will also show how this project can be extended and what other features can be added as future development.

2. Project Background

The aim of this chapter is to give an overview of the general and theoretical context of the project. It gives an overview of the scientific workflows and how they are important in the field of science. It describes some of the popular scientific workflow management systems that are used for creating workflows. This chapter also gives an overview of Yahoo!pipes definition and how they are created.

2.1 Scientific Workflow Management Systems

Science is no longer just about testing hypotheses through analysis and data collection, it is also about combining and mining the huge amount of data that is already available [1]. As the data used by scientists is growing every day, it is becoming hard to manipulate and analyse the data manually and it is important to introduce applications that help the scientist to organize their work. In this chapter we will give an overview of scientific workflow systems and their advantages, we will explain the idea of workflow reuse. Also in this chapter we will give an explanation of three popular scientific workflow management systems known in the academic field (Taverna, Kepler, Triana). We will highlight the different components of each system and how they all share the concept of a visual programming environment.

2.1.1 Scientific Workflows

Scientific work flow is a process that consists of multiple steps. This process controls multiple tasks and each task is considered a computational process [1]. Managing the data in the field of science is time and effort consuming, since it is growing rapidly. As mentioned in [1,p.137], *“From 2001 until 2009 the number of databases reported in Nuclien Acids Research jumped from 218 to 1170”*. Science includes different types of data from fields such as medicine, biology, chemistry, mathematic and many other fields which in turn includes different types of data such as: images, raw numeric data, and published articles. Scientific workflow is also considered as a documentation paradigm. It is used to keep track of the experiments and documents their results to be used by others [14].

This is why a scientific workflow is needed in order to aggregate the huge amount of data and manipulate it according to the scientist's needs. The process of designing any scientific workflow might be done by people from different backgrounds [14,15]. Scientists who are willing to produce the workflow using any graphical workflow editor may not know the details of the workflow language [15]. As mentioned in [15], there must be a separation between experts who deal with the infrastructure and the implementation details of the workflow and between the scientists who need to produce the workflow. In our project, the aim is to show that scientific workflows can be produced using Yahoo! Pipes (will be explained in section 2.2) by non-technical users, and then they can run them locally using our program instead of running them in Yahoo!Pipes.

Workflow is an ideal way to run procedures that need to be accurate and to be done repeatedly. Workflow will provide [1]:

- 1- Validating and normalizing data.
- 2- Archiving data in a secure and efficient way.
- 3- Managing data from different sources, such as sensors and instruments.
- 4- Updating the data sources.

Producing a scientific workflow needs time, effort and resources, and these workflows are often complex. So, any scientific workflow is considered a valuable piece of work, and to be able to share and reuse this work is very important [2, 14]. Workflows can be reused by other researchers, and researchers can provide new data and they can examine previous results by reusing the workflow. Yahoo! Pipes is a tool that allows the reuse and sharing of the scientific workflow by others.

There are many applications that allow scientists to publish and share their scientific work flow. An example is myExperiment. myExperiment is a virtual research environment, it provides a scientist with different shared research objects such as work flows [16]. Scientists can use this environment to share their own scientific workflows with others, or they can search for scientific workflows.

Scientific workflows are important in science fields because they allow the control and management of the huge amounts of data in different science fields; producing these workflows would be time and effort consuming if they were to be done manually. This led to the implementation of scientific work flow management systems.

2.1.2 Practical Scientific Workflow Management Systems (SWFMS)

In order to help scientists manage the huge amounts of data, scientific workflow management systems were introduced. These systems help their users to produce the workflow in an easy way. Scientific work flow systems are platforms for producing the workflow [20]. They have many advantages for scientists from different fields, as follows [20]:

- 1- They provide a simple way to manage the huge amounts of data in different ways, such as: processing the data, analyzing the data, and visualizing the data.
- 2- They help in exchanging ideas between scientists from different fields and they allow the use of data sources by a wide range of application developers and scientists [1,20].
- 3- They help scientists to concentrate on producing the workflow without worrying about the low-level details of the system.
- 4- They help scientists to produce parallel and complex tasks.

Scientists usually use Excel Sheets to organize their data, and it is an evident that scientists had to be involved in computer software in order to organize their work [17]. We can recognize from this the need for developing scientific workflow management systems (SWFMS) that allow scientists to use a visual programming environment to create the scientific workflow which is considered as easy to create by non-programmers.

In [17] it is mentioned that any workflow system can be enhanced by knowing the user's needs. They conducted an experiment including a number of biologists who were asked to draw workflows in the field of biology. The workflow they produced had major features such as data, functions and control structures; these are the common features of workflows. Scientists were asked if they would have to create these workflows and data analyses manually in a repeated process, their reply was that this process would take time and they would not have enough time to test the correctness of the data. These observations and feedbacks led to the idea of having a visual programming environment.

In the scientific field there are many scientific workflow management systems (SWFMS) such as Taverna, Kepler, Triana, VisTrails, Pegasus, Swift and View [4]. We will describe Taverna, Kepler and Triana since they are the most common ones used in the scientific fields. These systems work on local desktop computers [4] where Yahoo! Pipes is a web based system (as we will see in section 2.2). These systems are also used to create the workflows for academic fields only, and Yahoo!pipes is used in different areas.

2.1.2.1 Taverna

Taverna is an open source management system for the scientific workflow, it was launched in 2004 [3, 5]. It is a suite of tools used to execute and design scientific workflow, it is used in many scientific areas such as medical researches, astronomy, biology and music [3]. Taverna is also an environment that works on local desktop computers, which helps in producing the workflow [4].

Taverna is a tool with a graphical user interface. The workflow in Taverna is composed of a set of processors connected by data links [3] as the workflows show in Figure 2.1.2.1-1 [5] (This picture is copied from Ref.[5]). Each of these processors is used to manipulate a set of data inputs to produce a set of data outputs. Taverna, as mentioned earlier, is used in different fields of science, in [5] it is used

to create the bioinformatics workflows, which are created using (SCUFL) language, and this is simple conceptual unified flow language. It is a high level XML based language.

In bioinformatics, different data resources are needed during the experiments to produce the workflows. Examples of these sources are: EMBL or Swiss-Prot databases, also, analysis tools are needed like Blast and ClustalW [5]. Accessing the bioinformatics data and using the analysis tool is provided by different organisations based on web services [5]. These web services help in developing client applications that use the data, but scientists did not want to be involved in the programming side of the web services while using them, so this led to the Taverna project [5]. This is the main idea in the workflow management systems that we will see in this chapter.

Scientific workflow created in Taverna consists of three main parts. These are: 1) processors, 2) data links, 3) Coordination constraints [5].

- 1- Processors accept the data input and produce the data output. When creating the workflow the processors have execution status such as, initialization, waiting, running, complete, failed or aborted [5]. This status makes it easy for the user to understand the workflow creation process, and this is an advantage of the visual programming environment.
- 2- Datalinks: these represent the flow of data between data input and the data output [5]. These links are the same as the wires in Yahoo! Pipes (Figure 2.2.2-2).
- 3- Coordination constraint: these constraints are used to control and link the processors. For example, one processor can change its status to running when the status of the other processor is complete. This is required when data is needed to be executed in a certain order [5].

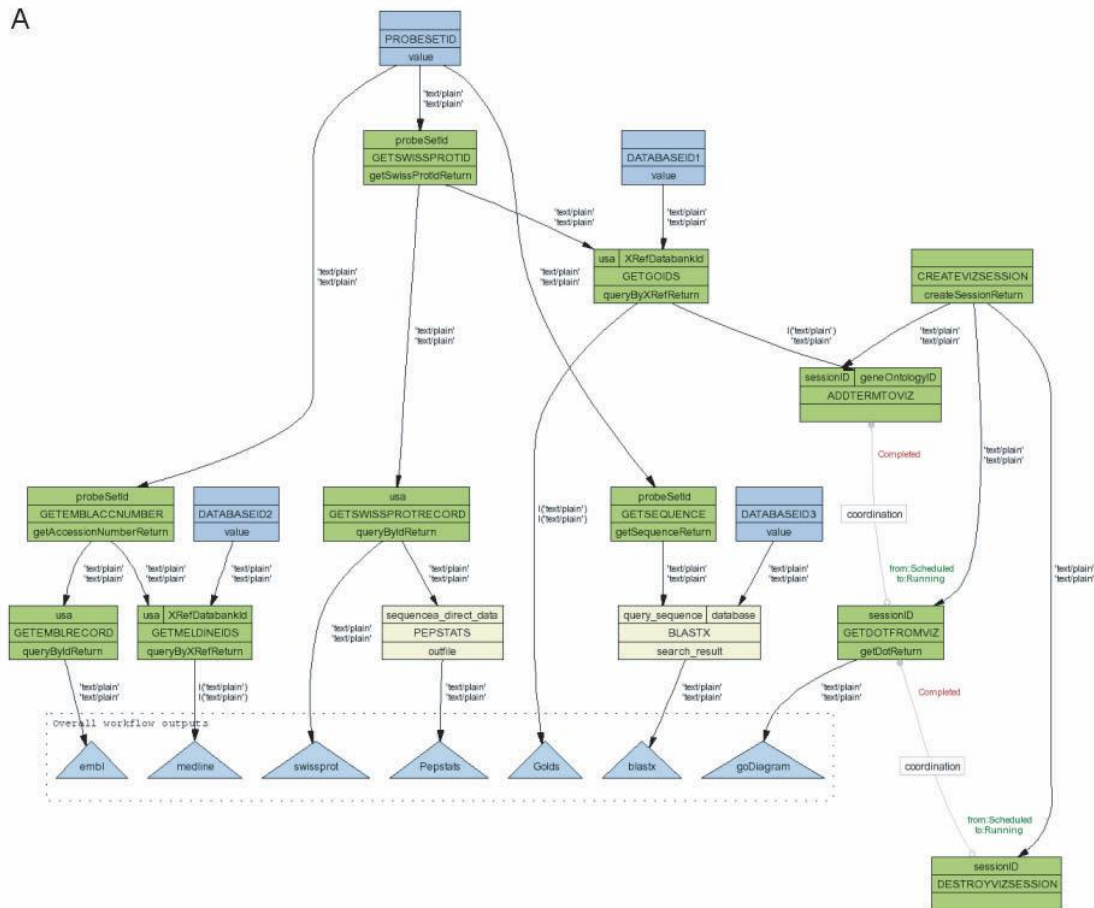
There are different types of processors in Taverna, such as the Arbitrary WSDL type, Soaplab type, Talisma type, Nested workflow type, String constant type, and the local processor type [5]. These types are similar to the modules used in Yahoo! Pipes. In Taverna there is an application called the “Scufl workbench”, this application helps the user to produce the workflow in an easy way without needing to know Scufl language [5]. As explained in [5], Taverna allows the users to view the workflow in different formats. Users can check the XML format of the workflow, they can also check the graphical format which consists of the processors connected with links. Users can also view the workflow as a PNG image using the “Graph Viz” tool. These different types of views help the users to manipulate the workflow in different ways according to their needs.

Taverna, as mentioned before, has been widely used since 2004 and there were different requirements among the years that led to the re-design of Taverna to meet them [3]. In [3], researchers focused on 2 main improvements to Taverna. The first one was the support of high-level constructors such as the while loop, and the second was to enhance the execution of the workflow for both the execution time and the amount of data to be executed. These improvements help in manipulating large amounts of data without facing any problems such as slow speed during the execution of the workflow, or the abortion of the workflow without returning any results. Such problems occur in Yahoo! Pipes while executing workflows that deal with big data sizes.

The different workflow management systems mentioned above support only single scientist to produce the workflow as mentioned in [4], but with the huge amount of data in science there is the need for collaboration between scientists from different fields to produce collaborative workflows. These workflow management systems face challenges because they don't support collaborative workflows [4]. In this research [4], researchers adapted the Taverna system to produce a collaborative system and called it Co-Taverna. Taverna was chosen because it is commonly used for creating scientific workflows using a graphical interface.

Taverna is also connected to myExperiment [4], which is a social network that allows scientists to publish their workflows and share them with others; so users of Taverna can use the specific format of their own workflow to publish it in myExperiment and others can use these workflows [4]. As we see above, Taverna is a scientific workflow management system with a graphical user interface that is helping scientists to create workflows in an easy way.

A



B

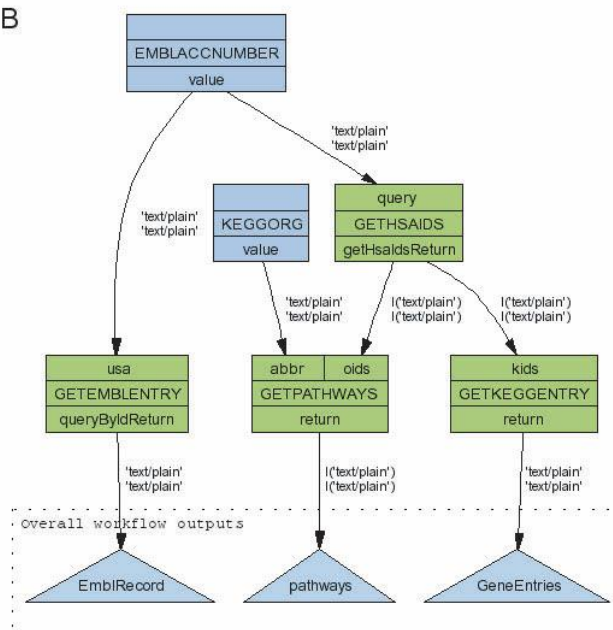


Figure2.1.2.1-1: Examples of two workflows in Taverna [5]. This picture is copied from Ref.[5]

2.1.2.2 Kepler

Kepler is another SWFMS that also allows scientists to produce scientific workflows in a visual programming environment in different fields of science such as chemistry, biology and geology. The graphical interface of Kepler makes it easy for the scientist by just dragging and dropping different components in the system to create the workflow. Kepler extends the PTOLEMYII system, this system is designed for engineering applications and concurrent systems [19].

Kepler uses components known as ‘actors’, and the workflows are executed by controllers known as ‘directors’. Kepler is a java-based SWFMS [4, 18]. Users can create the workflow using the actors and place them in the design view of the system, these actors have input and output ports and these are used to connect the actors together [19]. Each of these ports have parameters, these parameters control the behaviour and configuration of the data [21] The concept here is the same as the wires connecting different modules in Yahoo! Pipes and the datalinks in tavern. An example of a workflow created in Kepler is shown in Figure 2.1.2.2-1[19] (This picture is copied from Ref.[19]).

There are different actors in Kepler, such as, job creator actor, job manager actor, job submitter actor, and job status actor. The job creator actor specifies the input parameter and the job manager actor specifies the launching parameters. Then the job submitter actor is used to submit the job and run the submission job [22]. Other actors include, authentication actor, copy actor, monitoring actor, storage actor, data discovery actor, and service discovery actor [25]. Authentication actor is responsible for the certificate source for other actors, copy actor is responsible for copying during the workflow execution from one source to another [25]. Kepler has the ability for error recovery by using the monitoring. This actor notifies the user if there is a failure in the execution [25]. These different actors are similar to processors in Taverna and modules in Yahoo! Pipes.

The directors are responsible for the execution of the workflow. There are different types of directors in Kepler, these are some of them [19]:

- 1- Synchronous data flows
- 2- Process network
- 3- Continuous time
- 4- Discrete even

The style of interaction between actors in Kepler is modelled by modules of computation (Moc), the (Moc) represents the communication between the ports and the data flow between the actors. The (Mocs) are implemented by the directors, so any change in the directors will also change the overall execution of the workflow [25]. This is also the case in Yahoo! Pipes, if the user makes any changes to the modules, these changes will affect the entire workflow and output.

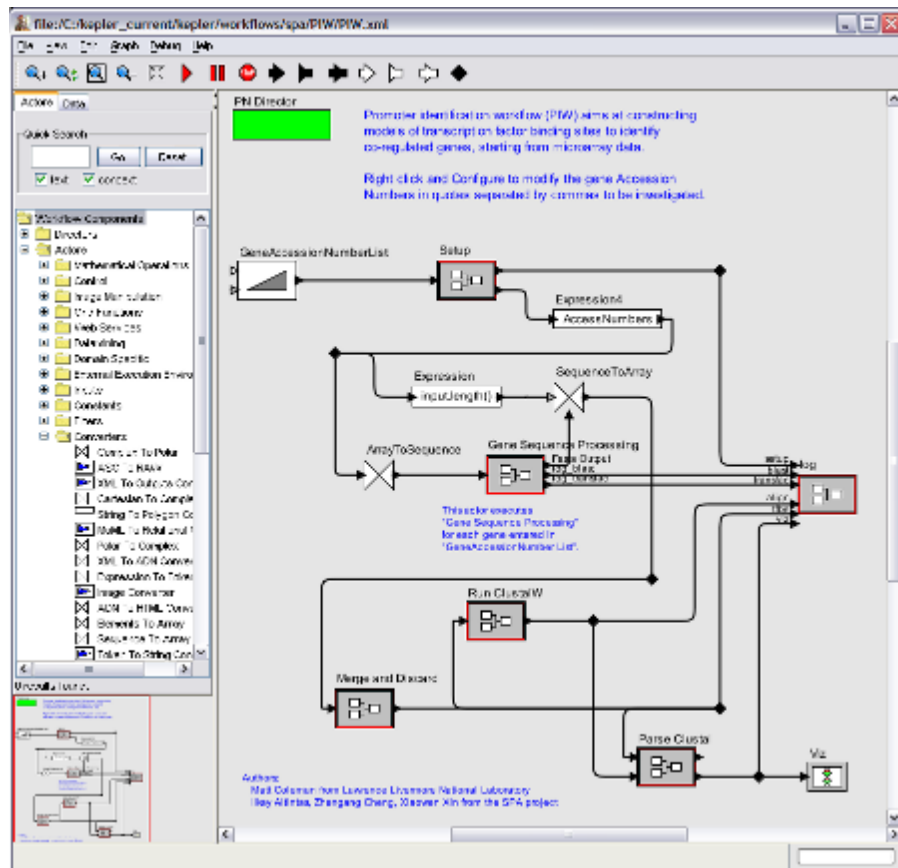


Figure2.1.2.2-1: The graphical interface of Kepler and a workflow created in Kepler [19]. This picture is copied from Ref.[19]

The workflow created in Kepler can be transformed into XML format, and the different actors run as java threads. Kepler has a web service actor that allow the users to execute any WSDL services (web services description language), it also contains special actors for executing the job in the Grid, such as Proxy Init for certificate-based authentication, and Globus Job to submit the grid job [23].

In Kepler, executing the workflow and connecting with different data sources and different data sizes (GB-TB) can result in an overhead of data transfer which makes the process of creating the workflow slow and may cause abortion of the workflow without getting the results [22]. This problem also occurs in Yahoo! Pipes when dealing with big data sizes.

Kepler is also considered as a visual programming environment for creating the workflow as it has a graphical interface with many features that helps users to create the workflow easily.

2.1.2.3 Triana

Triana is a scientific workflow management system used in different areas of science, which also has a graphical user interface that helps the scientist to create the workflow very easily without the need of any programming background. Triana was developed in 1990 by the Geo600 investigator to be a data analysis system for the GEO600 gravitational wave project [6], it contains almost 400 tools for data analysis. Triana is considered as a flexible system because it is a collection of components that are applicable to any application scenario and environment. Triana is used to produce and edit scientific workflows, and can be used also as a problem solving environment for composing,

compiling and running applications for specific areas. It is being used in many scientific areas such as datamining, radio astronomy, gravitational wave analysis and galaxy visualization [6]. Its graphical interface was designed in Java. Figure2.1.2.3-1 [8] (This picture is copied from Ref.[8]) shows a workflow created in Triana and it shows the graphical interface of Triana.

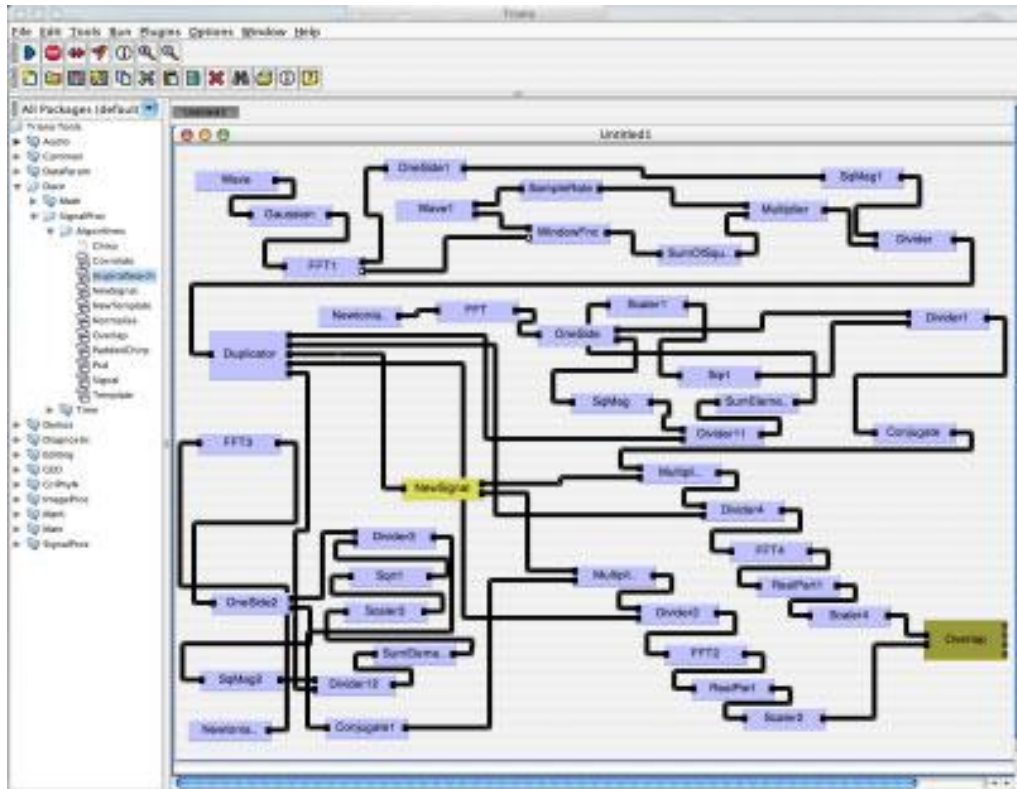


Figure2.1.2.3-1: The graphical interface of Triana and a workflow created in Triana [8]. This picture is copied from Ref.[8]

In Triana, the workflow is represented as units connected with cables. These cables represent the type of data between the units. Users can drag these units from a tool box and drop them in a workspace [6]. Scientists can create the workflow as a set of units and cables in the graphical interface or they can represent the workflow as a directory-type structure, which makes it easy to reuse these units and make any needed changes [6]. This principle is the same as what we found in other workflow systems such as Tavern, Kepler and Yahoo! Pipes, which supports the visual programming environment idea, Tavern also has many representations of the work flow, as mentioned earlier. In Triana the units and workflows are represented in XML language [8]. Each of these units consists of a process or algorithm. They have name, single process method, input and output ports and optional name/value parameters, the definition of these units is encoded in XML format, which is similar to (WSDL) format [8].

Triana consists of 3 main components [7]:

- 1- Triana services (TS)
- 2- Triana controller Services (TCS)
- 3- Triana user interface (TGUI).

This TGUI allows users to access a network of computers using the Triana system through a gateway. TCS component is used to control the execution of the Triana network and it has the option of giving the network to a Triana service (TS). The Triana service component consists of three parts: a server, a client and a command service controller. The TCS and the TGUI can connect with each other. The client in the TCS can contact the TGUI and this will result in a distributed task graph. Clients can

remotely build a Triana network by logging into the TCS, and they can see the results in their own machine [7].

The researcher in [6] explains that Triana is not considered as a data flow system, but as a workflow system, because when creating the workflow there are web services executed and the data is transferred by other data management systems. We agree with this idea because as with what we saw in the other workflow management systems (Taverna, Kepler, Yahoo! Pipes), web services are executed to create the workflow also. Triana supports different programming concepts such as logic units (if statements), and loops (repeat until, do-while), these will help in the creation of the workflow in its graphical shape [7]. These programming concepts are also supported in other workflow management systems such as Taverna and Yahoo! Pipes.

Scientific workflow management systems are important for scientists as they are needed in order for them to perform their jobs that are related to data analysis, gathering and manipulating very easily. As we found from the above description of different scientific workflow management systems, the graphical user interface makes it easy for users to create a scientific workflow without worrying about any coding or programming background. These different scientific workflow management systems are used to manage the scientific workflows and edit them in the visual programming environment. These different SWFMSs are the most popular ones in the academic field and they are used in different scientific fields, as we saw previously. Yahoo! Pipes is also a visual programming environment that helps in creating the workflows, as we will see in the next section.

2.2 Visual Programming and Yahoo! Pipes

In scientific workflows, data needs to be aggregated and manipulated in a certain way, since it contains a huge amount of data, and scientists would suffer if they had to analyse and collect the information manually, as discussed in section 2.1. In this section we will show how visual programming languages are important and how they developed, we will give an overview of Yahoo! Pipes, and what modules they contain, and we will also give an example of a Yahoo! Pipe we created during this research period. We also give a short description of how to create workflows in Yahoo! Pipes. Also, we will give a brief description of JSON data produced by Yahoo! Pipes as this will be explained in more details in chapter 3.

2.2.1 Visual Programming Languages

Research about visual programming languages started more than three decades ago [26]. In [27] visual programming languages are defined as programming languages that allow users to write the programme and then make any changes to the program elements in a graphical way rather than a textual way. The main feature of visual programming languages is that programmes written in them are represented in diagrams and graphical pictures [27].

Visual programming languages are becoming more popular, especially with users who have no programming background. The popularity of visual programming languages is because of the friendly visual interface, which helps the user to use the applications very easily. The aim of visual programming language is that non-programmers will be able to use the programming language in a different way to the known formal programming languages [26]. Another aim is to increase the performance speed of the programming tasks performed by the users [26].

There are many improvements in the visual programming languages, these improvements are in the communication features, such as the screen layout, immediate feedback and navigation tools [26]. These features make it easy for the users to use any application as what we explained in section 2.1 and as what we will explain in the next section about Yahoo! Pipes.

There are many workflow tools, such as Microsoft Popfly, Active BPEL designer, Sedna, P-Grade and A-WAR. They are considered as visual programming languages tools for creating workflows, and since the process of designing and creating workflows involves people from different backgrounds, they are not expected to know the programming details of the workflow languages, so these tools help in creating workflows without the need for any programming background [11].

The Active BPEL tool is based on Eclipse; using this tool the work flow is represented using the structure of BPEL (Business Process Execution Language), it has containers that allow the control of the workflow graph and allow the user to edit the workflow [11]. Sedna is also Eclipse based and BPEL workflow editor, it is used as a tool for creating the workflow in computational chemistry; in Sedna it is not possible to represent complex BPEL documents [11]. A-WARE is a workflow editor based on Business Process Modelling Notation (BPMN). The workflows designed in A-WARE are compiled to BPEL [11].

2.2.2 Overview of Yahoo! Pipes

Yahoo! Pipes was first introduced in 2007, and it was developed before the Microsoft Popfly tool, which is also a data aggregate tool, as mentioned in [10]. Yahoo! Pipes, as defined by Held and Blochinger that, is a workflow editor where an input string is provided using different input parameters, and an iteration process can be implemented on the input data [11]. It is also considered a visual programming environment that can be used easily by non-programmers [10]. It works by combining different modules to create the needed output.

Yahoo! Pipes contains many modules (as will be explained in the next section), and using different combinations of them will give a wide range of possibilities to merge different data feeds and use information from various resources [10]. These data modules should be easily used and understood by the users who want to create the Yahoo! Pipes depending on their needs.

There are many workflow tools or aggregators, such as Microsoft Popfly, Active BPEL designer, Sedna, P-Grade and A-war [11]. Yahoo! Pipes is considered to be a strong tool because it provides a complex structure that helps people who have programming skills to develop complex applications according to their needs [10]. So, Yahoo! Pipes can be used to produce simple or complex pipes depending on the input and output data needed.

Another strength of Yahoo! Pipes is that it can produce the output in different formats: XMLRSS, JSON data (JavaScript Object Notation) and PHP. XML does not support the arrays but JSON does support it [11]. In Yahoo! Pipes there is no need to download a software like other workflow management systems (Taverna, Kepler, Triana as explained in chapter 2), it is a web-based application. The user will only need to sign up in order to use the Yahoo! pipe. Yahoo! Pipe technology is similar to UNIX pipes where the user has to connect commands and create the output [24]. Yahoo! Pipes consists of many modules as mentioned in [9], and shown in Figure 2.2.2-1. Yahoo! Pipes has 11 categories, the following are the important module categories in Yahoo! Pipes, they are 8 important categories [9]:

- 1- Sources (fetch CSV, fetch feed, Google base, Yahoo search).
- 2- User input (Data input, text input, URL input).
- 3- Operators (count, create RSS, filter, loop, split, union).
- 4- String (private string, string builder, string replace, translate).
- 5- URL (urlbuilder)
- 6- Date (date builder, date formatter).
- 7- Number (simple math).
- 8- Location

The other three categories: favourites, Mypipes and Deprecated. Favourites allow the user to see his favourite pipes, 'Mypipes' is a list of the user's pipes and 'Deprecated' is a list of the pipes that the user no longer use them[24].

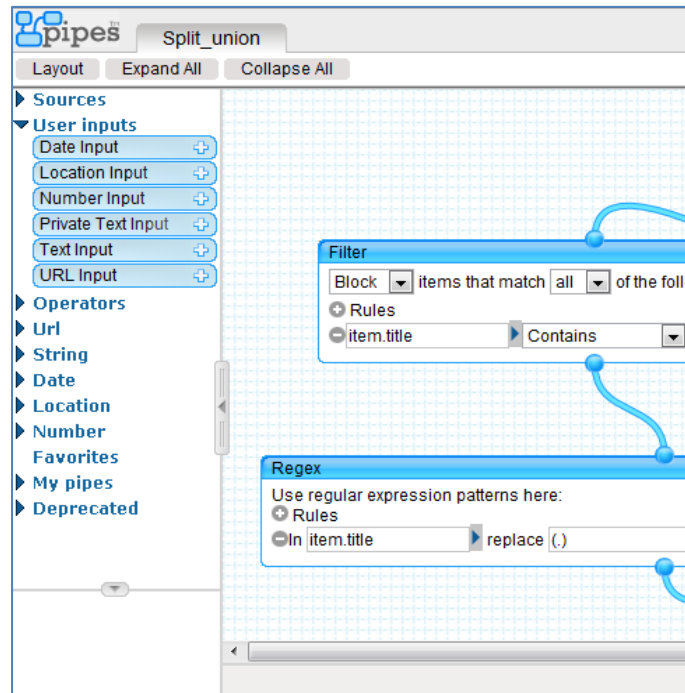


Figure2.2.2 -1: on the left hand side are the modules of Yahoo! Pipes, these are from the pipe we created during the research period in Figure2.2.2-2

We will list some of the modules with a brief description of each of these modules [9][24]:

- 1- 'Text Input': this module will allow the user to enter a text and this text will be used in other modules.
- 2- 'loop': this module allows another module to be embedded inside it, the embedded module will be executed once for each item in the input to the loop module
- 3- 'Regex': regular expression module is used to perform search and replace operations, and pattern matching.
- 4- 'Filter': this module will allow the user to filter the data (include or exclude items) by Specifying some rules.
- 5- 'Split': this module will split the input data into two identical output and the user can perform any operations on each of these outputs.
- 6- 'Truncate': this module allows the user to limit the number of items he wants to display of the data.
- 7- 'Unique': this module will remove the duplicate items from the input data.
- 8- 'simple math': this module is for implementing mathematical operations, such as: addition, modulo, subtraction, power, division and multiplication.
- 9- 'URL input': this module allows the user to enter a URL that will be used in a workflow, and this URL can be used in other modules.
- 10- 'sort': this module is used to sort the data according to a specific field.

During this project research of Yahoo! Pipes, different simple Yahoo! Pipes were created for testing and to give an idea of how Yahoo! Pipes works, Figure 2.2.2-2 shows a screen shot of a Yahoo! Pipe that was created during the research process. This Yahoo! Pipe is using the RSS feed of the University of Bristol news as the input. The data from this feed is then split into two parts of news, one for Bristol news and one for art news using the filter options in the operator module. Then the “Regex” operator is used to replace each title with a header, and finally, a “union” operator is used to combine the news to the pipe output.

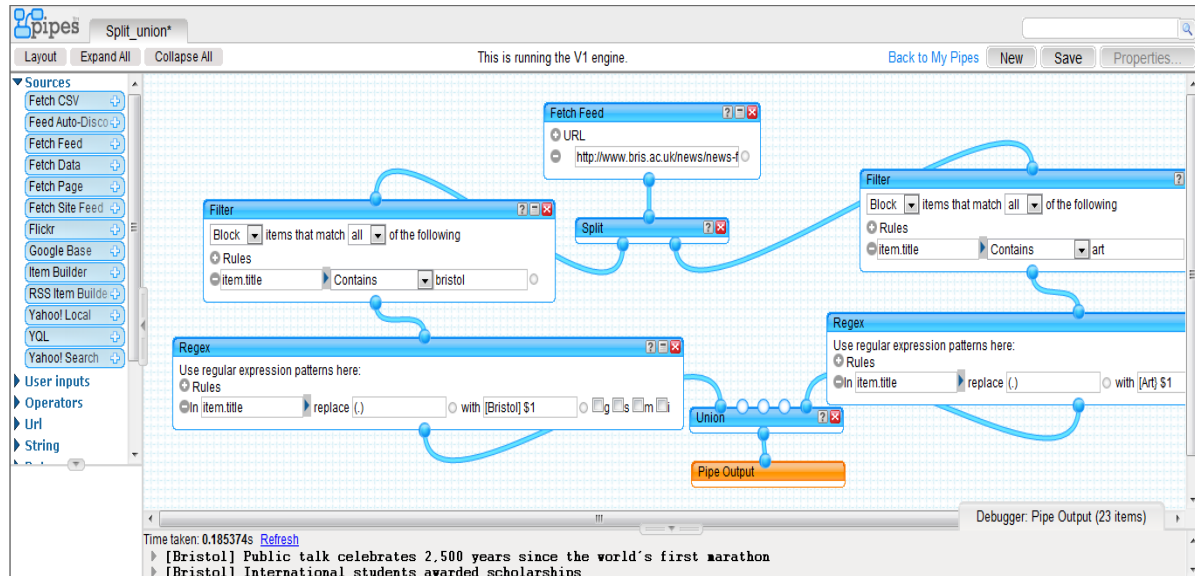


Figure 2.2.2-2: Yahoo! Pipe using the RSS feed of The University of Bristol news, this pipe we created during the research period, the following is the link of this pipe <http://pipes.yahoo.com/pipes/pipe.info?id=5dee151294047ecfab0c7fe7a194f35a>

2.2.3 Creating a Yahoo! Pipe

Using Yahoo! Pipes as a visual programming environment for creating the scientific workflow should be an easy process. The aim of using Yahoo! Pipes is that scientists can create the workflow without the need for any programming background. Yahoo! Pipes consists of the basic building block known as the “modules”, these modules are connected with each other by “wires” [24]. This is similar to the data links in Taverna, ports in Kepler and cables in Triana (as explained in section 2.1).

The following will be a brief description of the process of creating a Yahoo! Pipe. There are several main steps for creating the Yahoo! Pipe, they are as follows:

- 1- Sign up.
- 2- Choose the modules of the Yahoo! pipe.
- 3- Connect the modules.
- 4- Save the pipe.
- 5- Run the pipe.
- 6- Publish the pipe.

To start creating the pipe, the scientist (or user) must have a Yahoo ID and sign in using the Yahoo! Pipe website (<http://pipes.yahoo.com>) [24]. Scientists can also sign in using a Facebook or Google ID. They can easily click on the “Join Now” link if they do not have an ID and follow the instructions to join [24]. After signing in the user can easily access the main page of the Yahoo! pipe, and the user can click on the “Create pipe” link to open the pipe editor and start creating the pipe.

Figure 2.2.2-2 shows on the left hand side the “Module library”, in the middle the “Canvas”, and at the bottom the “Debugger”. The “module library” is a list of modules the user can choose from to create the pipe, in the “Debugger” the user can check the output of the pipe, and the “canvas” is the working area where the user can choose any of the modules and place them in the canvas to create the pipe [24]. Each of the modules has input parameters depending on the user’s needs and what data he/she wants to display. For example, the “Yahoo search” module will allow the user to make a search in Yahoo depending on what values entered in the “search for” field [24]. After creating the modules, they have to be connected using the “wires” at the button of the “pipe editor”. Above the “Debugger” there is the “Pipe output”, when the user clicks on it he/she can see the output data in the “Debugger” area [24].

After creating the pipe, the user can click on the “Save” button to save the pipe and give it a name. The final step is to publish the pipe if the user wants the pipe to be seen by others. Each pipe will be given a unique (URL) with a unique id number such as this:

http://pipes.yahoo.com/pipes/pipe.info?_id=5dee151294047ecfab0c7fe7a194f35a, this is the address of the pipe created in Figure2.2.2-2.

There are other links that will allow the user to change the properties of the pipe, edit the source of the pipe, get the (RSS or Jason) format of the pipe, or to check other pipes the user made [24]. Figure2.2.3-1 shows the output of the pipe that we created in Figure2.2.2-2. This figure shows different links in the main page where the user can use, for example: Get as RSS which allows the user to get the RSS feed of the pipe, Get as JSON link that allows the user to get the JSON representation of the pipe. From Figure2.1.2.1-1 and Figure2.2.2-2, which both show different workflows, the first one is done using Taverna and the second in Yahoo! Pipes, we can see that the workflows share the same concepts of the basic building blocks of the work connect together. In Taverna there are processors connected by datalinks and in Yahoo! Pipes there are modules connected by wires.

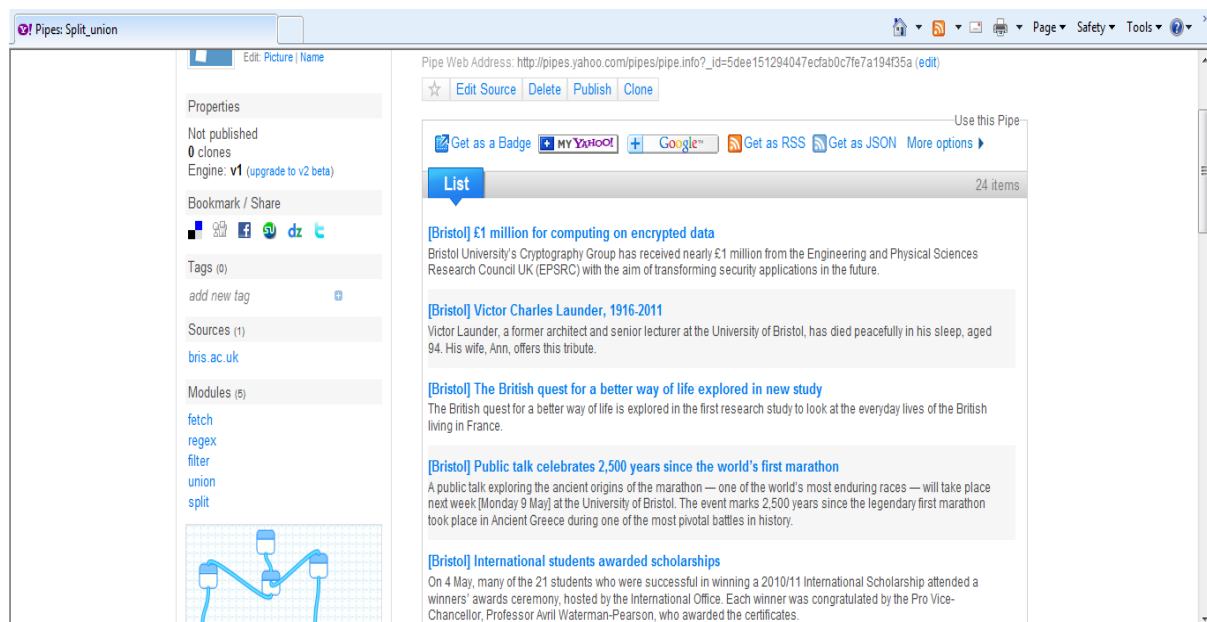


Figure2.2.3-1: This is the output of the pipes we created during the research period in Figure2.2.2-2.

2.2.4 Yahoo!Pipes structure

JSON (java scripts object notation) is a data format used to send and receive data in web applications [12]. As was found in [10], Yahoo! Pipes uses three types of data format: XMLRSS, JSON data (java scripts object notation) and PHP, and we found also that JSON data supports arrays data type while XML does not. This is also supported in [12], where it is mentioned that XML is a mark-up language that uses tags to predefine the data, this is not the case in the JSON data format.

JSON data represents the data as javascript objects and arrays. The objects are represented by curly braces ({}) and the arrays are represented by square brackets ([]) [12] as we will explain in more details in the next chapter. The JSON data format, as described in [13], is used by programmers who are familiar with C-family languages such as C++,C#,Java,Java script, Perl, and Python.

The JSON data format has many features that make it better than other data formats. These features are as follows [12,13]:

- 1- JSON data is a text-based data format that is easy to be read and be written by people and at the same time it is also easily parsed and generated by machine.
- 2- JSON is also faster during parsing than XML. This explains the reasons we choose the JSON data format created by Yahoo! Pipes, it represents the data we need to be parsed after creating the workflow.
- 3- JSON data can handle more complex structures than name/value pairs.
- 4- It also transforms the data of an object in Java script into a string, this string can be passed very easily from one function to another function

Table 2.2.4-1 [12] (This table is copied from Ref.[12]) shows the different features of the Ajax (Asynchronous JavaScript + XML) data format, Ajax is used to develop web applications on the client side. One of the data format used in Ajax is the JSON data format [12]. It shows that the JSON data format is a good representation for complex structures, and this supports our aim of running the workflow locally so that retrieving the data will be faster. The table also shows that JSON format is better used to get repose from Ajax applications and send requests from Ajax applications where XML is good only for response to Ajax applications [12].

Data Format	Features	
	<i>Request</i>	<i>Response</i>
Name/value pair	Common	×
JSON	Good	More than one parameter Arrays, complex objects
XML	Not good	More than one parameter
HTML	×	The value of one parameter, It is good for Element interaction
TEXT	×	The value of one parameter

Table2.2.4-1: Features of different Ajax data format [12]. This table is copied from Ref.[12]

From all of the above we can see that Yahoo! Pipes is a visual programming environment where the user can drag and drop different modules and connect them together to create the workflow in an easy way. We also found that the JSON data format of the Yahoo! Pipe have many advantages over other data formats. But there are some limitations in Yahoo, such as the data size to be executed and the number of queries the user can run per day [24].

2.3 Summary

This chapter presented scientific workflows and the most popular scientific workflow management systems used to create and edit workflows, such as Kepler, Triana and Taverna. It also provided an overview of Yahoo! Pipes as a workflow editor and the modules used to create different workflows. Workflow management systems (either scientific or business) share the same aim of making work much easier for users, and they aim to make it easier for most users to use the system without the need for any programming background, which will save time and effort for the users. Scientific workflow is a process that consists of multiple tasks. There are many scientific workflow management systems that were developed for a specific area and then expanded to include other scientific fields. For example, Triana was designed for the Geo600 gravitational wave project and then expanded for use in other fields.

After carrying out the research process, we found that the current scientific workflow management systems (such as Taverna, Kepler, Triana) all share the same principle of a visual programming environment. They all have a graphical user interface and they all have the basic building blocks for creating workflows. In Taverna, in order to create a work flow, scientists use processors and data links, in Kepler they use actors and directors, and in Triana users use units and cables. Yahoo! Pipes is also a visual programming environment, it has the same principle of the previous systems: it consists of modules connected with wires. Creating the workflow in Yahoo! Pipes is done using easy steps that start with signing up to the Yahoo! Pipes site and then creating the workflow. We found other workflow tools similar to Yahoo! Pipes, for example Microsoft popfly, Active BPEL, Sedna and P-Grade. These are visual programming language tools, they are used for creating workflows.

We found also that the aim of these workflow management systems is to make the workflow reusable. For example, Taverna is related to myExperiment, where a scientist can share their workflow and others can see and reuse it. This is the case in Yahoo! Pipes, where the created workflows can be published and can be seen and used by others. We also found that Taverna, Kepler and Triana have to be installed on a desktop to be used, whereas Yahoo! Pipes is web based, so scientists need only to have an ID and login to start working. It was found also that Yahoo! Pipes has limitations, such as data size to be executed and the number of queries the user can run per day. We also found that the workflow created in Yahoo! Pipes can be represented in different formats (XML, JSON, and PHP). The JSON format has advantages, it is easy to be read and written by people and is easily parsed by machine. It is faster at parsing than the XML format and can handle complex structures. In our project we used the JSON data produced by Yahoo! Pipes to emulate the workflow created in Yahoo! Pipes. In this chapter we also gave a brief introduction to JSON data that represents the workflow created in Yahoo! Pipes. More details will be given in chapter 3.

3. Scientific Workflow Interpreter Design and Implementation

In this chapter we will describe the project design and implementation. We will explain the scientific workflow as an interpreter and give a general overview of the system. We will describe the topological sorting algorithm that we implemented in our project and describe how we decide which modules to choose and implement, as we already mentioned that Yahoo! Pipes contains many modules. Also, we will give an overview of the JSON data and the most important objects of JASON data that support our process of emulating the different modules of the workflows created in Yahoo! Pipes.

This chapter will also describe how the project was implemented using the Python programming language. It will describe the main Python modules we developed that are responsible for parsing the JSON data and executing and emulating the Yahoo!Pipes modules. We will give examples of implementing single modules and control flow modules. We will also describe the advantages of Python and how it was supportive during the implementation process of the modules.

3.1 The Scientific Workflow Interpreter

In our project the program we developed will emulate the Yahoo! Pipes modules as an interpreter. An interpreter is an emulator that can execute any computations according to the source program [34]. This source program is transformed into an internal representation; according to this the internal representation (which is a type of data structure) will execute the computation. A compiler will also transform the source program into an internal representation, but will execute it as an output in the same target language [34].

Interpreters and compilers have advantages and disadvantages. Compilers are considered to be complex programs; this leads to high development and maintenance costs. An interpreter is simpler and easy to develop [39]. We designed our program as an interpreter, such that it will be easy for the user to execute the program from the command prompt only once by entering the workflow id, as we will explain. In the case of a compiler, the user would have to run the program which will produce a Python script containing all the modules of the workflow, and then the user have to run this script again to produce the output.

3.2 System General Overview

A general overview of the project system is shown in Figure 3.2-1. The figure gives an idea of how the system works, the steps for implementing the project and how each step is important during implementation for the next step. The following is a general explanation of each of these steps:

- 1- Parse JSON data:
The program first will open the workflow's URL and parse the JSON data. From the JSON data we will discover what modules each workflow consist of.
- 2- Sort the modules:
Then the program will sort the modules using the topological sorting algorithm (this will be explained in the next section). The workflows cannot be executed without specifying the order of executing the modules and this order will be determined by this algorithm.
- 3- Execute the modules:
After sorting the modules they will be executed according to each module's JSON parameters (this will be explained in more detail in section3.5).

- 4- Print the output:
When the 'OUTPUT' module is reached it means the end of the workflow has been reached, so the output is then displayed on the screen.

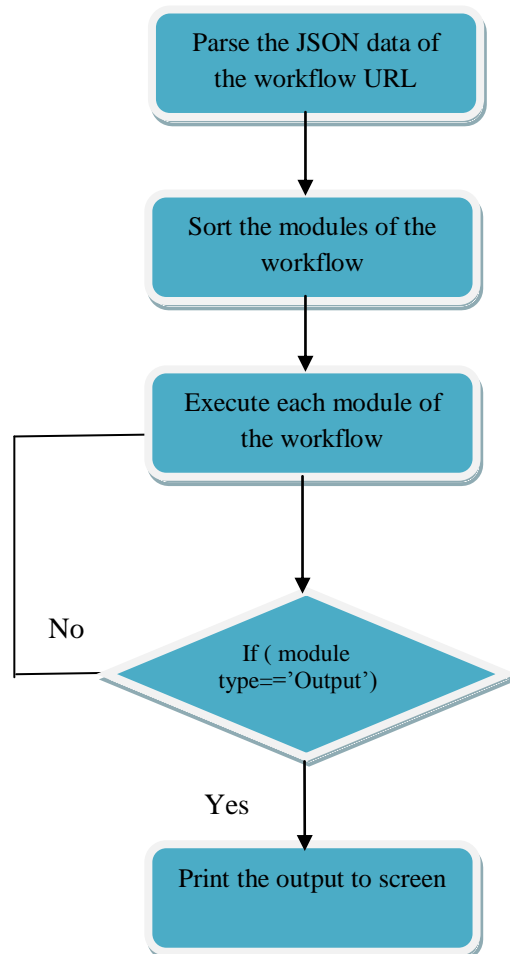


Figure3.2-1: General Overview of the System

During the design stage we had to consider some problems that might occur during the running of the program. For example, we set a timeout limit of 10 seconds, if no connection could be made to the URL of the workflow in this time then the program would stop running and a message would appear for the user showing that there is no response from the URL source. When parsing the JSON data some errors might occur, so error handling is done to check if any exceptions or errors might occur during parsing.

There are also some modules that need data to be entered by the user, such as the 'numberinput' module, which will ask the user to enter a number, if the user enters anything other than digits, or if he enters a negative number, then a message will appear asking to enter the correct data. The 'urlinput' module will ask the user to enter a URL and will display a message if the string entered by the user is not the correct URL string.

Some modules deal with date format: the 'dateformat' and 'dateinput' modules, it is possible that the user might enter an invalid date format, so this is handled by error messages. Some of the URL date

fields use different time zones that cause value errors during the implementation, so this was handled by applying other date formats for these exceptions. As we will explain later we need to save the output of the workflow as JSON format. Some of the output data fields, such as the (date) field, cannot be serialized to JSON data and will raise a type error. This problem was dealt with by changing the values causing the problem into a JSON date format.

3.3 Topological Sorting Algorithm

In our project we needed to find an algorithm for sorting the modules in order to execute them. The workflow is presented as a directed graph that consists of vertices (modules) and edges connecting these vertices (wires). There are many algorithms in the graph theory for directed graphs, such as the topological sort algorithm, Kruska's algorithm and Dijkstra's algorithm [31]. We chose the topological sort algorithm to sort the modules. The topological sort algorithm for a directed acyclic graph (DAG) consists of vertices and edges $D = (V, E)$, this algorithm will map each vertex to a priority order such that $ord_D(x) < ord_D(y)$ [32] where x and y are vertices.

The modules in JSON data will be order randomly, the topological sort algorithm is used to organise the tasks according to the precedence constraints, so, for a set of tasks, some must be completed before the execution of others. This algorithm will determine the order of the modules of the workflow (vertices), where these modules cannot be executed before the previous module is executed. This provides us with a method of executing each of these modules in our program. The modules of the workflow are saved in a list with their attributes. The following is the topological sorting algorithm:

```
def topological_sort(graph):

    Define a new list (N)

    Let a list (L) contains all the modules of the workflow

    While (L) is not empty:
        -remove the module (m) from the list (L)
        - insert the module (m) to the list (N)
    -check that the module has incoming edges or not:
        for each module (n) having an edge from m to n:
            -remove the edge
            - if no more incoming edges from n then:
                Insert (n) to the list (N)

    Return list N
```

This algorithm takes the JSON modules object (will be explained in section 3.5) and sorts the modules of the workflow. It will define first and empty list, it will loop through the list of modules in the workflow and check that the modules having incoming edges (wires) from other modules, and then remove the edges and insert the module to the new list. At the end of the loop we will have a list of sorted modules that will be used for the execution.

Figure3.3-1 shows a workflow that consists of different modules (three fetch modules, a union module, a split module, two truncate modules and the output module). When running this workflow using its id in our program, these modules will be sorted first and then be executed according to the

sorted list of modules. We used the topological sort code in [35] [38] in our program in order to perform the module sorting process, we imported the 'topological_sort' function in the Python module 'pipe_data.py' that we will explain in section 3.6. Figure 3.3-2 shows the sorted modules in Figure 3.3-1. It shows the module id and the module type (name).

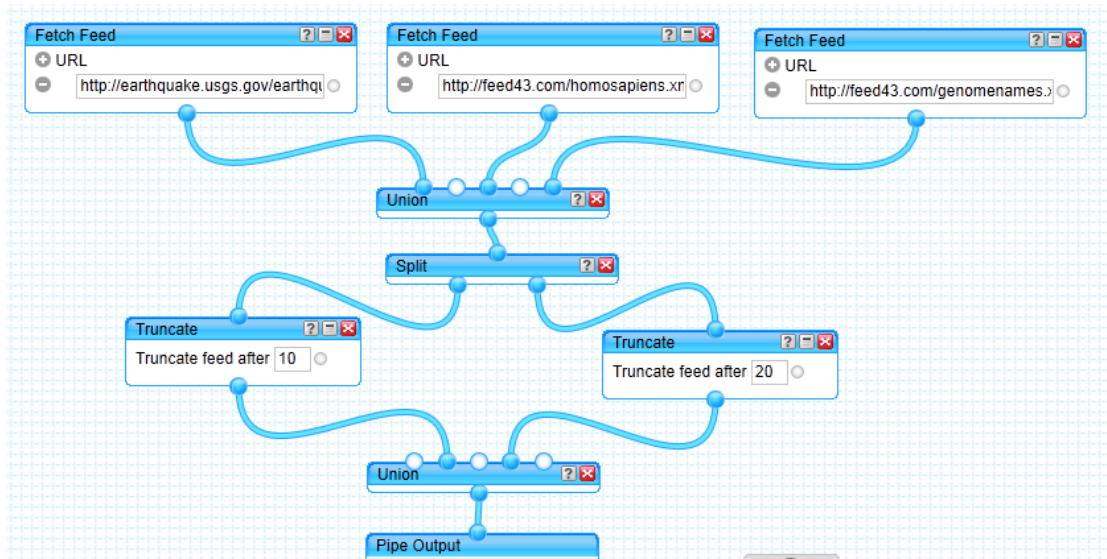


Figure3.3-1: This is the workflow we created in Yahoo! Pipes

```

Administrator: Command Prompt
conf:
  type: union
  id: sw-254
conf:

the sorted Graph is:
Module id: sw-319 , Module type: fetch
Module id: sw-327 , Module type: fetch
Module id: sw-195 , Module type: fetch
Module id: sw-214 , Module type: union
Module id: sw-244 , Module type: split
Module id: sw-233 , Module type: truncate
Module id: sw-203 , Module type: truncate
Module id: sw-254 , Module type: union
Module id: _OUTPUT , Module type: output

```

Figure3.3-2: This is the sorted modules of the workflow in Figure3.3-1

3.4 Modules chosen

An important part of our project is to decide upon the most useful modules we want to emulate, because this decision will effect what workflows we can create in Yahoo! Pipes, and then run them locally using our program. Yahoo! Pipes has 11 categories of modules, each one has a different number of modules (as explained in chapter 2). In chapter 2 we found that Yahoo! Pipes has 8 important categories, the total number of modules in these categories is 48, so we had to decide which modules to emulate.

We checked almost 50 pipes (workflows) which are considered to be popular in Yahoo! Pipes. In order to help us make the decision, we wrote a program in Python that will check each module in each of these popular pipes and count them, then it will count the total number of modules in all of the popular pipes. We then used Microsoft Office Excel to produce a column chart which shows the module name in the x-axis and the total number of each of these modules in the y-axis, as shown in Figure 3.4-1. This figure shows the most used modules in the popular pipes.

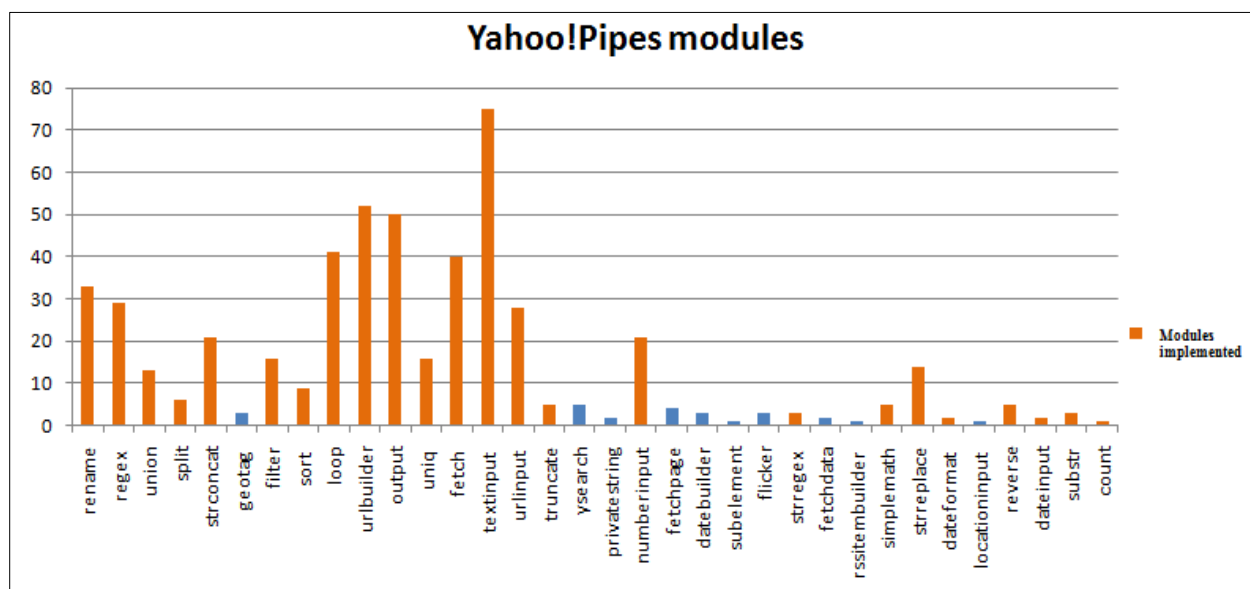


Figure 3.4-1: The most popular modules used in Yahoo! Pipes

The decision of which modules to emulate didn't depend on the chart results only, it also depended on the following factors:

- 1- Priority:
Some of these modules are prioritised over others because we had to use them during our implementation phase of the pipes. So even if they were not popular we had to implement them first (for example the split module and the union module).
- 2- Complexity:
Some of these modules have a simple structure and do not require much time to be implemented, but others have a complex structure with different parameters and will consume time in their implementation (such as loop, filter, regex, urlbuilder, dateformat modules).
- 3- Popularity:
Some of these modules, according to Figure 3.4-1, are popular and used a lot in different pipes, but for us they might not be useful to implement (flicker and privatestring module).
- 4- How useful is the module:
We have to choose modules that are useful for our project and when we implement them they will be useful even if they are not very popular. For example, in Figure 3.4-1 it is shown that the ranking of the split module is not high, but in our project we decided that it would be useful to implement this module as it will be involved in creating different workflows.

According to the results of Figure 3.4-1 and to the previous factors we have chosen the modules that we implemented, they appear as the orange columns in the figure, we implemented 24 modules. The modules that we didn't implement appear in blue columns either because of time factor and priority or because some of them were not going to be of use for us in the project, such as the 'flicker' module, which is for searching for images from the flicker site, and the 'privatestring', which allows the user to enter passwords.

3.5 JSON Data Structure of Workflows

This section will describe the JSON data structure of the workflows and the JSON objects we used for the implementation. As we found in chapter 2, JSON (Java Scripts Object Notation) is a data format used to send and receive data in web applications. JSON data represents the data as java script objects and arrays. The objects are represented by curly braces ({}) and the arrays are represented by square brackets ([]). In our project we needed to know the structure of the workflows (pipes) and the modules they consist of.

JSON data of any workflow consists of different objects and arrays. The most important objects for us are the 'modules' objects that are illustrated in Figure3.5-1, and the 'wires' objects, as shown in Figure3.5-2. These two objects can be found in the main object 'PIPE 'working' of the workflow. The 'modules' object contains all the modules and attributes of the workflow. For each module there is a unique module id and 'conf' object which contains the information that we used in our program to parse the JSON data and execute the modules of any workflow.

As shown in Figure3.5-1, this is part of the JSON data of a workflow that consists of a fetch module, a sort module, a filter module and an output module. The JSON data of any workflow is in the form of nested objects of lists and dictionaries, and it would be difficult to find the needed data without formatting them. There is a tool provided by Yahoo called Yahoo Query Language 'YQL', this tool helps to filter, query and join data across the web services. YQL allows the developers to shape and access the data across the internet through a simple language: SQL [33]. This tool helped in viewing the JSON data in a well format that allowed us to easily read and check the parameters of each module and to know what parameters of the module we will use in the implementation. We wrote an SQL statement to retrieve the JSON data of any work flow and run it in YQL, the result is formatted JSON data. The query we used is as follows:

“select PIPE.working from json where

url=http://pipes.yahoo.com/pipes/pipe.info?_out=json&_id=e7768a3c40d8c4786af8a13d7b1ed6c0”

From this statement we can see that we provide the 'id' of the workflow in order to get the JSON data.

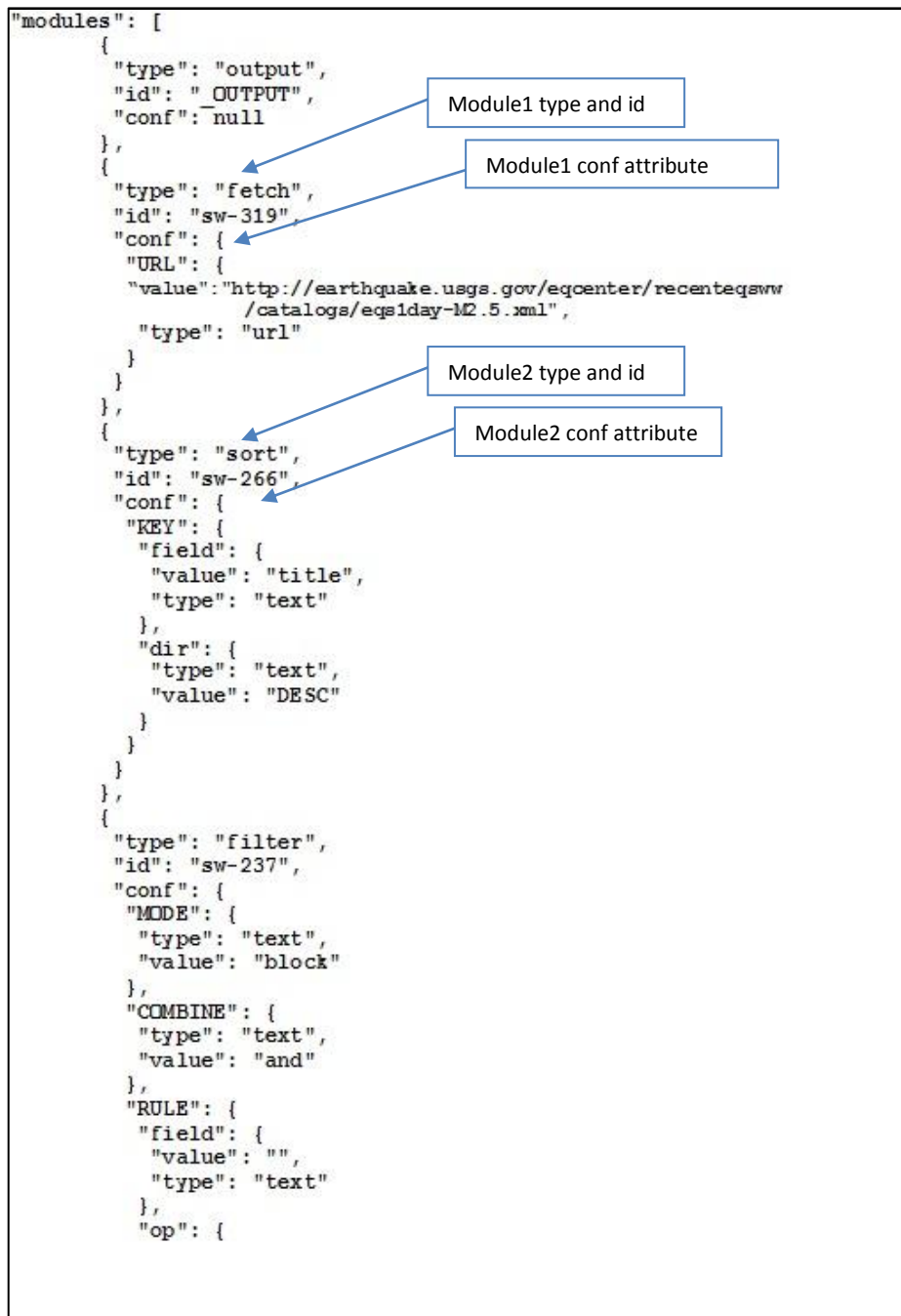


Figure3.5-1: This is the 'modules' object of the JSON data of a workflow.

For example, the fetch module as shown in Figure3.5-1 consists of:

- 1- "type": "fetch",
- 2- "id": "sw-319",
- 3- "conf": {
 "URL": {
 "value": "http://earthquake.usgs.gov/eqcenter/recenteqswww/catalogs/eqs1day-M2.5.xml",
 "type": "url" }

This data gives us the values we need for this module, it shows that it has the type 'fetch', which is also the module name. The 'id' value is the unique number each module has and here it is 'sw-319' for the fetch module. The 'conf' object, which is a very important object of each module, has the value 'URL', this shows the URL value in order to fetch the data from it. This is the same case for each of the modules. So the 'conf' object of each module will tell us what are the parameters we will use for emulating and executing each module as we will explain in the next section.

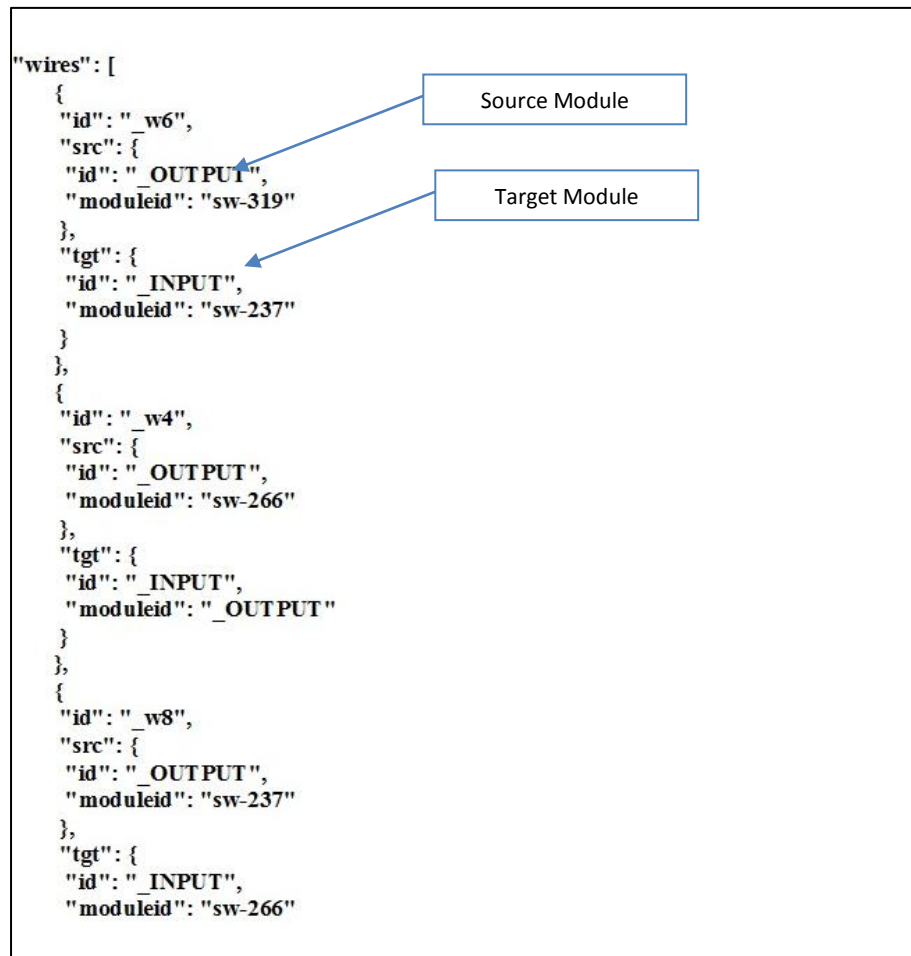


Figure3.5-2: This is the 'wires' object of the JSON data of a workflow.

The 'wires' object illustrates how the modules of the workflow are connected. Each wire has a unique id and consists of the source and target modules. For example, from Figure 3.5-2, the wire with id '_w6' has the following attributes saved as a dictionary of items:

```

{
  "id": "_w6",
  "src": {
    "id": "_OUTPUT",
    "moduleid": "sw-319"
  },
  "tgt": {
    "id": "_INPUT",
    "moduleid": "sw-237"
  }
}

```

This shows us that the wire has an id of "_w6", each wire consists of a source item and a target item. Each source and target has an 'id' and 'moduleid', so this tells us that the module with the id 'sw-319', which is a fetch module, as illustrated in Figure 3.5-2 is the source module, and the output of this module will be sent to the target with the id 'sw-237', which is a filter module. The 'wires' object was used during the implementation process to determine the output of each module that will be the input to the target module.

3.6 Scientific Workflow Interpreter Implementation

This section will discuss how the project was implemented using the Python programming language and how Python was supportive during the implementation process. It will describe the main Python modules we developed that are responsible for parsing the JSON data of the workflow, and the one responsible for executing each of the 24 modules that we emulated. We will then give examples of implementing different modules out of the 24 modules.

The implementation of this project was carried out in three steps:

- 1- Open the URL of the workflow, parse the JSON data of the workflow and find out what modules it consists of.
- 2- Sort the modules in order to execute them.
- 3- Write the functions that will execute each of the modules.

We will explain these steps in more detail and show what Python modules and functions we used during the implementation.

3.6.1 Python Language and Implementation

As we found in chapter 2, different scientific workflow management systems produce the work flow in XML format or JSON format. In our project we will use the JSON format to emulate the workflow created in Yahoo! Pipes. JSON data can be parsed using different languages such as Java, Perl, Python, Java script, and Ruby. Since the JSON data consists of objects and arrays, we found that a good choice would be Python, because it supports different data structures.

Python has many features; here we will list some of them and explain why we have chosen it for our project [36]:

- 1- High level data structures:
Python is considered to be a high level data structure language that decreases development time. Python provides useful data structures like lists (arrays) and dictionaries (hash table), these are built into the language.
- 2- Portable:
Python is written in C, and because C is portable then Python is also portable. Any program written on Python on one system will run on another system with little or no modification.
- 3- Easy to read and learn:
Python is easy to learn because it has a simple structure, few keywords and very clear syntax, so anybody can learn it in a short time. Python doesn't include the mandatory symbols found in other languages for code block definition and pattern-matching, such as the dollar sign (\$), semicolon (;) and the tildes (~), this makes the Python code easy to read.

4- Memory management:

With some languages, like C and C++, memory management is the responsibility of the programmer, where the programmer has to handle memory access, modification and management. In Python, memory management is handled by the Python interpreter, so the programmer can focus on the application development.

5- Extensible:

As the Python code increases during any project implementation, the programmer can still organize it by splitting the code into different modules or files; he can then access any module from another one.

The computational tools needed in science should be able to deal with various problems - not just numerical problems. They should be able to deal with different formats from large datasets, and deal with computational systems like the internet and database servers [29]. Python is an open source programming language, it can be used to build a graphical user interface and can be active script compatible in Windows [28]. The issues we mentioned before cannot be solved by using simple arrays and simple mathematical operations. In the previous chapter we mentioned the structure of JSON data (objects and arrays); we can have multiple values within other multiple values.

Python provides rich data structures that solve problems that cannot be tackled with the normal known data structure, as the solution could be too complex [29]. Python supports different data types such as strings, variable size lists, arrays which are known as “dictionaries”, integers, floating points and complex numbers. Python libraries allows the user to process text, send data and interact with web servers and databases. It also helps in creating specialized objects for scientific workflows, such as the “Numpy” arrays, which are a multi-dimensional array [29]. The rich data structures in Python will help us manipulate the JSON data of the workflow. Codes in Python can be written in a simple and direct way due to the libraries it contains; it also supports data visualization by using the “Matplotlib” [29].

3.6.2 The Scientific Workflow Interpreter Main Modules

As mentioned in the previous section, Python has many supportive modules that help during the implementation process. In this section we will describe the most important Python modules and describe the main Python modules that are used for parsing the JSON data and executing the Yahoo! Pipes modules.

Python allowed us to use the command line where the user has to enter the workflow id. Python has the ‘argparse’ module that support writing user friendly command line, this module also generates usage and help messages when the user inputs an invalid argument [30]. Creating an argument parser is done by following these steps:

1- Create ‘ArgumentParser’ object:

```
parser = argparse.ArgumentParser(description=' This programme is for emulating the workflows created in Yahoo! Pipes')
```

This ‘ArgumentParser’ class will save all the information needed to parse the command line as a Python data type.

2- Add arguments:

This is done by using the ‘add_argument()’ method. This method will send the information of the program argument to the ‘ArgumentParser’. It will show how to take a string from the command line and change it to an object.

```
parser.add_argument('-id', help='enter the workflow id to run the code')
```

```
parser.add_argument('-s', dest='outfile',help='saving the result to a file, here you enter the file name as .txt').
```

3- Parsing arguments:

Python has the 'parse-args()' method that allows the 'ArgumentParser' to parse the arguments. From the command line each argument is converted to the suitable type and the correct action is executed using the parse-args()' method [30].

As shown in Figure3.6.2-3, the user has to enter the command line and passes the workflow id to the program to run the workflow.

In our project we first need to open the URL of each workflow, Python has the module 'urllib' that allows us to open any URL and read its data by using the function 'urlopen'. Figure 3.6.2-1 shows the algorithm of the Python module 'pipe_data.py' that parses the JSON data of the workflow, sorts and then executes its modules.

```
import urllib,json

def pipe_data (urlid):
    global pipe_json

    - open the URL
    - read the URL
    - load JSON of the URL

def main():
    global ur_lid
    global pipe_struct
    global sorted_graph

    - parse the arguments entered by the user and get the URL id
    -call the 'pipe_data' function and pass to it the URL id:
    pipe_data(ur_lid)

    - Call the 'modules_graph' function and pass to it the JSON data of the workflow
    - Call the 'modules_top_sort' function and pass to it the modules to be sorted
    - Call the 'check_excute' function and pass to it the sorted modules and the JSON data

    if (parse_argument=='-s'):

        save the workflow output to a text file
```

Figure3.6.2-1: The algorithm of parsing the JSON data of the workflow, sorting the modules and then executing them

The 'urllib' module is imported in order to call the functions 'urlopen' to open the URL and read from it. Here, 'urlid' is the id of the workflow, as we mentioned in chapter 2, each workflow has a unique id. Figure 3.6.2-1 shows that after opening the URL, the JSON data is parsed and saved to a variable. This variable will be sent to another function 'modules_graph' that will get the needed JSON objects for the execution process. The JSON objects will be sent to the function 'modules_top_sort' that will sort the modules using the topological sorting algorithm that we described in section 3.3. Finally, the 'check_excute' function will execute each of the modules and print the output to the screen.

The following code lines are used to open the workflow URL:

```
Import urllib
url='http://pipes.yahoo.com/pipes/pipe.info?_id='+urlid+'&_out=json'
url_response = urllib.urlopen(url, ,timeout=10)
url_content = url_response.read()
```

Then we needed to parse the JSON data of each workflow. Python has the 'json' module that allowed us to parse the JSON data by using the function 'load', which deserializes a variable to a Python object [30]. Python also has many supportive modules or libraries that helped us implement the different modules of Yahoo! Pipes. We will describe here the most important modules and built-in functions.

The following code lines show how to parse the JSON data:

```
import json
data = json.loads(url_content)
json_data=json.loads(data['PIPE']['working'])
```

The first line will load the JSON data and save it in the variable 'data', the second line will load the object that contains the modules of each workflow (pipe), as we mentioned in the previous section, the object we need to parse from JSON data is the 'modules' object, this object is the 'pipe''working' object. These objects are treated as dictionaries and the arrays as lists. Dictionaries and lists are types of data structures that Python supports as we mentioned above. Python also provides the 'dump' function that will serialize an item as JSON format, we used this function to save the output of the workflow in JSON format, we will explain this later.

Within the Python module is a file containing a Python code, and for each Yahoo! Pipes module (such as fetch, split, union, sort, truncate...) we wrote a Python module named after the Yahoo! Pipes modules. For example, the Python module that is responsible for emulating the 'fetch' Yahoo! Pipes module is 'fetch_module.py', and for the 'split' module it is 'split_module.py', and so on. Each Python module consists of a function that will emulate what a Yahoo! Pipes module is performing. All of the Python modules are saved in one folder and are imported in the main program, as Figure 3.6.2-2 shows. Now we will describe the Python module that is responsible for calling the functions to execute and emulate the Yahoo! Pipes modules. The following is the general algorithm for the main program that is responsible for the execution process of the modules:

```
Import python modules
def check_excute (pipe,sorted_graph):

For module in sorted module list:

    - Type=module['type']
    - Conf=module['conf']

    For each wire in the wires
        If ( module = wire_tragetid):
            -retrieve the data of the source module as the input of the target module
            -execute the module
            -save the output in a list
    If ( module type='Output'):
        -Print the final output to the screen
        - return the final output to be saved as txt file
```

Figure 3.6.2-2: The General Algorithm of Executing the Modules

After sorting the modules of the workflow, we now know what modules the workflow consists of and the execution order of these modules. Figure 3.6.2-2 illustrates how our program executes or emulates the Yahoo! Pipes modules of any workflow. The Python modules that we wrote and that are responsible for the execution process are imported. In this figure there is a function called 'check_excute', this function has the parameters (pipe and sorted_graph), 'pipe' is the list of the JSON data of the workflow and 'sorted_graph' is a list of the sorted modules. As mentioned in the previous section, the Yahoo! Pipes modules in JSON data are sorted, therefore the sorted list of modules is used here. For each module from the sorted list (such as: fetch, split, union...) the program will get the 'type' and 'conf' parameters. These parameters are from the JSON data that we parsed. The JSON 'wires' will also be used here; they will help in managing the process of passing the data from one module to another. Now we have the main parameters required for executing the module: type, conf and the data from the previous module, so the module is executed using these parameters and the execution output is saved in a list.

If the program reaches the 'output' module, this means that this is the end of the workflow and the final output is printed to the screen. We discussed earlier that the program will parse the arguments the user enters; there is also one option the user can type which is the '-s' argument, this will allow the saving of the output of the workflow as a '.txt' file, as follows:

```
pipe_code_sort\pipe_data.py -id 754ffe7c8ad36b2f3823579817097191 -s workflow.txt
```

Where 'workflow.txt' is the name of the text file the user specifies.

As we mentioned before, Python has many libraries and functions that we used when emulating the Yahoo! Pipes modules; we will mention some of them here. For emulating the 'urlbuilder' module of Yahoo! Pipes, Python has a module (library) known as 'urllib' that has the 'urlencode' function. This function converts an object to an encoded string that is suitable to pass to the function 'urlopen' to open any URL [30]. There is also another Python module that has a role in parsing the URLs, it is 'urlparse', this module is used to break any URL into its individual components, such as: network location, path, and addressing scheme. It matches the internet RFC of any URL and supports different URL schemes: http, https, ftp, file, telnet and many others. This module consists of the function 'urlparse', this function will parse any URL passed to it into six components. These components will match the general structure of a URL: scheme://netloc/path;parameters?query#fragment [30]. We used this module to make sure that the user enters a valid URL.

There is also the 'datetime' modules which contain 'strptime' and 'strftime'. These functions convert a string representing a time to a string time with a specified format passed to the function [30]. This function helped to implement the 'dateformat' module of Yahoo! Pipes. One of the most helpful modules in Python is the regular expression module known as the 're' module. This module provides an operation that matches regular expressions which we needed to implement the 'regex' and 'strregex' modules of Yahoo! Pipes. There are also many other built-in functions (sorted, all, any, sum) that were helpful during the implementation of different modules.

From Figure 3.4-1 in the previous section, and according to the factors we discussed earlier, we have implemented 24 modules. As illustrated in the figure, the modules we have implemented are represented as orange columns. We will give examples in the next section of some of these modules and how they were implemented. We will explain also how the JSON data of any workflow has a role in the implementation. We will focus on the JSON object of the modules and how we used their parameters in order to execute the module.

We will give an example of how a single module is implemented, and another example of how the control flow module 'loop' is implemented. We started by creating a simple or basic workflow which consists of two modules (fetch and output). This start helped us in understanding the JSON structure of the workflow and using it in order to emulate any workflow created in Yahoo! Pipes. Then we implemented other modules that helped us create more complicated workflows involving different modules and then run them locally.

This collection of implemented modules will allow the user to create different workflows according to their needs. Figure 3.6.2-3 shows the steps of running the program. The user will first create the workflow in Yahoo! Pipes, each workflow will have a unique id. The user will enter this id in the command prompt and run the workflow, if the user has to enter any value that will be used during the execution process a message will appear asking for this value. Finally, the data of the workflow will be displayed on the screen.

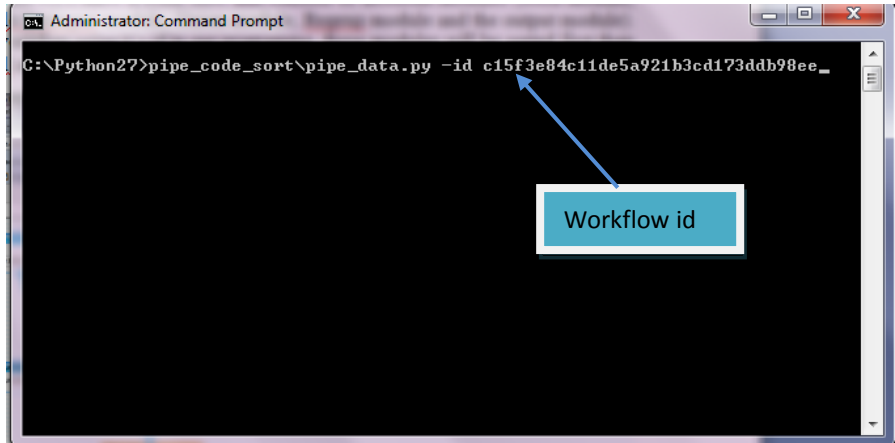

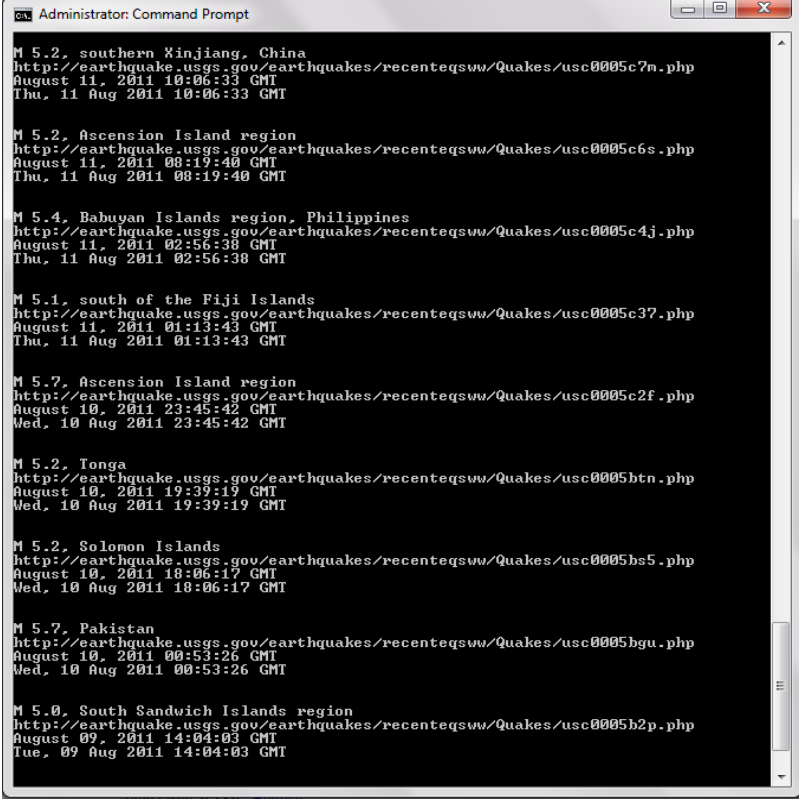
<p>First the user will enter the id of the workflow created in Yahoo! Pipes in the command prompt.</p>	
<p>If the user needs to input some data then a message will appear asking for the input.</p>	
<p>Then the output of the workflow will be printed on the screen.</p>	

Figure3.6.2-3: Steps of running the program in order to run the workflow locally

3.6.3 Interpreting Single Module

In the previous section we explained how our scientific workflow interpreter works and executes the Yahoo! Pipes modules. In this section we will give an overview of how a single module is executed. We will also show how the JSON parameters are used to execute the module and produce the output. We will give an example of executing the 'textinput' module and the 'filter' module, and describe the execution of the 'loop' module.

3.6.3.1 Text Input Module

As illustrated in the previous section, the JSON data of any workflow is important, and we explained the essential JSON objects that were involved in the execution process, such as the 'modules' and 'wires' objects. Here we will explain in more detail the parameters of the JSON data used in the execution. We will describe the 'text input' module which is used to allow the user to enter some text. Each Yahoo! Pipes module consists of the 'conf' object that holds all the parameters of the module. These parameters are involved in the execution process. If for example we have a workflow that asks the user to enter a word to search for in a specific data, as in Figure 3.6.3.1-1:

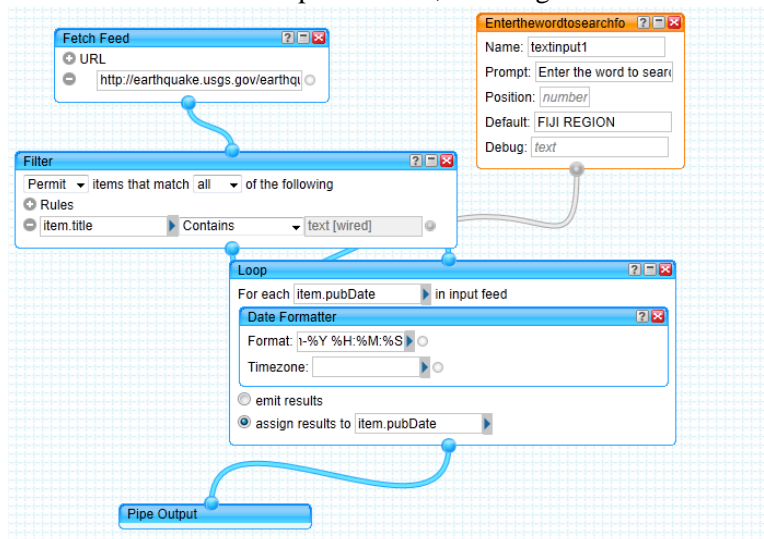


Figure 3.6.3.1-1: Workflow containing a 'Text Input' module

The 'conf' object of the 'Text input' module is the following:

```
"type": "textinput",
  "id": "sw-235",
  "conf": {
    "name": {
      "value": "textinput1",
      "type": "text"
    },
    "prompt": {
      "value": "Enter the word to search for",
      "type": "text"
    },
    "position": {
      "value": "",
      "type": "number"
    },
    "default": {
      "value": "FIJI REGION",
      "type": "text"
    }
  }
```

The 'conf' object of the 'textinput' module will be passed to the Python module 'textinput_module.py' as it is described in the previous section, this object consists of the conf['prompt']['value'] and conf['default']['value'] parameters. The 'prompt' parameter is a message that appears asking for the value to be entered by the user. The 'default' parameter holds the default value that will be used if the user pressed enter without entering any value. The algorithm of the Python module of 'textinput' is:

```
def textinput (conf,out_result,input_to,gsorted)

    -get the values from the 'conf' object

        -prompt=conf['prompt']['value']

        -default= conf['default']['value']

    text=raw_input(prompt,default)

    If text ==Null then:

        text=default

    return text
```

The python module 'textinput_module.py' consists of a function called 'textinput'. It will check the parameters of the 'conf' object. These parameters are saved in the variables 'prompt' and 'default'. The 'raw_input' is a built-in function from the Python library. It displays the message for the user then reads the data the user enters, converts it to a string and then returns this string. If the user didn't enter a value then the default value is used and returned by the function.

3.6.3.2 Filter Module

Filter module allows the user to exclude or include items from the data according to his needs. The following is the 'conf' object of the filter module in Figure 3.6.3.1-1:

```
"type": "filter",
  "id": "sw-239",
  "conf": {
    "MODE": {
      "type": "text",
      "value": "permit"
    },
    "COMBINE": {
      "type": "text",
      "value": "and"
    },
    "RULE": {
      "field": {
        "value": "title",
        "type": "text"
      },
      "op": {
        "type": "text",
        "value": "contains"
      },
      "value": {
        "type": "text",
        "terminal": "RULE_1_value"
      }
    }
  }
```

The conf object will be passed to the Python module 'filter_module.py' that is responsible for the execution process. This object consists of the following parameters:

- 1- conf ['MODE']['value']: this parameter shows the mode of the filter module, it could be either 'Permit' or 'Block'. 'Permit' will show the items that match the rules and 'Block' will block the items that match the rules and return the rest.
- 2- conf ['Combine']['value']: this parameter has two values, either 'and' or 'or', this will allow the items that specify all the 'filter' rules or any of the rules.
- 3- conf ['Rule']: this parameter contains the rules that the 'filter' module will use to filter the data. It contains the 'field' parameter that the filtering process will be performed on, and the operations of the 'filter' module. These operations could be: 'contains', 'does not contain', 'is greater than', 'is', 'is less than', 'is after' and 'is before'.

The algorithm of the Python filter module is:

```
def filter_pipe (conf,out_result,input_to,gsorted):
-get the values from the 'conf' object
    mod=conf['MODE']['value']
    combine=conf['COMBINE']['value']
    pipe_rules=conf['RULE']
    module_rules=[]
    rl_com = {"and": all, "or": any}

    for item in input_to:
        for rule in module_rules:
            final_result.append(check_rules(item,rule))

        check= rl_com [combine](final_result)

        if (check and mod=='permit') or (not check and mod=='block'):
            return item

def check_rules(item,rule):
-check the filtering rules such as 'contain, greaterthan, lessthan...'
-return Boolean variable
```

The python module 'filter_module.py' consists of a function called 'filter_pipe', this function will first parse the 'conf' object and save it to variables that will be used for execution. This function will then check the filtering rules for each item and will call another function, 'check_rules', that will check the different rules of filtering we mentioned before. This function will return a list of boolean variables. The built-in function 'all' and 'any' supported by Python will be used, 'any' will return true if any item of the iterable variable is true, and false if it is an empty variable, 'all' return true if all the items of the iterable variables are true. So, the 'combine' variable will be either 'and' or 'or', and it is used to access the corresponding function in the dictionary 'rl_com'. Then it will check the mode of the 'filter' module, if it is 'permit' then it will return the elements that specify the rules, if it is 'block' then it will block the items that specify the rules and return the rest of the items.

3.6.4 Interpreting Control Flow Module

One of the most popular and important modules in Yahoo! Pipes is the 'loop' module. The loop module is responsible for executing a set of statements several times representing the control flow in the workflow. The 'loop' is different from other modules because this module, as we mentioned in chapter 2, allows another module to be embedded inside it, the embedded module will be executed once for each item in the input to the loop module.

Figure 3.6.3.1-1 illustrates that the 'Date Formatter' module is the embedded module in the 'loop' module. Most of the Yahoo! Pipes modules can be embedded in the 'loop' module except the modules in the category 'User input', such as the ('Text Input', 'number input', 'URL input',..) and those in the category 'Operators', such as 'count', 'sort', 'union' [9]. The 'conf' object of the loop module is illustrated in Figure 3.6.4-1, this will be passed to the Python module 'loop_module.py'. This object consists of the following parameters: `conf['mode']['value']`, `conf['embed']['value']`, `conf['emit_part']['value']`, `conf['assign_part']['value']`, `conf['assign_to']['value']`. Each of these parameters holds values that are involved in the execution of the 'loop' module. We will give a brief description of each of the parameters:

- 1- `conf ['mode']['value']`: this parameter shows the mode of the 'loop' module. There are two types of mode, either 'emit' or 'assign'. The 'emit' mode means that the data from executing the embedded module will only be included in the output result, whereas the 'assign' mode means that the data from the original input module will also be included in the output result with the data from the embedded module and it will assign the value to a specific field.
- 2- `conf ['embed']['value']`: this parameter holds the information of the embedded module, such as its id and the 'conf' object of the embedded module.
- 3- `conf ['emit_part']['value']`: this holds the parameter of the 'emit' option we mentioned earlier.
- 4- `conf ['assign_part']['value']`: this holds the parameter of the 'assign' option we mentioned earlier.
- 5- `conf ['assign_to']['value']`: this holds the field that the result from executing the embedded module will be assigned to.

```

"type": "loop",
  "id": "sw-240",
  "conf": {
    "with": {
      "value": "pubDate",
      "type": "text"
    },
    "embed": {
      "value": {
        "type": "dateformat",
        "id": "sw-248",
        "conf": {
          "format": {
            "value": "%d-%m-%Y %H:%M:%S",
            "type": "text"
          },
          "timezone": {
            "value": "",
            "type": "text"
          }
        }
      },
      "type": "text"
    },
    "type": "module"
  },
  "emit_part": {
    "type": "text",
    "value": "all"
  },
  "mode": {
    "type": "text",
    "value": "assign"
  },
  "assign_part": {
    "type": "text",
    "value": "all"
  },
  "assign_to": {
    "value": "pubDate",
    "type": "text"
  }
}

```

Figure 3.6.4-1: The 'conf' object of the loop module

The algorithm of the Python loop module is:

```
def loop (conf,out_result,input_to,gsorted):
    -get the values from the 'conf' object
        -embed_id=conf['embed']['value']['id']
        -mod=conf['mode']['value']
        -emit_part=conf['emit_part']['value']
        -assign_part=conf['assign_part']['value']
        - assign_to=conf['assign_to']['value']text
        -Get the input from the source module
        - execute the embedded module
    If mod=='assign' then:
        For each item in input:
            Item[assign_to]=values from executed embedded module
        return item
    Else if mod=='EMIT' then:
        For each item in input:
            return values from executed embedded module
```

The Python module 'loop_module.py' consists of a function called 'loop'. This will check the parameters of the 'conf' object. These parameters are saved in the different variables. It will check the mode of the 'loop' module: if the mode is 'assign' then it will execute the embedded module and assign the result to the 'assign_to' field from the input data, and if the mod is 'emit' then it will execute the embedded module and these values will be only returned as the output.

During the process of implementation we used some Python code that helped us implement some of the modules. For example, to implement the 'fetch' module we used the universal feed parser, which is an open source written in Python that is used for the process of parsing and downloading syndicated feeds. This can be used as part of any large Python program, it consists of one file 'feedparser.py' that can handle different arguments, such as a local file name, a URL, or a raw string of data in a different format [37][38]. We used and included the 'feedparser.py' module from [37] [38] in our code folder and imported it in the 'fetch_module.py' module and we used it also in the program that counts the number of modules in workflows that we mentioned before. The fetch module takes a URL and parses its data, the URL is then passed to the feed parser to parse its contents that will be used during the implementation. We also used the code from [38] that is responsible for getting the 'modules' and 'wires' JSON objects in 'modules_graph.py', but we changed it according to our needs.

3.7 Summary

This chapter introduced the general overview of the scientific workflow interpreter program. It also described the topological sorting algorithm and described how we decided upon which modules to implement and the factors that affected our decision. We discussed the steps of implementing our project, we illustrated the advantages of the Python language, what data structures it provides and the different libraries it includes. We described how a single module was interpreted and how the JSON data was important in implementing each of the modules. The 'loop' module which represents the control flow in the workflow was described also. As a consequence we found that choosing Python for the implementation was successful, as Python provided different data structures and libraries.

4. Scientific Workflows for Testing and Evaluation

In this chapter we will explain how we tested our system by creating different workflows in Yahoo! Pipes and running them locally using our program. Before creating the workflows we first needed URLs of scientific data. We found on the internet a URL that provides data about earthquakes around the world. Another useful URL we used is the HMM library and genome assignment server, which provides data about genomes and proteins of different species, Dr. Julian Gough from the department of computer science kindly provided us with this link. By using these links we created several workflows using the earthquake data and using the genome data.

We designed a brief use case of these data for each workflow where we found that each of these workflows will be useful for scientists in each of the fields. We will explain in detail each of these workflows.

4.1 Scientific workflow of earthquake data example1

The URL we used to get our testing data is 'http://earthquake.usgs.gov/'. This site provides various information about earthquakes around the world. It provides the area where the earthquake happened, the 'magnitude' of the earthquake, which represents the earthquake size, the 'depth' of the earthquake, the date and time that the earthquake happened and other information. We designed a workflow that will ask the user to enter the minimum and maximum earthquake depth, the workflow will then find the average and display all the earthquakes with a depth greater than the average, this workflow is illustrated in Figure4.1-1.

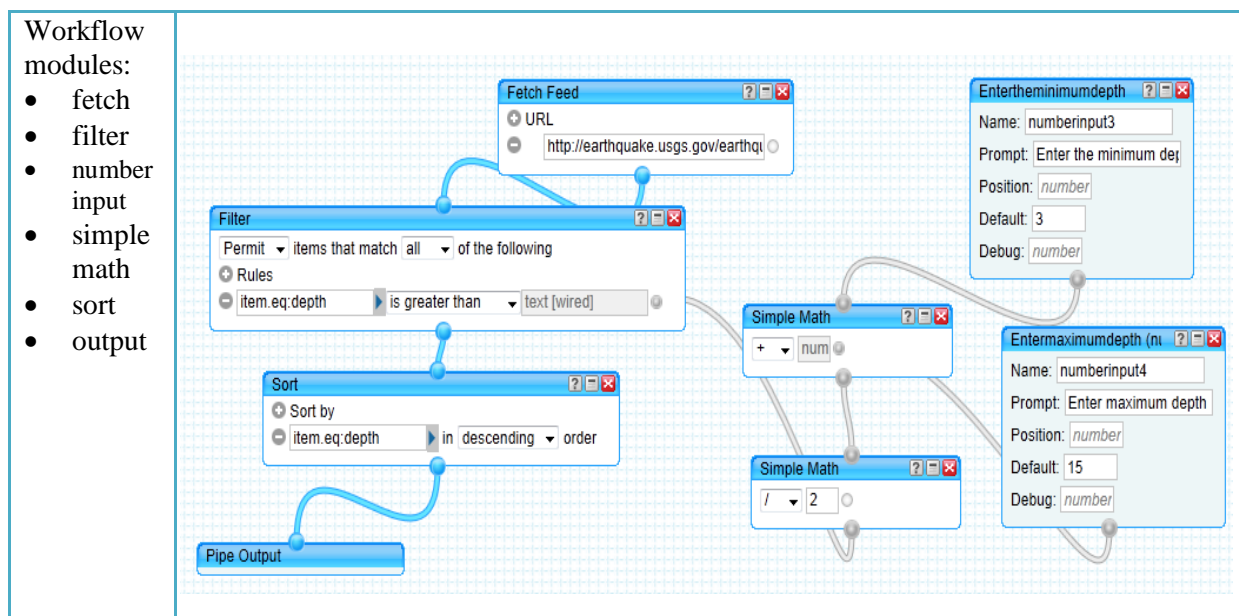


Figure4.1-1 Workflow of earthquake data example1

The modules used for executing this workflow are shown in Figure4.1-1. These are as follows:

- 1- 'fetch' module: this module is used to fetch the data from the URL.
- 2- 'numberinput' modules: these modules are used to ask the user to enter the minimum and maximum depth.
- 3- 'simplemath' module: these are two 'simplemath', one for addition and the other is used to calculate the average of the minimum and maximum depth by using the 'division' operation.
- 4- 'filter' module: is used to get only the places that have a depth greater than the average
- 5- 'sort' module :will sort the results according to the depth in descending order.
- 6- 'output' module: print the final output of the workflow.

S.No.	Data of the workflow
1	{'area': '5.8 - SOUTH OF THE FIJI ISLANDS'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c00057yj/'} {'description': 'Lat/Lon: -23.8997/179.098Depth: 538.2'} {'Date': 'Wed, 03 Aug 2011 19:32:58 +0000'}
2	{'area': '6.7 - SOUTH OF THE FIJI ISLANDS'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c00055wy/'} {'description': 'Lat/Lon: -23.6508/179.822Depth: 521.7'} {'Date': 'Fri, 29 Jul 2011 08:54:35 +0000'}
3	{'area': '6.2 - FIJI REGION'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005glw/'} {'description': 'Lat/Lon: -16.5256/-176.905Depth: 413.2'} {'Date': 'Fri, 19 Aug 2011 04:54:35 +0000'}
4	{'area': '5.6 - SANTA CRUZ ISLANDS REGION'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005a1t/'} {'description': 'Lat/Lon: -11.7947/168.385Depth: 411.9'} {'Date': 'Sun, 07 Aug 2011 06:22:35 UTC'}
5	{'area': '5.6 - FIJI REGION'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/2011pmag/'} {'description': 'Lat/Lon: -21.704/-177.104Depth: 243'} {'Date': 'Fri, 12 Aug 2011 07:28:28 +0000'}
6	{'area': '5.6 - EASTERN NEW GUINEA REG, PAPUA NEW GUINEA'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005f5p/'} {'description': 'Lat/Lon: -5.5753/147.124Depth: 190.8'} {'Date': 'Wed, 17 Aug 2011 14:18:29 +0000'}
7	{'area': '5.7 - ECUADOR'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005eg3/'} {'description': 'Lat/Lon: -1.8001/-76.9919Depth: 166.8'} {'Date': 'Mon, 15 Aug 2011 03:07:20 +0000'}
8	{'area': '7 - NORTHERN PERU'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005j3l/'} {'description': 'Lat/Lon: -7.6441/-74.5063Depth: 145.1'} {'Date': 'Wed, 24 Aug 2011 18:33:27 +0000'}
9	{'area': '5.6 - SOUTH SANDWICH ISLANDS REGION'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005hn9/'} {'description': 'Lat/Lon: -56.3911/-27.5397Depth: 127.7'} {'Date': 'Sun, 21 Aug 2011 13:01:05 +0000'}
10	{'area': '5.8 - VANUATU'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c0005jib/'} {'description': 'Lat/Lon: -13.5994/166.95Depth: 124.5'} {'Date': 'Thu, 25 Aug 2011 10:56:51 +0000'}
11	{'area': '5.6 - GUAM REGION'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/global/shake/c00056m4/'} {'description': 'Lat/Lon: 12.8559/143.22Depth: 120.1'} {'Date': 'Sat, 30 Jul 2011 20:52:59 +0000'}
12	{'area': '5.14 - 43.0 miles WSW of Talkeetna'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10281084/'} {'description': 'Lat/Lon: 62.0485/-151.303Depth: 86.5283'} {'Date': 'Mon, 01 Aug 2011 23:14:42 +0000'}

Table4.1-1: The output after running the workflow of earthquake data example1 locally

We used the id of this workflow in order to run it using our program as we explained in the previous chapter in Figure3.6.2-3. In this figure we showed that the output is printed to the screen, we saved the output to a text file and demonstrated it in a table format to aid clearer understanding. Table4.1-1 shows the result of running the workflow locally using the program that we developed; we showed part of the output because it is a long list of data. These are the earthquake data on different areas in the world and they have a 'depth' value greater than the average depth calculated in the workflow, depending on the values entered by the user. This output is the same that the user would get if he ran the workflow using Yahoo! Pipes.

4.2 Scientific workflow of earthquake data example2

The following workflow also used the earthquake data but with a different case. Here we used two URLs, one the user enters which represents the earthquakes that happened 30 days ago, and another that represents data for earthquakes that happened 7 days ago. This workflow will find earthquakes that happened in the same area more than once during a period of 30 days and of 7 days and have a depth greater than 33km. Figure4.2-1 shows the workflow and the modules used in this workflow.

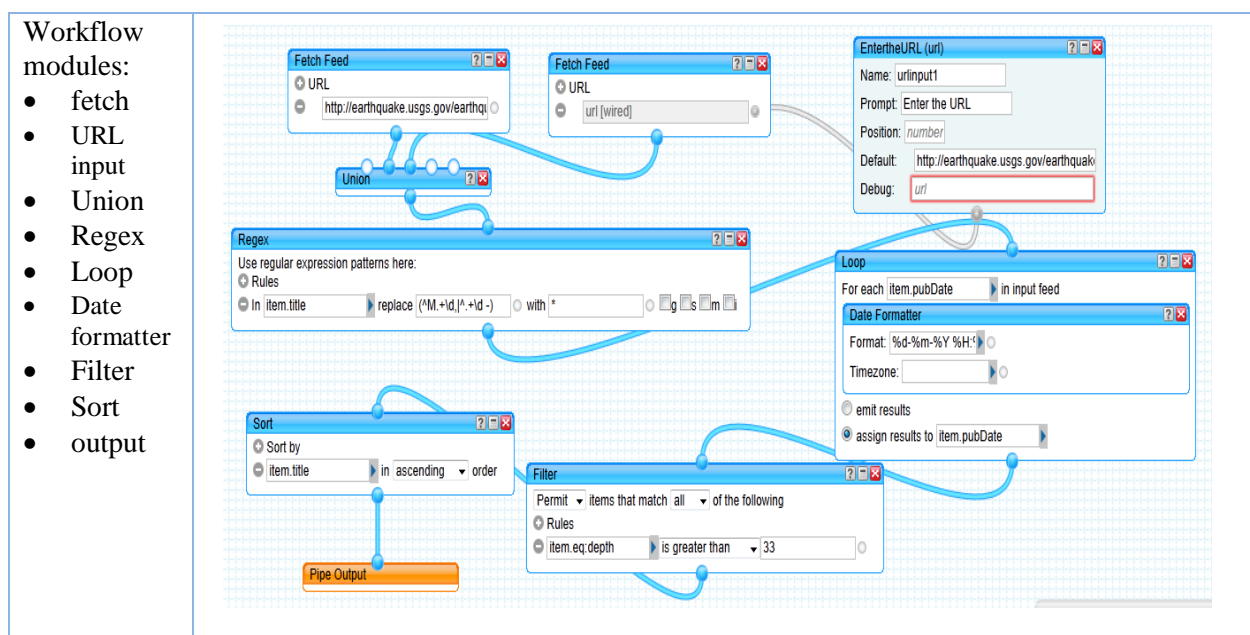


Figure4.2-1 Workflow of earthquake data example2

The modules used in this workflow are shown in Figure4.2-1. These are as follows:

- 1- 'fetch' module: here 2 'fetch' modules are used, one will fetch the data for earthquakes that happened in the 30 days and the other will give data for earthquakes that happened in the 7 days.
- 2- 'urlinput' module: will ask the user to enter a URL that will be used in the 'fetch' module.
- 3- 'union' module: this module will combine the data from the two URLs provided.
- 4- 'regex' module: The data items of each earthquake starts with the magnitude of the earthquake then the area name of the earthquake, since we need only the name we used the 'regex' module to remove the magnitude and keep the area name only.
- 5- 'loop' module: this module is used to loop through each item and execute the 'date formatter' module.
- 6- 'date formatter': this module is used to change the date format of each item according to the user's need
- 7- 'filter' module: this module is used to show only earthquakes with a depth greater than 33km, since by default shallow earthquakes have a depth of 33km (this information is from the earthquake site).
- 8- 'sort' module: this module will sort the data by the 'title' field.
- 9- 'output' module: print the final output of the workflow.

S.No.	Data of the workflow
1	{'area': '* 149.2 miles ENE of Sand Point'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10308407/'} {'description': 'Lat/Lon: 56.0369/-156.867Depth: 34.8667'} {'Date': '08-09-2011 00:58:32'}
2	{'area': '* 171.5 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10304023/'} {'description': 'Lat/Lon: 51.974/-172.634Depth: 52.2427'} {'Date': '31-08-2011 19:17:32'}
3	{'area': '* 20.7 miles NNW of Homer'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10316421/'} {'description': 'Lat/Lon: 59.9021/-151.823Depth: 53.7099'} {'Date': '15-09-2011 00:46:10'}
4	{'area': '* 212.2 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10313361/'} {'description': 'Lat/Lon: 52.1044/-171.682Depth: 46.2937'} {'Date': '13-09-2011 00:10:58'}
5	{'area': '* 212.6 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10302727/'} {'description': 'Lat/Lon: 51.8082/-171.676Depth: 44.3037'} {'Date': '27-08-2011 02:06:28'}
6	{'area': '* 213.4 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10310518/'} {'description': 'Lat/Lon: 52.1909/-171.66Depth: 43.2495'} {'Date': '03-09-2011 01:50:32'}
7	{'area': '* 219.6 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10308796/'} {'description': 'Lat/Lon: 52.0533/-171.504Depth: 41.706'} {'Date': '08-09-2011 20:34:11'}
8	{'area': '* 221.4 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10310823/'} {'description': 'Lat/Lon: 52.0171/-171.462Depth: 54.4995'} {'Date': '09-09-2011 19:31:39'}
9	{'area': '* 224.3 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10308446/'} {'description': 'Lat/Lon: 51.6523/-171.421Depth: 43.9654'} {'Date': '08-09-2011 20:23:39'}
10	{'area': '* 224.6 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10312101/'} {'description': 'Lat/Lon: 51.8911/-171.39Depth: 40'} {'Date': '12-09-2011 23:24:15'}
11	{'area': '* 224.7 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10308471/'} {'description': 'Lat/Lon: 51.7891/-171.393Depth: 42.3628'} {'Date': '08-09-2011 20:26:18'}
12	{'area': '* 225.7 miles E of Adak'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10310969/'} {'description': 'Lat/Lon: 51.8045/-171.368Depth: 39.8986'} {'Date': '09-09-2011 19:38:20'}
13	{'area': '* 36.1 miles SE of Seldovia'} {'link': 'http://earthquake.usgs.gov/eqcenter/shakemap/ak/shake/10301372/'} {'description': 'Lat/Lon: 59.0231/-151.084Depth: 40.9046'} {'Date': '25-08-2011 23:53:14'}

Table 4.2-1: The output after running the workflow of earthquake data example2 locally

Table4.2-1 shows the result of running the workflow locally using our program. These are the earthquake data on different areas around the world. The original data first shows the earthquake magnitude then the area name, for example: ‘2.46 - 149.2 miles ENE of Sand Point’, where ‘2.46’ is the magnitude, we see here the area name appears only without the magnitude value, and the date format is the format the user specified in the ‘date formatter’ module. Also this data is for earthquakes that have a depth greater than 33km and also shows areas where earthquakes happened more than once, such as in the ‘Adak’ area. This output is the same as the output the user would get if he ran the workflow using Yahoo! Pipes.

4.3 Scientific workflow of genome data example1

The other URLs we used were the genome and proteins data (http://supfam.cs.bris.ac.uk/SUPERFAMILY/web_services.html) and (<http://www.ensembl.org/index.html>). We designed a workflow illustrated in Figure4.3-1 that gives a list of genomes available in this URL. This URL provides a list of hundreds of genomes. We designed this workflow such that the scientists have only the genomes that belong to a specific category.

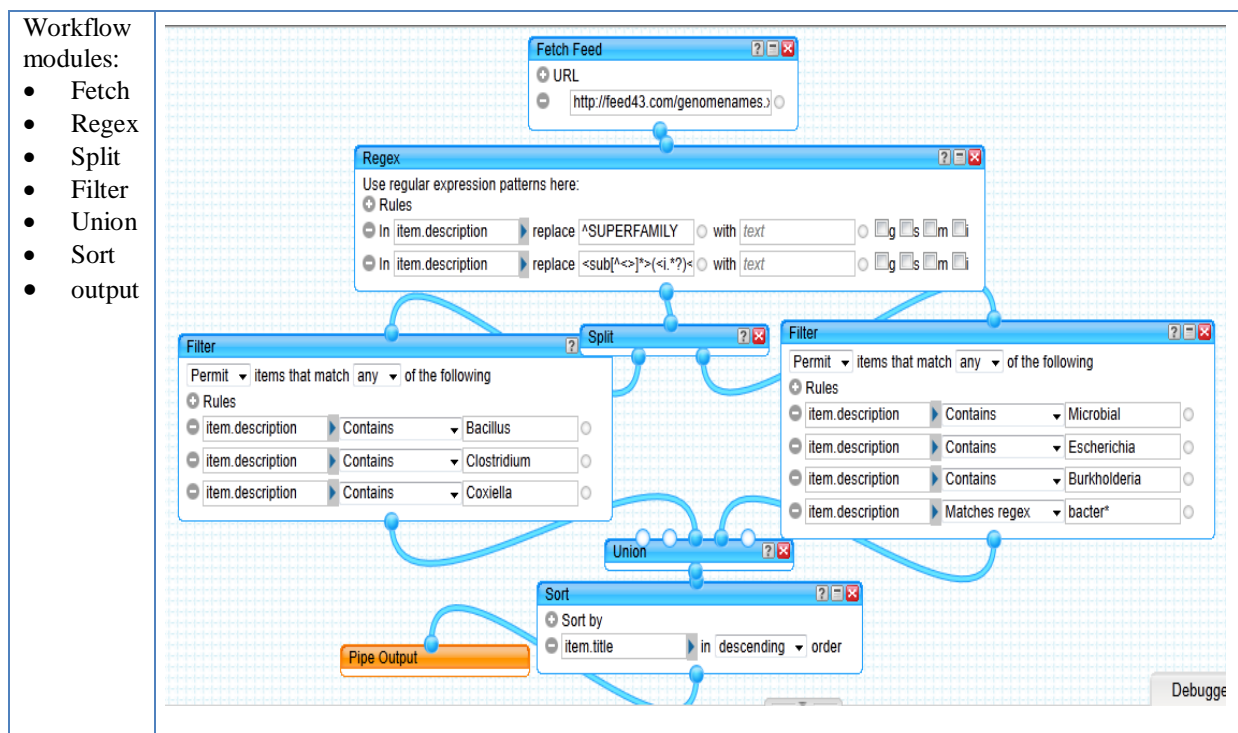


Figure4.3-1Workflow of genomes data example1

The modules used in this workflow are shown in Figure4.3-1. These are as follows:

- 1- ‘fetch’ module: this module will fetch the data from the genome URL.
- 2- ‘regex’ module: this module is to remove the word ‘SUPERFAMILY’ from the description item since it appears before every genome name and also removes the unwanted html tags from the description.
- 3- ‘split’ module: this module is used to split the data to two ‘filter’ modules.
- 4- ‘filter’ module: this module is used to allow the scientists to retrieve only specific genome that belong to a specific category, like ‘microbolia’, ‘bacter’ and ‘Bacillus’.
- 5- ‘union’ module: this module is used to combine the results from the filter modules.
- 6- ‘sort’ module: this module is used to sort the genome data by name.
- 7- ‘output’ module: print the final output of the workflow.

S.No.	Data of the workflow
1	{'id': 'x5'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Bacillus cereus E33L genome assignments'}
2	{'id': 'lp'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Helicobacter pylori SJM180 genome assignments'}
3	{'id': 'lc'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Bacillus licheniformis ATCC 14580 genome assignments'}
4	{'id': 'l7'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Burkholderia sp. CCGE1001 genome assignments'}
5	{'id': 'kx'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Escherichia coli SE11 genome assignments'}
6	{'id': 'kv'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Lactobacillus reuteri JCM 1112 genome assignments'}
7	{'id': 'kt'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Escherichia coli 55989 genome assignments'}
8	{'id': 'ks'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Burkholderia cenocepacia J2315 genome assignments'}
9	{'id': 'kp'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Bacillus cereus B4264 genome assignments'}
10	{'id': 'ki'} {'link': 'http://supfam.org/SUPERFAMILY/cgi-bin/das'} {'description': ' Acidithiobacillus ferrooxidans ATCC 53993 genome assignments'}

Table 4.3-1: The output after running the workflow of genome data example1 locally

Table4.3-1 shows the result of running the workflow locally using our program. These are the genome data the user wants after removing the ‘SUPERFAMILY’ word and the html tags according to the categories the user specified.

4.4 Scientific workflow of genome data example2

The workflow illustrated in Figure4.4-1 is using two URLs of different species: the squirrel and the opossum. Each URL gives the proteins of the genes in each species and the number of genes. We designed the workflow so that the scientist can find the proteins that are shared between these two species.

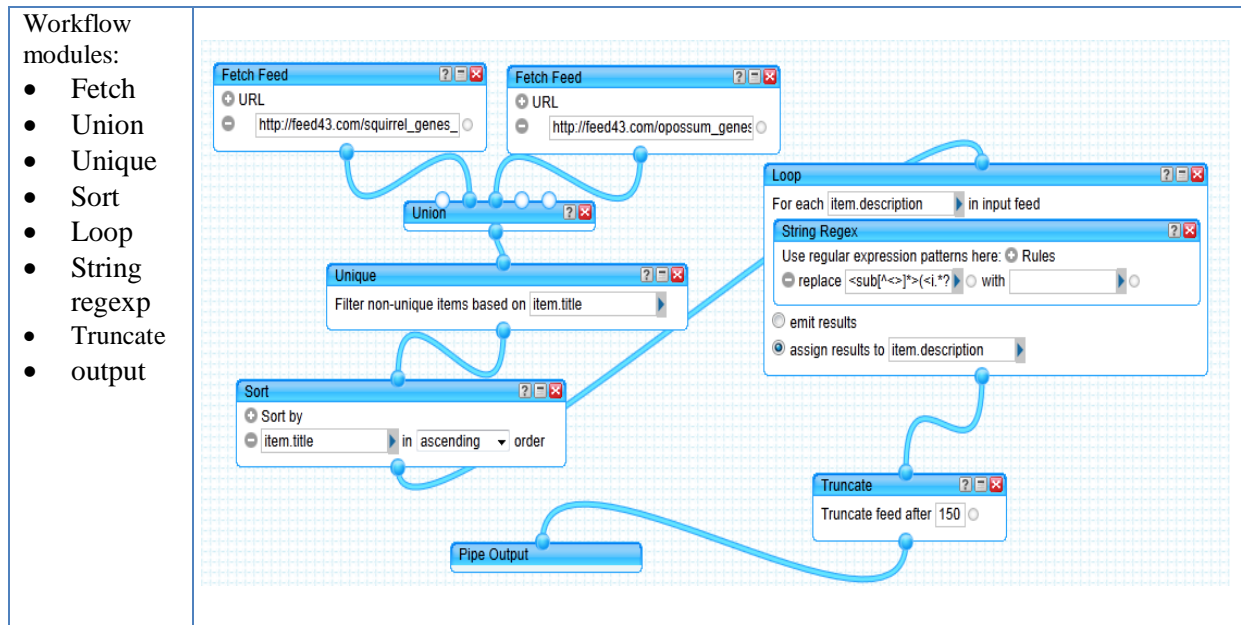


Figure4.4-1 Workflow of genome data example2

The modules used in this workflow are shown in Figure4.4-1. These are as follows:

- 1- 'fetch' module: here two 'fetch' modules are used to retrieve the data of the two species: squirrel and opossum.
- 2- 'union' module: this module is used to combine the data of proteins from the two species.
- 3- 'unique' module: this module is used to find only the proteins that are shared between the squirrel and the opossum.
- 4- 'sort' module: this module is used to sort the protein data by name.
- 5- 'loop' module: this module is used to loop through each item and execute the 'string regex' module.
- 6- 'string regex': this module is used to remove the html tags from the description field.
- 7- 'truncate' module: this module is used to truncate the result to 150 items, because the link has hundreds of items data and the scientist only needs to see 150 proteins that are shared between the species.
- 8- 'output' module: print the final output of the workflow.

S.No.	Data of the workflow
1	{'id': 'IPR000008'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000008'} {'description': 'C2 calcium-dependent membrane targeting,gene number:87'}
2	{'id': 'IPR000047'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000047'} {'description': 'Helix-turn-helix motif, lambda-like repressor,gene number:62'}
3	{'id': 'IPR000048'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000048'} {'description': 'IQ calmodulin-binding region,gene number:66'}
4	{'id': 'IPR000068'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000068'} {'description': 'GPCR, family 3, extracellular calcium-sensing receptor-related\ngene number:71'}
5	{'id': 'IPR000104'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000104'} {'description': 'Antifreeze protein, type I,gene number:51'}
6	{'id': 'IPR000108'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000108'} {'description': 'Neutrophil cytosol factor 2,gene number:39'}
7	{'id': 'IPR000152'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000152'} {'description': 'Aspartic acid and asparagine hydroxylation site,gene number:73'}
8	{'id': 'IPR000169'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000169'} {'description': 'Peptidase, cysteine peptidase active site\ngene number:71'}
9	{'id': 'IPR000195'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000195'} {'description': 'RabGAP/TBC,gene number:34'}
10	{'id': 'IPR000198'} {'link': 'http://www.ebi.ac.uk/interpro/IEntry?ac=IPR000198'} {'description': 'RhoGAP,gene number:51'}

Table 4.4-1: The output after running the workflow of genome data example2 locally

Table4.4-1 shows the result of running the workflow locally using our program. These are the proteins shared between two species: the squirrel and the opossum. This output is the same as the output the user would get if he ran the workflow using Yahoo! Pipes.

Each of these workflows modules will be emulated and executed locally using our program according to the algorithms we explained in the previous chapter, and the final output of the workflow will be printed to screen. As we referred to parsing the arguments from the command line in the previous chapter, the user can use the '-s' option to save the output of the workflow as a text file, and later he can use the file according to his needs.

Table4.4-2 shows the runtime in seconds for the workflows that we mentioned above. Python has the 'time' module that provides different time-related functions, it used it to calculate the time needed for the workflows to be executed. The average execution time of workflows in Yahoo! Pipes is 3 seconds [9], and as we can see from the table we have a reasonable execution time for the workflows.

workflow	Execution time (in seconds)
workflow of earthquake data example1	2.34
workflow of earthquake data example2	2.67
workflow of genomes data example1	2.89
workflow of genomes data example2	2.73

Table4.4-2: Execution time for the testing workflows

Figure 4.4-2 shows the execution time of the workflows we used for testing that we mentioned in the previous sections. This figure gives the time in the x- axis and the number of records each workflow consists of in the y-axis. We can see that time increases as the number of records in each workflow increases.

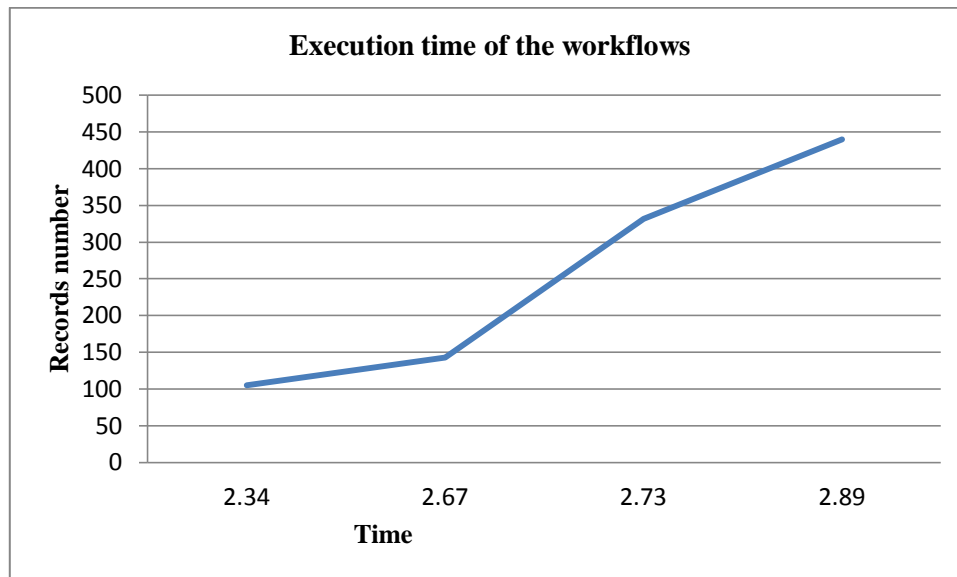


Figure 4.4-2: The execution time and number of records of the Workflows

4.5 Unit testing and workflow output

Yahoo! Pipes provides a facility for saving the output of the workflow in JSON format, as illustrated in Figure4.5-1, where we can see an option of 'Get as JSON'. In our program we saved the output of the workflows in JSON format and compared our output with the Yahoo! Pipes output to ensure that we got the same results, because the output consists of many records that can go into the hundreds.

Python provides unit testing to test any code the users write. We used unit testing to test the output produced by the 'output.py' module. Python provides the 'unittest' module that contains many classes and methods used for testing, such as 'assertEqual', 'assertNotEqual', 'assertTrue' and 'assertFalse' and many others. We imported the 'unittest' module and used the 'assertEqual' method to check our results; these will test if two values are equal or not, and if they are not equal the test will fail and this method will produce a report of the testing results.

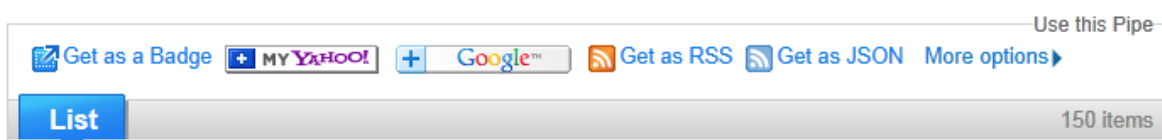


Figure4.5-1: The Yahoo! Pipes options for saving the output in a different format

The following are the steps for unit testing:

- 1- Get the JSON data of the workflow from Yahoo! Pipes.
- 2- Save the output of the workflow as JSON format after running it using our program.
- 3- Run the 'json_test.py' module that we wrote to compare the files from step 1 and 2.

After running the workflow locally, the output is saved as a JSON format file for testing purposes. This is different to the option the user can use to save the workflow output as a text file. The 'json_test.py' module will call the class 'test_json', which will loop through every JSON file (from our program and from Yahoo! Pipes) and compare them using the 'assertEqual' method. An overview of the testing module is shown in Figure4.5-2.

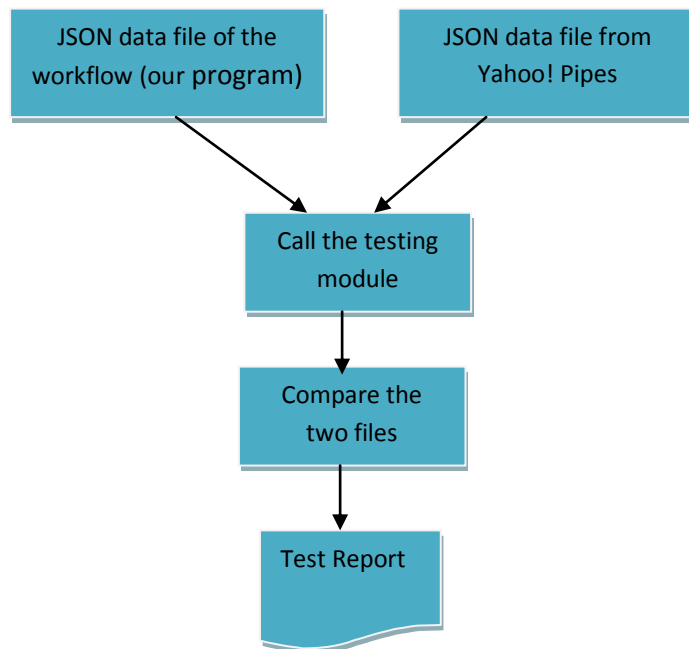


Figure4.5-2: an overview of the testing module

After running the first workflow example of the earthquake data and obtaining the result, we used unit testing to test our output and compare it with the output of Yahoo! Pipes. The testing gave us the following report:

“First differing element:

5.6 - SANTA CRUZ ISLANDS REGION

5.6 - EASTERN NEW GUINEA REG, PAPUA NEW GUINEA”

This workflow sorts the result by depth in descending order. When we checked the record of '5.6 - SANTA CRUZ ISLANDS REGION' it had a depth of '411.9km', so according to our output in Table4.1-1 that is produced after running the workflow using our program, this area is in the fourth record, but in the output of Yahoo! Pipes we found it ordered differently as shown in Figure4.5-3. The other workflow results passed the unit testing.

This shows that the 'sort' module in Yahoo! Pipes is not functioning properly; this is a bug, because Yahoo! Pipes had changed from a V1 engine to a V2 engine in August2011, and in the 'Yahoo! Pipes

Engine 2 Testers' forum there are other Yahoo! Pipes modules that users have reported as having bugs.

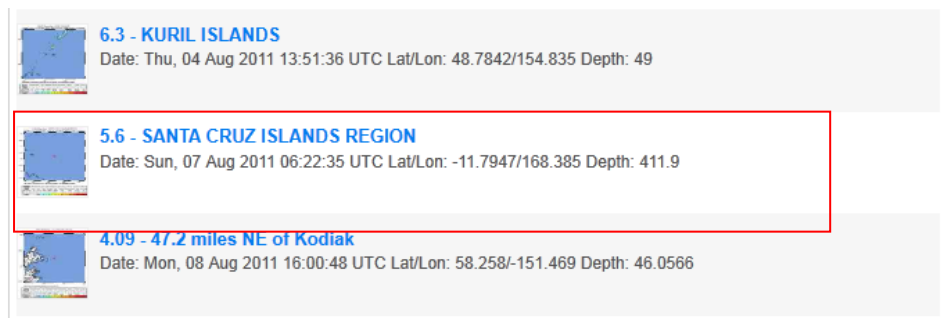


Figure4.5-3: Bug in the sort module of Yahoo! Pipes

There are some notable differences to Yahoo! Pipes V1 engine and V2 engine. During the research review period we created some workflows (pipes) in Yahoo! Pipes for testing, as we described in chapter2. At that time the JSON data consisted of the 'PIPE''live' object that contained all the modules of the workflow. When we started the project implementation, Yahoo! Pipes was in the testing phase of shifting from V1 to V2, and that object had changed to 'PIPE'working', as we mentioned in chapter3. The V1 engine is 4 years old and after the success of Yahoo! Pipes they decided to move to the V2 engine to improve the performance and to provide new features for the users. During the changeover some incompatibilities were found; the V2 engine still receives updates [9].

There are some differences and limitations with V2 engine, these limitations make users think differently when they create their workflows (pipes) not to prevent them from building feature-full workflows (pipes) [9]. The V2 engine limitations are [9]:

- 1- Module inputs: modules can have dynamic inputs, V1 users could add as many inputs as they like to some modules, but now in V2 they have a limit of 10 fields per input parameter.
- 2- 30 second execution timeout: all workflows (pipes) have to run within 30 seconds, if the data URL the user uses takes more than 30 seconds to fetch the data then the user must think of the quality of the URL and think to replace it, as it will affect the workflow performance.
- 3- 1.5 MB feed size: Yahoo! Pipes has a limitation of 1.5 MB on the size of data users can fetch, because Yahoo! Pipes uses YQL that forces this limitation, there are still investigations into configuring this value and changing it.
- 4- Subpipes: another limitation is in the pipe nesting using subpipes, this is limited to a nesting depth of 3. Subpipes are to be used as functions, so if the user nests functions more than a depth of 3, this makes the tracing of the pipe long and complex and this will affect the performance of the pipe.

4.6 Evaluation

In general the aims and objectives of this project were achieved successfully according to time and the resources that were available during the implementation period. The aim of this project was to use the Python language to write a program that parses the JSON data of the workflow created in Yahoo! Pipes, and use this data to emulate the Yahoo! Pipes modules and run the workflow locally. This aim was achieved successfully by writing a program that will first parse the JSON data of the workflow and know the important objects of JSON data for the implementation the 'modules' and 'wires'. This program then sorts the modules, executes each of these modules, and finally prints the output of the workflow. Yahoo! Pipes consists of different categories with different modules. We emulated 24 modules by writing a Python module for each, we found these modules useful and they will help to create a range of useful workflows.

As we mentioned earlier, parsing JSON data can be done using different languages such as JAVA, Perl, Ruby and Java scripts. These languages also have their own features and libraries, but we also mentioned that one of the features that Python has is that it is easy to read and learn. This was an important factor in our project since we are limited with time and for us it was the first time we had used Python; we found it an easy language to learn and with it it was easy to compile and run the code. It also has other features that we mentioned in chapter 3, such as the data structures and libraries it supported and helped in implementation.

We started to create workflows in Yahoo! Pipes using general data such as the University of Bristol news, software products and Yahoo news. Then we ran these workflows locally using our program and found what errors occur when implementing the modules and solved them to make sure that each module was performing properly. But our project is based on the scientific workflows, so we used scientific data that we mentioned in the previous section. The project was successful in using the scientific data to create workflows in Yahoo! Pipes and then run them locally using the developed program. Although Yahoo! Pipes is not a tool specialised in producing scientific workflows like other scientific workflow management systems such as Taverna, Kepler and Triana (explained in chapter 2), we succeeded in using it to create useful scientific workflows using scientific data such as the earthquake and genome data, these workflows ran locally, as explained in the previous section.

Using the topological sorting algorithm for sorting the modules of the workflow was also successful. This sorting algorithm was an important step in our project because it helped us decide which module should be executed before the other. The program can sort the modules of different workflows before executing them. Each of the Python modules emulate the work of the corresponding Yahoo! Pipes module and then print the output of the workflow to the screen. It also allowed the user to save the workflow output as a text file so that they can use the results according to their needs.

Splitting the implementation into steps(as explained in chapter 3) helped in organizing the implementation process, since completing each step was important for implementing and understanding the requirements of the next step. The suggestions for improvements and extensions in the next chapter mean that the program can perform better based on the different information we obtained in the project.

4.7 Summary

In this chapter we presented how we tested our program by running different workflows created in Yahoo! Pipes locally. We achieved our aim of creating the workflows in Yahoo! Pipes and then ran them locally using the program we developed. The modules we implemented allowed us to run different workflows using our program. We created workflows using scientific data URLs and showed the results of running these workflows. We also showed how we used unit testing to test the output we got after running the workflows locally and how we found the bug in the Yahoo! Pipes 'sort' module because of the transfer of Yahoo! Pipes from the V1 engine to the V2 engine. We discussed also how our project was successful in achieving its aims and objectives of parsing the JSON data of any workflow created in Yahoo! Pipes; sorting the modules of the workflows using the topological sort algorithm and then emulating these modules and executing them locally to get the output of the workflow.

5. Conclusions and Future Work

In general the aims and objectives of this project were achieved successfully. This chapter will summarise the conclusions and findings of the project and give suggestions for future extensions and improvements for the project.

5.1 Conclusions

This project aims to emulate the modules of Yahoo! Pipes and run the workflows created with it locally by first parsing the JSON data of the workflow and finding what modules it consists of, and then sorting these modules using the topological sorting algorithm and finally executing each of the modules and obtaining the final output of the workflow. This was done using the Python language, which has many libraries and functions that help with the execution process.

Existing literature (chapter 2) evidenced that different scientific workflow management systems, such as Taverna, Kepler and Triana, all share the same principle of a visual programming language. They all have a graphical user interface and the basic building blocks for creating workflows. Yahoo! Pipes is also a visual programming environment which also consists of the basic building blocks for creating the workflows: modules and wires. The aim of Yahoo! Pipes and other workflow management systems is to make the workflows the users create reusable. For example Taverna is related to myExperiment, where scientists can share their workflows with others and reuse them. This is also the case in Yahoo! Pipes, where the workflows can be published and reused by others, as explained in chapter 2.

The Python language is supportive by the libraries, built-in functions and data structures it consists of. We can parse the JSON data of different workflows using the Python libraries and methods. The built-in functions it provides were helpful when executing and emulating different modules, as we explained in chapter 3. We emulated 24 modules of Yahoo! Pipes. It provides a way of creating user friendly command lines that allow the user to run the workflow locally and save the output of the workflow as a text file. Execution of the modules of any workflow cannot be done without first sorting the modules, so the topological sorting algorithm is important for sorting the modules of any workflow to know the order of module execution. The scientific workflows that we created, as explained in chapter 4 using the scientific URLs, showed that Yahoo! Pipes can be used to produce useful scientific workflows like other scientific workflow management systems (Taverna, Kepler, Triana), although it has some limitations, such as the data size. But as we mentioned earlier, moving to the V2 engine provided many improvements and allows users to make more creative workflows. Executing these scientific workflows locally was done in a reasonable time according to the number of records in each workflow, as explained in chapter 4.

Although the core of the project was implemented, there are still some improvements and extensions that can be done. These improvements will make the application perform better and continuously satisfy the user's needs, as we will explain in the next section.

5.2 Extending and improving the interpreter

As we mentioned earlier we implemented and emulated 24 modules of Yahoo! Pipes, so our program can be extended by implementing the other modules that we did not implement, and these modules will be useful for the users and will support creating different workflows. Another issue is creating a

graphical interface; a good visualization will be beneficial for the usability of the program, making it more user friendly and also adding analysis tools that will help the user to analyse the results of the workflow. This will be helpful for the users as we found in chapter 2 that other scientific workflow management systems has a user interface like Triana, Taverna and Kepler.

During the implementation we used the ‘topological’ sort algorithm for sorting the workflow modules. There are other sorting algorithms for sorting the graphs that can be tested on the workflow and might have different effects on the workflow execution process. Further investigation into some of the modules, such as the ‘regex’ and ‘stringregex’ modules, can be carried on. Python might have different representations to some of the regular expressions used in Yahoo! Pipes.

The program can be tested in different ways. We created workflows in Yahoo! Pipes and then ran them locally using our program, which was our main aim. This can be tested also by scientists from different science fields in the university, they can create the workflows they need in Yahoo! Pipes and run them locally. From the feedback returned from the scientists different improvements can be made. The scientists will also use different scientific data that will help to adapt the program according to their needs, the variation of the scientific data will help in understanding what workflows the program can run and those that it cannot, and also what limitations it might have in the scientific field.

This chapter presented the findings and conclusions of the project that we found during the period of research and implementation. It concluded how scientific workflows systems are important in the field of science and how did we parse the JSON data of the workflow in order to run them locally using Python language. It also presented what extensions can be added to project for future improvements and better performance.

References

- [1] Hey, T., Tansley, S., and Tolle, K. The Fourth paradigm, data-Intensive Scientific discovery, version 1.1, pg.137-145, 2009.
- [2] Wolstencroft, K., Oinn,T., Goble,C., Ferris,J., Wroe,C., Lord,P., Glover,K., and Stevens, R., Panoply of Utilities in Taverna. In First International Conference on e-Science and Grid Computing, pg.156-162, 2005
- [3] Missier,P., Soiland-Reyes,S., Owen,S., Tan, W., Nenadic,A., Dunlop,I., Williams,A., Oinn, T., and Goble, C. Taverna, Reloaded. In Scientific and Statistical Database Management, Vol.6187, pg.471-481, 2010
- [4] Zhang,J., Co-Taverna: A Tool Supporting Collaborative Scientific Workflows. In IEEE International Conference on Services Computing, pg.41-48, 2010.
- [5] Oinn,T., Addis,M., Ferris,J., Marvin,D., Senger, M., Greenwood,M., Carver.T. , Glover.K., R. Pocock,M., Wipat. A. and Li,P., Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics, Vol. 20, pg. 3045–3054, 2004.
- [6] Taylor, I., Triana Generations.In second IEEE International Conference on e-Science and Grid Computing, pg.143, 2006.
- [7] Majithia, S. ,Shields, M. ,Taylor, I. and Wang, I., Triana: A Graphical Web Service Composition and Execution Toolkit. In Proceedings. IEEE International Conference on Web Services, pg.514-521, 2004.
- [8] Churches, D., Gombas, G., Harrison, A., Jason Maassen, Robinson, C., Shields, M., Taylor, I., and Wang, I., Programming scientific and distributed workflow with Triana services, Concurrency and computation: practice and experience,vol.18, issue.10, pg.1021–1037,2006.
- [9] Yahoo! Pipes, <http://pipes.yahoo.com/> (last accessed 03/09/2011).
- [10] Meza,R. and Buchmann, R. A.Real-time Social Networking Profile Information Semantization Using Pipes and FCA. In IEEE International Conference on Automation, Quality and Testing, Robotics, pg.1-5, 2010.
- [11] Held.M. and Blochinger, W., Structured collaborative workflow design. In Future generation computer systems, vol.25, iss:6, 2009.
- [12] Kim,H., Kim,H., Jang,M., Han,S. and Ceong,H. A Comparison of Features of Ajax’s Data Formats for the Medicinal Plants Application. In 2nd International Conference on Information Technology Convergence and Services, pg1-6. 2010.
- [13] Jun,Y., Zhishu1,L. and Yanyan,M. JSON Based Decentralized SSO Security Architecture in E-Commerce. In International Symposium on Electronic Commerce and Security, pg.471-475, 2008.
- [14] C. B. Medeiros , J. Perez-Alcazar, L. Digiampietri, G. Z. Pastorello Jr, A. Santanche, R. S. Torres, E. Madeira, E. Bacarin, Woodss and the web: annotation and reusing scientific workflow, ACM SIGMOD Record,vol.34, issue.3, pg.18-23, 2005.
- [15] Zhao,Y.,Raicu, I. and Foster,I. Scientific Workflow Systems for 21st Century, New Bottle or New Wine?.In IEEE Congress on Services, pg.467-471, 2008.

- [16] DeRoure,D.,Goble,C.,Bhagat,J.,Cruickshank,D.,Goderis,A.,Michaelides,D.and Newman, D., myExperiment: Defining the Social Virtual Research Environment. In IEEE Fourth International Conference on eScience, pg.182-189, 2008.
- [17] Gordon,P., Barker,K. and Sensen,C., Helping Biologists Effectively Build Workflows, without Programming, Data Integration in the Life Sciences, vol. 6254, pg.74-89, 2010
- [18] Mouallem,P., Craw,D., Altintas,I.,Vouk,M., and Yildiz,U. A Fault-Tolerance Architecture for Kepler-Based Distributed Scientific Workflows. In Proceedings of the 22nd international conference on Scientific and statistical database management, vol.6187, pg.452-460, 2010.
- [19] Bowers,S.and Lud"ascher, B., Actor-Oriented Design of Scientific Workflows, Conceptual Modeling – ER 2005, vol.3716, pg.369-384, 2005.
- [20] Liu, Y., Deng, H., Wang, F., Ji, K., The Key Techniques of Scientific Workflow System. in: Information Technology and Applications (IFITA), pg.259-262, 2010.
- [21] Ludascher,B., Altintas,I., Berkley,C., Higgins,D., Jaeger,E., Jones, M., A. Lee, E., Tao, J. and Zhao, Y., Scientific workflow management and the Kepler systemJing, CONCURRENCY AND COMP UTATION: PRACTICE AND EXPERIENCE, Vol.18, issue.10, pg.1039-1065, 2006
- [22] Chase,J.,Gorton,I., Sivaramakrishnan,C., Almquist,J., Wynne,A.Chin,G.,Critchlow, T. Kepler + MeDICi –Service-Oriented ScientificWorkflow Applications. In Congress on Services, pg.275-282, 2009.
- [23] Altintas, I., Berkley, C., Jaeger,E., Jones,M., Ludascher,B., Mock, S. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In Proceedings. 16th International Conference on Scientific and Statistical Database Management, pg.423-424, 2004.
- [24] Loton, T., Working with Yahoo! Pipes No Programming Required, Lotontech Limited, 2008.
- [25] Altintas, I., Birnbaum,A., K. Baldridge, K., Sudholt, W., Miller,M., Amoreira,C., Potier,Y., and Ludaescher,B., A Framework for the Design and Reuse of Grid Workflows, Scientific Applications of Grid Computing, vol.3458, pg.295-299, 2005.
- [26] Chang,S.K., Barnett,M., M.Levialdi,S., Marriott,K., Pfeiffer,J.J., Tanimoto, S.L. The future of visual languages. In Proceedings 1999 IEEE Symposium on Visual Languages, pg.58-61,1999.
- [27] Xiajiong, S., Ge,W., Jun,G., Xinfa,D. A Novel Visual Programming Method Designed for Error Rate Reduction. In International Symposium on Computer Science and Computational Technology, pg.280-283,2008
- [28] Feng-Cong,L., Yi-Nan,Z., Xiao-Lin,Q. Secondary Development of Visual DSP++ Using Python for Debugging Peripherals of DSP. In First International Conference on Pervasive Computing, Signal Processing and Applications, pg.899, 2010
- [29] Rez, F., Granger, B.E. and Hunter, J.D, Python: An Ecosystem for Scientific Computing, In Computing in science & engineering ,vol.13, iss.2 pg.13-21, 2011.
- [30] Python v2.7.2 documentation, <http://docs.python.org/library/urllib.html> (last accessed on 8/8/2011)

- [31] Sudhakar, T.D., Srinivas, K.N. Power System Restoration Based On Kruskal's Algorithm. In Electrical Energy Systems (ICEES), 2011 1st International Conference on, pg.281-287,2011.
- [32] David J. Pearce, Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. In Journal of Experimental Algorithmics (JEA), v10.11, 2006
- [33] <http://developer.yahoo.com/yql/> (last accessed on 19/8/2011)
- [34] Neumann, G., A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation, Logic Programming, v10. 592, pg.349-360, 1992.
- [35] Topological sorting algorithm, <http://www.logarithmic.net/pfh/blog/01208083168> (last accessed in 5/7/2011)
- [36] Chun, Wesley J., Core Python programming. 2nd ed., pg.6-11, 2007.
- [37] Universal feed parser, <http://feedparser.org/docs/introduction.html> (last accessed in 22/8/2011)
- [38] <https://github.com/ggaughan/pipe2py> (last accessed in 23/8/2011)
- [39] Klint, P., Interpretation Techniques. In Software: Practice and Experience, vol.11, issue.9, pg.963-973,1981.

Appendix: Source Code

Pipe_data.py module

```
"""Author:Sheikha Al-yaqoubi
This programme is for reading the json data from a Yahoo!pipes and print
it to the screen
in a format that is easy to be read.
```

```
This programme can be run like this:
python pipe_data.py -id 6b2f11634f243cfc09f656871561f0fb
so the pipe id should be included to run the code
```

```
and in python2.7 you can run it like this:
pipe_data.py -id 6b2f11634f243cfc09f656871561f0fb
and to save the output in a text file then you write:
pipe_data.py -id 6b2f11634f243cfc09f656871561f0fb -s output.txt
```

```
this line if for the programme to run on *nix system: #!/usr/bin/python
as pipe_data.py -id 6b2f11634f243cfc09f656871561f0fb without the need
to include python at the begining
```

```
arguments:
url-- is the workflow URL used to parse the JSON data
```

```
"""
```

```
#!/usr/bin/python
import signal
import time
import pydoc
import sys
import urllib, json, urllib2
import argparse
import timeit
import itertools
import pickle
from timeit import Timer
import modules_graph
import modules_top_sort
from jsonFormat import json_format
from modules.check_excute import check_excute
```

```
#this argument to save the pipe modules in a dictionary
pipe_json={}
ur_lid=None
```

```
def pipe_data(urlid):
    """ function to get the pipe data This the function that opens the URL
    and parse the JSON data of the workflow """
    global pipe_json

    #global ur_lid
    #get the id from the command prompt and make the url by concatenating
    it with the strings

    url='http://pipes.yahoo.com/pipes/pipe.info?_id='+urlid+'&_out=json'
    url_response = urllib.urlopen(url)

    try:
        url_response = urllib2.urlopen(url, timeout=10)
    except urllib2.URLError:
        print 'can not connect to the URL, time out'
```

```

# open the url, read the data as json dat
url_content = url_response.read()
data= json.loads(url_content)
json_data=json.loads(data['PIPE']['working'])
count= 0

print 'PIPE:', '\n'
print '\tid:', data['PIPE']['id'], '\n'
print '\tname:', data['PIPE']['name'], '\n'

pipe_json=json_data

for item in json_data.items():
    for value in item[1]:
        # print the type, id and conf parts
        if value.has_key('type') and value.has_key('conf') and value.has_key('id'):
            print '\t\ttype:', value['type'], '\n'
            print '\t\t\tid:', value['id'], '\n'
            print '\t\t\tconf:', '\n'

def main():
    """ This is the main function that calls 'pipe_data' function to parse
    the JSON data then it calls the 'modules_graph' function that will
    specify the needed JSON data(modules,wires)It will call after that
    the 'topological_sort' function that will sort the graph and
    finally the 'check_excute' function' is called to excute the
    modules of the workflow and print the output

    arguments:
        pipe_struct: the JSON structure of the workflow
        sorted_graph: the sorted modules of the workflow
    """

    global ur_lid
    global pipe_struct
    global sorted_graph

    #read the yahoo pipe id from the screen and pass it to the pipe_data
    function
    # argumets to give help about how to run the code
    parser = argparse.ArgumentParser(description='This programme is for
    emulating the workflows created in Yahoo!Pipes')
    parser.add_argument('-id', dest='id', help='enter the yahoo pipe id to
    run the code')
    parser.add_argument('-s', dest='text_out', help='saving the result to a
    file, here you enter the file name as .txt')

    args=parser.parse_args()

    ur_lid=args.id
    pipe_data(ur_lid)
    pipe_struct =modules_graph.modules_graph(pipe_json)
    sorted_graph=modules_top_sort.topological_sort(pipe_struct['graph'])
    result=check_excute(pipe_struct,sorted_graph)

    # save the result to a text file
    if args.text_out!=None:
        result2=iter(result)
        text_out=open(args.text_out,mode='w')
        for item in result2:
            item2=str(item)

```

```

        text_out.write(item2)
        text_out.write('\n')

    text_out.close()

if __name__=='__main__':
    try:
        import json
        json.loads
    except (ImportError,AttributeError):
        import simplejson as json

main()

```

check_execute.py module:

"""
 This module is to excute the modules function like 'fetch_module'
 it checks each modules id in the sorted graph dictionary and excute it,
 and it passes the output of the previous module to the next module
 at the end it will use the 'output' module to print the output of the
 workflow

arguments:
 pipe--which is the json data from the pipe, it contains the module type,
 id and conf of each module
 sorted_graph-- it is the list of modules sorted using the topological sort
 algorithm
 wires-- it is the wires connecting the modules

```

"""
import sys,os
import timeit
import time
from jsonFormat import json_format
import json
from timeit import Timer
from fetch_module import fetch
from _OUTPUT import _OUTPUT
from truncate_module import truncate
from split_module import split
from union_module import union
from count_module import count
from sort_module import sort
from unique_module import uniq
from urlbuilder_module import urlbuilder
from textinput_module import textinput
from regex_module import regex
from strregex_module import strregex
from numberinput_module import numberinput
from urlinput_module import urlinput
from simplemath_module import simplemath
from rename_module import rename
from strconcat_module import strconcat
from loop_module import loop
from filter_module import filter_pipe
from dateinput_module import dateinput
from dateformat_module import dateformat
from reverse_module import reverse
from substr_module import substr
from strreplace_module import strreplace
import itertools
from datetime import datetime

```

```

result2={}

#save all the functions of the modules in a dictionary to make it easy to
get the function name and excute it
func_dic={'fetch':fetch,'truncate':truncate,'split':split,'union':union,'c
ount':count,'sort':sort,'uniq':uniq,'urlbuilder':urlbuilder,'textinput':te
xtinput,'regex':regex,'strregex':strregex,'numberinput':numberinput,'urlin
put':urlinput,'simplemath':simplemath,'rename':rename,'strconcat':strconca
t,'loop':loop,'filter_pipe':filter_pipe,'dateinput':dateinput,'dateformat'
:dateformat,'reverse':reverse,'substr':substr,'strreplace':strreplace,'_OU
TPUT':_OUTPUT}

index=0
previous_mod=None
outresult={}
#empty dictionary used for the first excutes module since it doesn't have
a previous modules
emp={}
input1={}

#wire_out is to save each output after the module is excuted with wire id
and module source id
wire_out=[]
union_out=[]
url_out=[]
source=None
flag=False
target1=None
excuted=False
excutedu=False
starttime=0
embedflag=False
looptype=''
mod3=''
part=''
sorted_list=[]
flage=False
loopconf=[]
embedid={}
loopid=''

def check_excute(pipe,sorted_graph):
global count
global result2
global index
global outresult
global previous_mod
global source
global flag
global target1
global wire_out
global excuted
global excutedu
global starttime
global looptype
global mod3
global part
global flage
global loopconf
global embedid
starttime=time.clock()
mod=pipe['modules']
wires=pipe['struct_wires']

```

```

graph=pipe['graph']
embed=pipe['embed']

for i in embed:
    embedid=embed[i]['id']
    embedtype=embed[i]['type']
    embedconf=embed[i]['conf']
    embedtarget=embed[i]['id']

#remove modules that has no target from the sorted graph
for node in sorted_graph:

    targetnode=pipe['graph'][node]

    if targetnode==[] and node !='_OUTPUT':

        sorted_graph.remove(node)

#this is for appenending the sorted modules with its type
for k in sorted_graph:

    for o in mod.items():
        for y in o[1]:
            if o[1][y]['id']==k:
                sorted_list.append({'id':k,'type':o[1][y]['type']})
                break
        break

print'\n\n'
print 'the sorted Graph is:'
print '\n'
for t in sorted_list:
    print 'Module id:',t['id'],',', 'Module type:',t['type']
print '\n'

#check loop conf to send it to the embed module
for h in mod:

    if mod[h]['type']=='loop':
        loopconf=mod[h]['conf']
        loopid=mod[h]['id']
        loopconf['id']=loopid

for gsorted in sorted_graph:
    ptype=mod[gsorted]['type']
    pconf=mod[gsorted]['conf']
    pid=mod[gsorted]['id']

    if ptype=='loop':
        looptype='loop'
        mod3=pconf['mode']['value']
        part=pconf['emit_part']['value']

    if ptype=='filter':
        ptype='filter_pipe'

for wire in wires:

    if ((gsorted==wires[wire]['tgt']['moduleid']) and gsorted!='_OUTPUT'):

```

```

source=wires[wire]['src']['moduleid']
flag=True

for wire in wires:
    if wires[wire]['src']['moduleid']==gsorted:
        target2=wires[wire]['tgt']['moduleid']
        source2=wires[wire]['src']['moduleid']
        gtype=mod[gsorted]['type']
        gconf=mod[gsorted]['conf']
        tgtid=wires[wire]['tgt']['id']

        if gtype=='filter':
            gtype='filter_pipe'

for tgt in wire_out:
    if tgt['tgt']==gsorted and tgt['source']!=gsorted and
        tgt['source']==source :
        if (gtype!='union' and gtype!='urlbuilder' and gtype!='truncate'
            and gtype!='simplemath' and gtype!='rename'
            and gtype!='strconcat' and gtype!='loop' and
            gtype!='filter_pipe' and gtype!='strregex' and gtype!='regex'
            and gtype!='sort' and gtype!='dateformat' and gtype!='reverse'
            and gtype!='substr' and gtype!='strreplace'):

            out=tgt['out']
            out=list(out)
            input_to=tgt['input']
            input_to=list(input_to)
            res2=func_dic[gtype](gconf,out,input_to,gsorted)
            res2=list(res2)
            wire_out.append({'source':gsorted,'out':res2,'tgt':target2,
                'ptype':gtype,'input':input_to,'targetid':tgtid,
                'embedflag':False})

    elif (gtype=='union' and excuted==False):

        for tgt2 in wire_out:
            if tgt2['tgt']==gsorted :
                out2=tgt2['out']
                out2=list(out2)
                union_out.append({'id_src':gsorted,
                    'source':'union','out':out2,'tgt':target2})

elif (gtype=='urlbuilder' or gtype=='truncate' or gtype=='simplemath' or
    gtype=='rename' or gtype=='strconcat' or gtype=='loop' or
    gtype=='filter_pipe' or gtype=='strregex' or gtype=='regex' or
    gtype=='sort' or gtype=='dateformat' or gtype=='reverse' or
    gtype=='substr' or gtype=='strreplace') and
    excutedu==False:
    print '\n'
    print '*****'

    res4=func_dic[gtype](gconf,wire_out,emp,gsorted)
    res4=list(res4)
    wire_out.append({'source':gsorted,'out':res4,'tgt':target2,
        'ptype':gtype,'input':emp,'targetid':tgtid,'embedflag':False})
    excutedu=True
    if gtype=='loop':
        mod3=''
        part=''

    if (gtype=='union' and excuted==False):

```

```

        res3=func_dic[gtype](gconf,union_out,emp,gsorted)
        res3=list(res3)
        excuted=True
        wire_out.append({'source':gsorted,'out':res3,'tgt':target2,
        'ptype':gtype,'input':emp,'targetid':tgtid,'embedflag':False})

    if embed !={}:
        if gsorted==embedid:
            res5=func_dic[embedtype](embedconf,wire_out,loopconf,gsorted)
            res5=list(res5)
            embedtarget1=graph[gsorted]
            embedtarget1=embedtarget1[0]
            wire_out.append({'source':gsorted,'out':res5,'tgt':embedtarget1,
            'ptype':embedtype,'input':emp,'targetid':embedtarget,
            'embedflag':True})
            excutedu=True
            flage=True

excuted=False
excutedu=False

if flag!=True and gsorted!='_OUTPUT':
    if gsorted !=embedid:
        result=func_dic[ptype](pconf,emp,input1,gsorted)
        result=list(result)

    else:
        result=func_dic[ptype](pconf,wire_out,loopconf,gsorted)
        result=list(result)
        wire_out.append({'source':gsorted,'out':result,'tgt':loopid,
        'ptype':ptype,'input':result,'targetid':tgtid1,'embedflag':True,})

        flag=False

for w in wires:
    if wires[w]['src']['moduleid']== gsorted:
        target1=wires[w]['tgt']['moduleid']
        tgtid1=wires[w]['tgt']['id']
        targettype=mod[target1]['type']

        if targettype=='loop' and gsorted==embedid:
            wire_out.append({'source':gsorted,'out':result,'tgt':target1,
            'ptype':ptype,'input':result,'targetid':tgtid1,
            'embedflag':True,})
        else:
            wire_out.append({'source':gsorted,'out':result,'tgt':target1,
            'ptype':ptype,'input':result,'targetid':tgtid1,
            'embedflag':False,})

flag=False

# print the output and save it as JSON format
for out_res in wire_out:
    outtype=out_res['ptype']
    outtarget=out_res['tgt']

if outtarget =='_OUTPUT' and outtype=='union' :
    final_result=out_res['out']

```



```

result_txt=func_dic['_OUTPUT'](final_result,looptype,mod3,part)
with open('file_out.json',mode='w') as json_file:
    json.dump(result_txt,json_file,default=json_format)

elif outtarget == '_OUTPUT':
    final_result=out_res['out']
    result_txt=func_dic['_OUTPUT'](final_result,looptype,mod3,part)
    with open('file_out.json',mode='w') as json_file:
        json.dump(result_txt,json_file, default=json_format)

endtime=time.clock()
timet=endtime-starttime
print 'Excution time:'
print timet
return result_txt

```

Text_input.py module

```

""" This function is for textinput module:
This function is to get the text input from the user

arguments used:
conf--which is the conf object of the yahoo pipe:
    message=conf['name']['value']: message displayed for the user
    prompt_value=conf['prompt']['value']: prompt value
    default_value=conf['default']['value']:default value

out_result--the result from the previous module
input_to--not used here

returns:
    text-- the value entered by the user, if the user didn't type a value,
           then the default value is taken
here we used the 'raw_input' built in function in python to print a
message for the user and ask to enter a value
"""

```

```

def textinput(conf,out_result,input_to,gsorted):

    text=None
    message=conf['name']['value']
    prompt_value=conf['prompt']['value']
    debug=conf['debug']['value']
    default_value=conf['default']['value']
    print '\n\n\n'

    #if 'prompt' value is not nulll, this means that a message is printed
    asking for the user to enter a value

    if prompt_value != '':

        text=raw_input(prompt_value.encode('utf-8') + (" (default=%s) " %
            default_value.encode('utf-8'))

        if text=='':
            text=default_value
    else:
        text=raw_input((" (default=%s) " % default_value.encode('utf-8')))
        if text=='':
            text=default_value

    print '\n\n\n'
    yield text

```