VisFlow: A Visual Language Based Workflow Querying System

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Xin Mou

Major Professor: Hasan M. Jamil, Ph.D.

Committee Members: Clinton L. Jeffery, Ph.D.; Xiaogang Ma, Ph.D.; Robert Rinker, Ph.D.;

Lucas Sheneman, Ph.D.

Department Administrator: Terry Soule, Ph.D.

May 2019

Authorization to Submit Dissertation

This Dissertation of Xin Mou, submitted for the degree of Doctor of Philosophy with a Major in Computer Science and titled "VisFlow: A Visual Language Based Workflow Querying System," has been reviewed in final form. Permission, as indicated by the signatures and dates below is now granted to submit final copies for the College of Graduate Studies for approval.

Major Professor:	Hasan M. Jamil, Ph.D.	Date
Committee Members:	Clinton L. Jeffery, Ph.D.	Date
	Xiaogang Ma, Ph.D.	Date
	Robert Rinker, Ph.D.	Date
	Lucas Sheneman, Ph.D.	Date
Department Administrator:	Terry Soule, Ph.D.	Date

Abstract

In recent years, availability of massive amounts of data, the global nature of collaborative research, the emergence of web services platforms, and interconnected resources on the Internet lead to the heavy uses of data integration, data exploration and data analysis tools for domain researchers in their everyday work. In this new model, modern scientific applications rely heavily upon accessing, integrating and analyzing online data utilizing computational tools from numerous sources in highly ad hoc and dynamic ways. Meanwhile, open online data repositories are often heterogeneous and outside users' control. What's more, their services are offered as is - without any coordination with the site.

Domain researchers (domain experts) have extensive domain knowledge but not necessarily skills and knowledge on web protocols, databases, and formal query languages. Although new technologies such as microservices, semantic web, and cloud computing have helped, a system in which domain researchers are comfortable designing new applications and developing them without technical support from computer scientists is yet to be realized. It turns out that many available artifacts are not suitable for such a computing paradigm without substantial adaptation or customization, and many well-regarded systems insist on substantial manual intervention. A friendly graphical interface and a web services based system that is clunky but covers plenty of resources has not been created yet. The question that still needs to be answered is if users can state the query needs in an easy way that computers understand it and execute it without further help from experts. When queries involve a large set of complex, heterogeneous and distributed sciences databases and tools, the answer is truly nontrivial.

In this dissertation, we present a visual querying language, named VisFlow, to frame arbitrary workflow queries over a distributed set of online databases. We discuss the query execution model by illustrating how it is translated to a script querying language called BioFlow and how BioFlow is mapped to executable code. We also discuss its query mapping strategy that aims to improve efficiency using parallelization whenever possible. We highlight its novelty by evaluating VisFlow workflow composition and execution, showing workflow composition and comparing the time of execution in a geoscience pipeline. Five examples are present to show that VisFlow is capable of seamlessly dealing with traditional, web service based and semantic web databases and online analytics. Finally, we conclude our discussion with VisFlow's planned future research.

Acknowledgements

First and foremost, I would like to acknowledge the contribution of my advisor, Dr. Hasan M. Jamil, in making this work possible. I would like to express my grateful and sincere appreciation to him for his guidance, advice, understanding and patience. I have learned numerous things from him, from paper writing, presentation skills and approaches of conducting researches.

I would like to express my gratitude to my committee members: Dr. Clint Jeffery, Dr. Xiaogang Ma, Dr. Robert Rinker, and Dr. Lucas Sheneman for their valuable discussions and accessibility. Their enthusiasm for science is extremely contagious and has deeply influenced me.

Last but not least, I would like to thank Paul Fisher, Murilo Borges, Davy Cats and Dr. Carlos R. Rivero inspiration in the discussions and validation of the workflows.

Dedication

To my beloved family,

for making all of this possible.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Workflow Technology and Workflow Management Systems

Workflow technology is a type of software product that offers a generic mechanism to integrate diverse types of available libraries and resources to improve the design of information systems. Workflow technology originated from office automation in the business world, which includes repeatable processes of business activities that transform documents, organize tasks, provide services or processes information. For example, many standard procedures that exist inside of a company, like the processes of taxi service, customer service to answer customer questions, and insurance claims processing, etc. are repeated and standardized. What's more, these types of procedures are always complicated. They either require many documents or multi-way communication to accomplish their goals. These procedures involve many workforces, and it is straightforward to make mistakes. There are huge demands that create emerging technologies to describe and standardize these processes to coordinate and synchronize tasks among people to improve organizational efficiency. This brings the requirement of workflow and workflow applications, which have been developed over the last three decades, to improve productivity and accountability and help the company succeed in their field. Traditional business workflow are classified into two types: Enterprise application integration and document-oriented workflow. Enterprise application integration based workflow is the process of linking isolated systems like payroll and human resources systems automatically to simplify the business process. Document-oriented workflow, on the other hand, aims to show the presence of the content process, like their states, status, etc.

As computer technology advances, it plays a more and more critical role in scientific research. When scientists noticed that business workflows were successful in the business domain, they borrowed this concept to create scientific workflow technologies to model and automate the execution of scientific processes. Scientific workflows and business workflows share significant common features. In the past several decades in scientific research, workflows have continued to gain in popularity among science disciplines. They are widely used in biology [87, 29], astronomy [24, 139], physics [103, 43], seismology [89, 133], etc. In current cutting-edge workflow systems mentioned above, scientific experiments always contain a variety of complicated procedures, which themselves include a large number of different small steps that require tremendous human efforts to accomplish. No significant scientific work can be achieved without a computer, meaning scientists are now required to master computer skills. The current state-of-the-art of scientific workflows always involves distributed large datasets, sophisticated algorithms that are applied to these datasets, and complicated standard/custom processes to explore the data. Although there are many scientific workflow systems have been purposed and used in many domains, scientific workflows have their unique requirements and face enormous challenges: we are still at a shallow level of abstraction. In another word, programming is involved in every step of the experiments.

Domain scientists are a group of people who have been trained for years in their discipline area and are masters of their domain knowledge. However, they are not experts in computer science, and they have limited knowledge of computer science cutting-edge technologies (like deep learning, big data, etc.), coding using any programming language, or network environment, etc. Although these limitations exist, scientists still need to overcome these shortages and do their experiments using computers. When they have problems, the easiest way is to ask computer scientists for help. The other solution is training domain scientists with computer science skills and teach them basic computer science concepts and standards. In recent years, computer science classes has been required among many universities to train the scientists to write custom functions and use third party libraries to do data analysis. These are very hot research areas, and new disciplines are created for these purposes, such as bioinformatics, data science, etc. The essence of all these disciplines involves the art of manipulating data. However, this approach is time-consuming, because it creates a steep learning curve for the domain scientists to master computer science knowledge in order to be productive in their research field. The last but most effective solution is that engineers provide a workflow management system using workflow technologies which is highly abstract, declarative and hides all the computer science concepts to help the domain scientist compose their works.

1.2 Problem Statement

Many scientific workflow management systems created in the past three decades aim to help domain scientists. Several powerful workflow systems have gained popularity among biologists and general scientists for their versatility and usability. For example, systems such as Taverna [73], Galaxy [65] and Kepler [9] bring more intuitive and effective ways (like integrated web services and custom code interfaces) recognizing the facts that usability of a system mostly handicaps domain scientists and the difficulty of expressing their needs in computer science. They also bring different power and computational flavors meeting diverse application needs.

Apart from traditional resources these systems support, numerous novel applications require access to resources that are either not supported by these systems at all or demand substantial customization before being useful. To meet these needs, new systems like Tigers [122], PheKB [85], Swift [147], Jflow [101], and systemPipeR [13] etc. are being developed at a steady pace.

The need for custom analytics using distributed computational pipelines led to the design of PEPIS [88] which supports custom coding not available in popular libraries. There are other systems developed for those who are not as comfortable with coding. For example, PROPHETS [88] maps conceptual descriptions to canned functions in R libraries in meaningful ways. PINT [47] combines visual abstraction and data integration in computational pipelines for mass spec data analysis. bioLQM [107] provides a Jave library to model Biological networks. Designing sophisticated and complex computational pipelines using visual tools also is the focus of FUNN-MG [37] and LONI [44] while NOW [48], focused on abstract and modular workflow design.

Data integration and application development platforms such as SADI [148] and InterMine [130], and other workflows [12, 18, 82, 100] support another class of applications where data integration is a significant focus. There are systems like Tavaxy [7] to support both workflow and data integration in a single platform, which attempts to harness the power of both Taverna and Galaxy.

These emerging application development platforms paint a picture of new needs and underscore the fact that there are serious gaps between contemporary systems functionalities and what domain scientists needs. For example, the needs entail that workflow management systems must work with distributed heterogeneous resources. Our position is that most of the contemporary systems are not comprehensive or abstract enough for the community to embrace as a common platform for workflow management forcing users to develop custom applications at high cost. The current state-of-the-art is a patch work of various solutions from which users can choose, or develop custom applications to fit their needs at a substantial cost. The primary challenge is that in order to provide a user interface, we need to ease the use of workflows for domain scientists and create workflows with a quick learning curve and high usability.

Most of the time, these systems still require users to write custom code or create new components, and do not fully support extension. One reason for this is because these workflow systems aim to support specific domain analytics. The other is they are not abstract enough or do not support enough analytics tools to be free of coding. The coding skills are one of the hardest knowledge that requires rookie learners to spend a considerable amount of time to learn, practice and finally master and use in their research experiments. One of the fundamental purposes of the computer science discipline is teaching students coding using different programming languages, which requires that students know mathematics, statistics, etc. If a system can generate the program code without requiring programming knowledge, it not only takes advantage of the programming languages but also improves the capabilities of the system.

Every time a workflow is generated, workflow execution becomes a challenging problem. The way of managing the data resources and controlling of message flow among systems is not an easy task. Most current workflow management systems require users to configure the nodes and set up the execution order, which also requires the computer science knowledge of parallel execution. Providing an environment which frees users of parallel configuration not only saves workflow composition time, it also improves the usability of the workflow system.

1.3 Contributions

In order to provide domain scientists a user-friendly workflow system that abstractly solves their daily experiment requirements, a system with high abstraction and high usability is needed. In this dissertation, we present VisFlow, which is a visual language based workflow system using declarative primitives for scientific application design. In VisFlow, our goal is to offer a declarative data management and analysis platform that improves usability, scalability, and user-friendliness by supporting high-level abstraction, visual programming, data integration, workflow and visualization in a single platform. VisFlow is a workflow management system for composing, executing and visualizing scientific workflows. The contributions of this dissertation are as follows:

- **VisFlow is a declarative visual language that offers domain scientists a highly abstract way to compose complex workflows.** We propose this highly abstract visual language to help domain scientists compose workflows. VisFlow language has fifteen icons with their metaphorical meanings to help users abstract and understand their workflow easily. VisFlow supports CSV files, XML documents, and relational tables as the data resources.

- **VisFlow visual query builder enables domain scientists to generate SQL/XQuery script without procedure programming knowledge.** The visual query builder is a user interface that allows users to fill out a form of information to extract and manipulate the data resources by generating a SQL/XQuery script. It takes advantage of these two declarative languages and reduces the complexity of workflow compositions.

- **VisFlow executes workflows in parallel as needed without user configuration.** VisFlow has the capability of executing workflows in parallel. No background knowledge is required to configure VisFlow run in parallel. VisFlow has an algorithm which automates the execution order and monitors the status of all the executions.

- **VisFlow is a domain independent workflow management system that can be introduced to different domains.** We provide a number of example workflows from different domains to show how VisFlow can solve real-world problems effectively and efficiently. We also use complex examples to show how quickly VisFlow can help domain scientists construct a meaningful workflow.

1.4 Dissertation Outline

The remainder of the dissertation is organized as follows. In chapter 2, we briefly review the previous work and discuss their visual language design patterns. In chapter 3 we propose VisFlow, a highly abstract visual language with its definitions, icons and data flows. In chapter 4, the transform algorithms from VisFlow to BioFlow are discussed. BioFlow execution algorithm is also present here. Chapter 5 discusses the user interface and the VisFlow system, plus the unique features in VisFlow. Chapter 6 evaluates the VisFlow workflow management system. An example workflow that was originally implemented in the Taverna workflow system is compared with the steps of composing the same example in VisFlow to show how effective VisFlow is. A quality comparison is also provided to show how VisFlow improves pipeline design and free of execution. Chapter 7 shows comprehensive examples to illustrated how VisFlow handles complex workflow queries across different domains powerfully. In chapter 8 we conclude this dissertation and outline the future research plan.

Chapter 2: Background and Related Work

In this chapter, we discuss the current state-of-the-art in visual language design and scientific workflow management systems. Then we talk about the design of workflow systems which are widely used in different domains and have a visual user interface.

2.1 Visual Languages

Communication is one of the critical technical skills humans have mastered. Communication not only involves speaking and writing letters, but also contains drawing and processing unordered information. Our brains excel at perception and imagination. Just as people can describe their thoughts, they can also visualize them. Before humans created the written character, they used paintings to communicate and record significant events. Although we have created words and used them as our primary communication tool, drawings and graphs still play a big part in our lives. For example, diagrams are used to show the relationship among families, while great drawings and paintings show the imagination of humankind. Visual languages illuminate a way to describe the perception and comprehension of visual icons or graphs. With the emergence of computer technologies and the big data era, the importance of visual languages has not diminished. It plays essential roles in human-computer interaction, and it will gain more influence in the future.

Visual languages have been studied and investigated by scientific researchers for over three decades. Following are the formal definition of visual languages:

Definition 1. *A visual language is a set of diagrams which are valid "sentences" in that language where a diagram is a collection of "symbols" in a two or three-dimensional space. [102]*

This definition is highly abstract and easy to understand. Traditionally, visual languages are applied to reasoning [128, 30, 72, 83], mathematics [50, 123, 36] and object modeling [57, 150]. For example, Figure 2.1 is a spider graph indicates n inside A and B indicates that the set $(A \cap B) - C$ has at least one individual in it. Figure 2.2 is a VEX graph represented a function: $e(\lambda x.e(xx)(\lambda x.e(xx) = e(Ye)$. Figure 2.3 is an example query to find Least Common Ancestor (LCA) in a biology database. Figure 2.4 explains the three-coins problem [1].

Visual languages are used in these fields to help researchers explain theories, define relationships and model objects. There are many research efforts to apply visual languages to help musicians compose music [27, 26, 111, 28]. Figure 2.5 shows one of the illustrative examples in the OpenMusic basic visual

[1]Throw a coin three times to get different probabilities of heads and tails

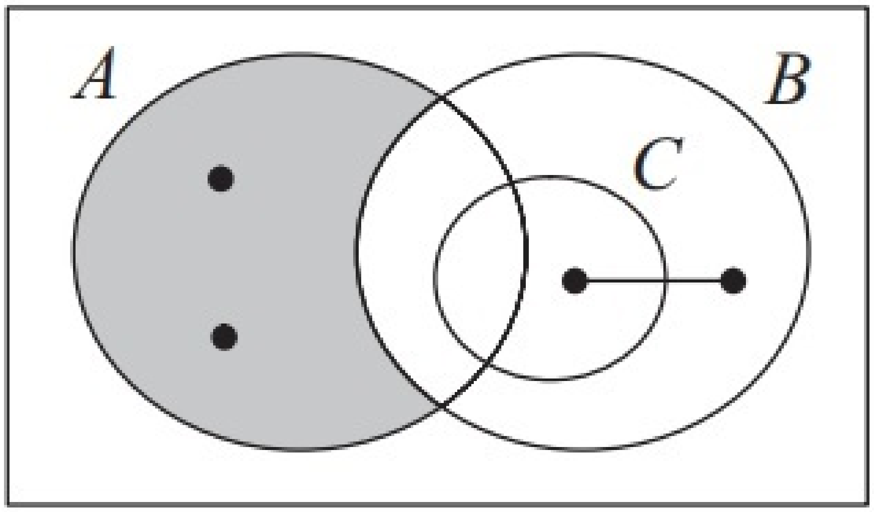


Figure 2.1: A spider diagram [30]

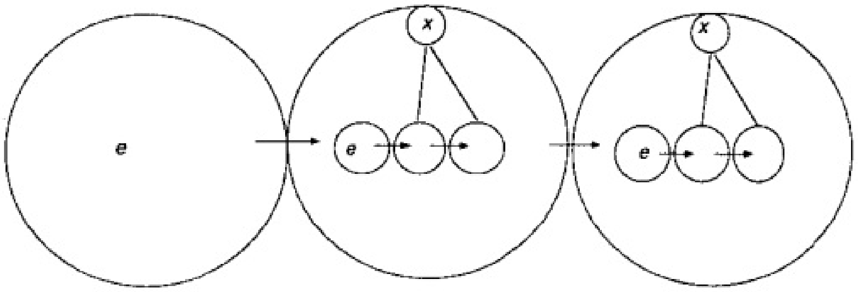


Figure 2.2: VEX [36]

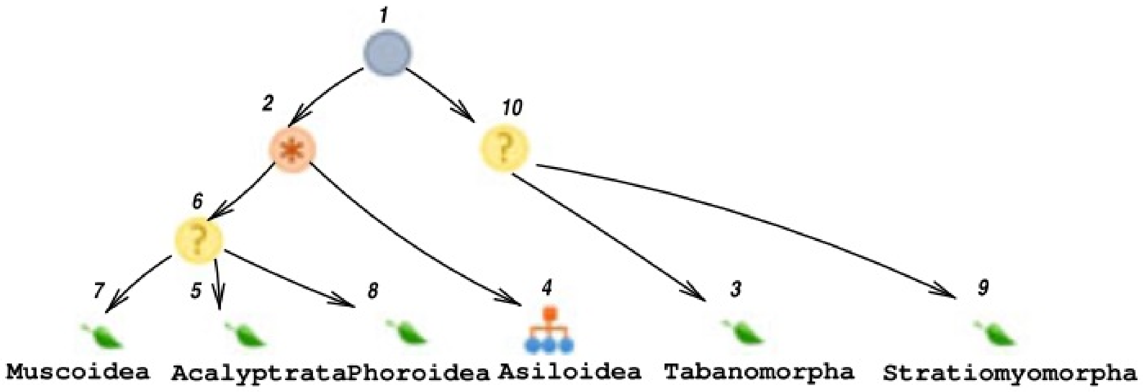


Figure 2.3: PhyQL [77]

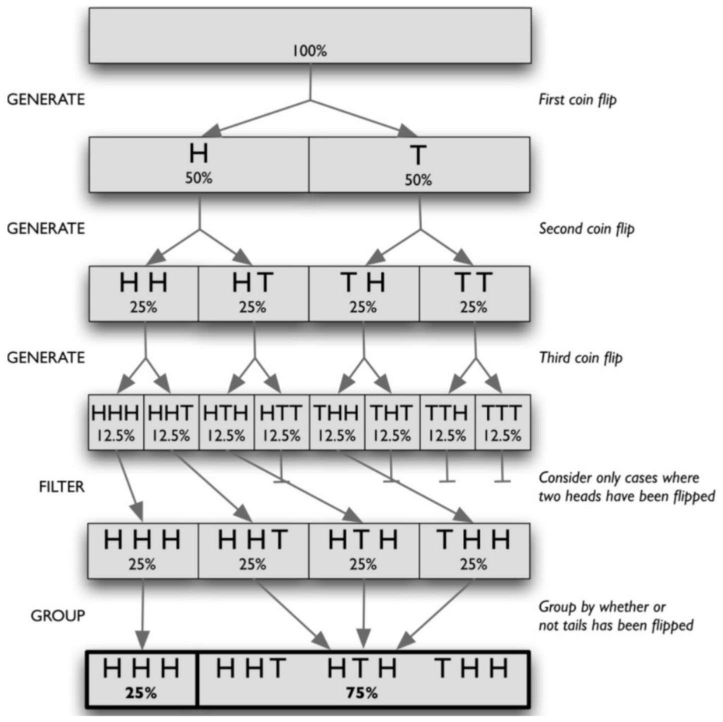


Figure 2.4: Probula [50]

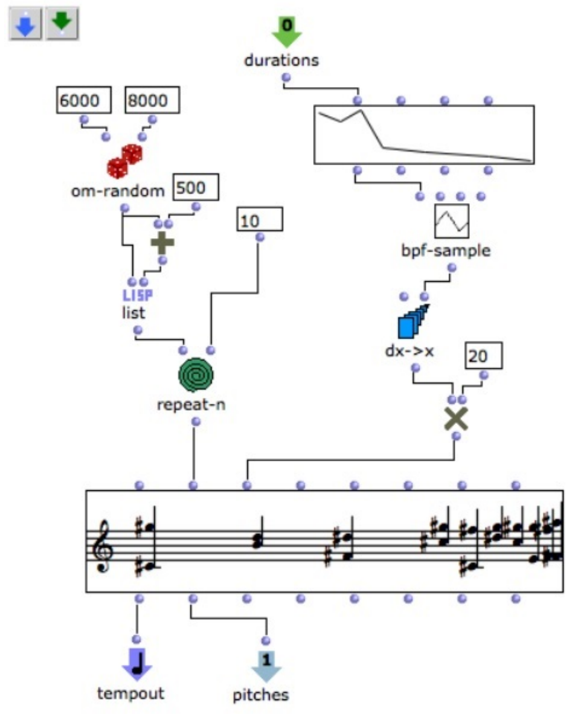


Figure 2.5: OpenMusic [27]

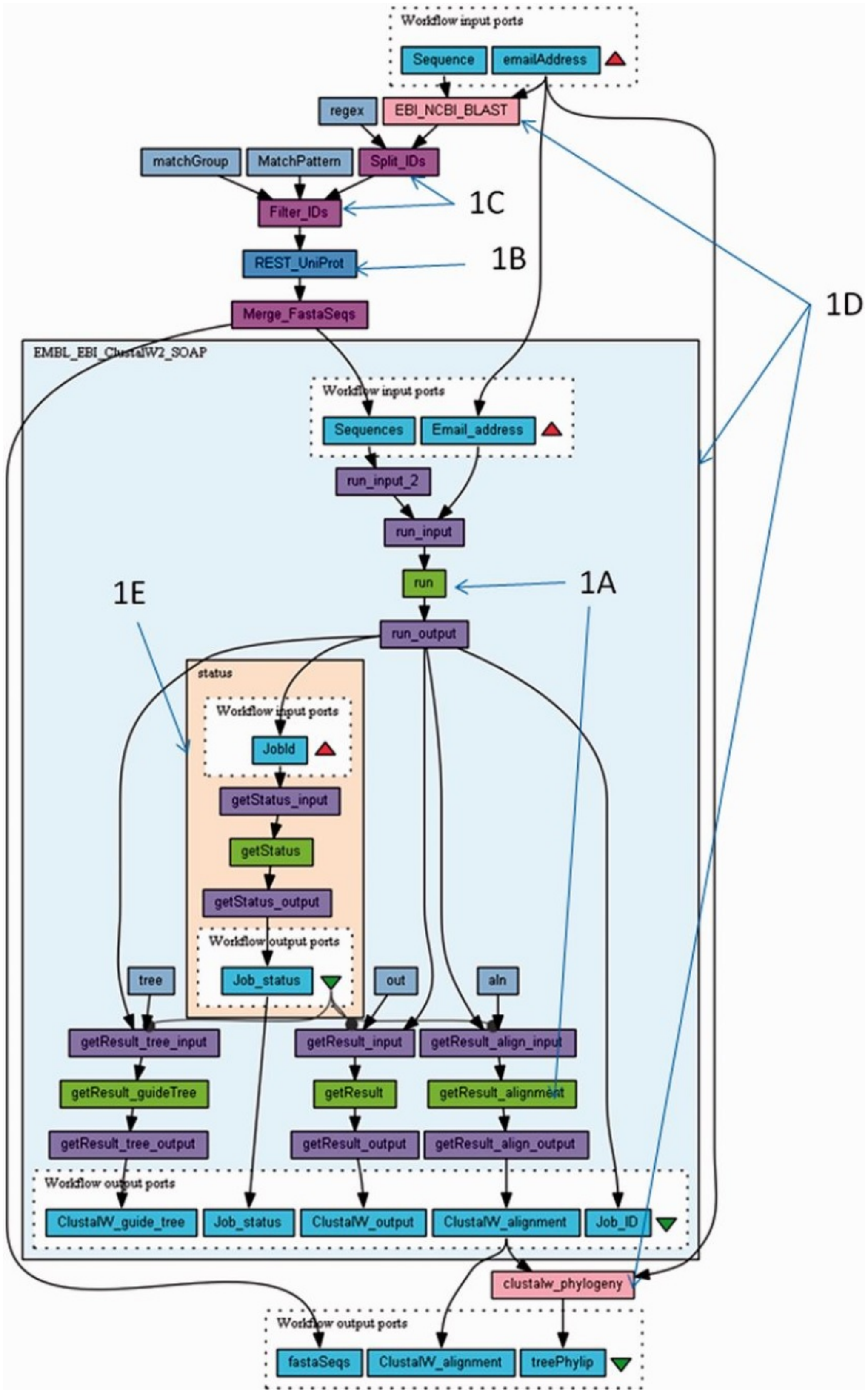


Figure 2.6: Taverna [149]

program editor to produce a piece of music. The prevailing philosophy of these visual languages is that the abstract functional elements, generated by the specific graphical editors like score editor and sound editors, are connected and configured by users to create rich music. Musicians are able to discover the pattern of music by viewing these abstract high-level graphs of the piece of music.

Another group of researchers is interested in designing a visual language to query specific data resources and retrieving results. For example, visual languages are designed to query linked data (graph database) by translating these languages to SPARQL [153, 156, 20, 121, 115, 67, 66, 62, 10, 21]. Visual graphic user interfaces also generate queries to heterogeneous databases [77, 79], traditional relational databases [119, 22, 138], XML documents [35], multimedia [151, 71], large network datasets [120], bibliography data[157], and semantic datasets [131, 132], etc. Visual languages not only help end users by automatically translating visual languages to database queries like SPARQL, SQL or XQuery and executing them on the back end, but also reduce the end user learning curve to grasp the lower level languages.

One of the key motivations for visual languages research is to facilitate interaction between humans and computers, which is the same goal that workflow management systems are interested. The differences between visual languages and workflow systems are in twofold: 1) workflow systems support more types of data resources and, 2) workflow systems provide more complex interactions and functions against datasets. We discuss workflow systems in detail in section 2.2. Not all workflow systems have a visual language based user interface, but many of them applied visual language concept in their workflow design. The basic idea for workflow management systems is to help domain experts to explore and analyze their data quickly and reproducibly. For example, Kepler [93], and Taverna [113] and Galaxy [65] are wildly used workflow management systems in bioinformatics, geoscience and astronomy disciplines. ASKALON [52], SCIRun [118], VisTrails [16] are all have graphics interfaces. Figure 2.6 shows an example workflow from Taverna that executes a phylogenetic tree workflow by running a similarity search and aligning similar gene sequences. There are many steps such as retrieving data from web servers, applying data analytics libraries against the datasets and using data visualization tools to show the results. These steps are configured in different icons and executed.

This section provides fundamental background information about visual language and different research areas that visual languages research has contributed. For more details about visual language research, please refer to other survey papers [92, 102, 49, 144, 25, 23, 108, 32].

2.2 Scientific Workflow Management Systems

Workflow technologies are emerging as the dominant approach to involve many trial tasks and communicate with collections of servers. Workflow is the glue for distributed services and is the child of the science and engineering. The formal definition of a workflow management system is as follows:

Definition 2. *A workflow management system (WMS) aids in the automation of those operations, namely, managing the execution of fundamental tasks and the information exchanged between them. [40]*

The concept of workflow has existed in the business and science world for a long time. The business workflow or business process is a collection of well-defined tasks that are in the official enterprise business. For example, reporting phishing email is a standard procedure whenever the possible email is scanned by the system or reported by employees and needs to be submitted to the anti-virus center for analysis and IP address comparison. These steps, such as scanning, submitting the email to the server and analyzing, are the same procedure for any possible email to identify if it is a phishing email. Another example would be hiring a person for a company and coordinating the interview process. These cases are quite similar and require a lot of the same operations. Automating these steps and abstracting them as business workflows reduces the enormous human effort and provides high business value to the companies. Companies like Microstrategy[2] and ServiceNow[3] are two successful companies in business workflows that offer service for modeling business processes like these. Nevertheless, business workflows are more like traditional procedural programming languages which are perfect to describe steps but are not good at dealing with large scaled data. Scientists are more interested in exploration and mining data sets, in addition to visualizing results in different formats. A scientific approach always contains a series of analytic steps, which include data collecting, data cleaning, and data processing. Scientific workflow systems provide an automated environment to help the scientists to describe the scientific discovery process. VisFlow is an workflow management system designed for domain scientists, so the business workflow systems discussion is skipped. For more information about business workflow systems and the comparison of business and scientific workflows, please refer to these papers [142, 94, 154, 8].

At high levels of workflow design, a workflow is an abstraction of a set of predefined actions and the relationship among them must be satisfied in order to accomplish the scientific experiment. A standard data mining process for a data scientist includes data collection, data cleansing, exploratory data analysis and communicating a visualization report as a typical workflow scenario. At lower levels,

[2]`https://www.microstrategy.com`
[3]`https://www.servicenow.com`

a workflow is a computer program that can be executed on computers. Generally speaking, there are four different stages of a scientific workflow lifecycle:

- Workflow composition: compose the workflow using different technologies, e.g., script, graphics, semantics
- Workflow translation: transfer and map the abstract workflow to lower level language that computers understand
- Workflow execution: execute the workflow parallel or continuously
- Result visualization: show the results to end user

There are three ways to compose a workflow: script based, semantic-based and graphics based. When the workflow is defined and configured, it is mapped to lower level languages like C++, Java, Python to be executed. A number of optimization technologies may be applied to reduce the execution time. After the final results are generated, visualization tools are used to show meaningful information to users. The scientific workflow systems can be classified simply by the composed type.

Many workflow systems like Jupyter[4] [86], BPEL [135], DAGMan [137], Pegasus [41] and CoG [145] use plain text editors as their user interfaces. These type of workflow systems have always defined the rules of grammar with complex semantics and extended syntax. These languages contain annotations to establish a relationship among process and resources. They also have control units to construct loops or execute parallels. The whole workflow is translated to a procedure or declarative language to execute after the workflow is defined. Although it worked very well in systems like Jupyter, writing a textual workflow program is still difficult and time-consuming.

Semantically based workflow refers to workflow systems that can help users compose a workflow. For example, Rönkkö et al. [124] design a preprocess to solve the selection and sequencing problem by using characterizations and a reachability algorithm. Gil et al. [60, 99, 84] design a workflow system that contains augmented editors with intelligent assistance to help users keep track of constraints across the components, specifying workflow flexibly. In these workflow systems, the workflow has been abstracted as a hierarchy structure and features a different level of descriptions of the workflow aspects. A semantic layer creates abstract workflow elements for a number of execution components, and artificial intelligence is applied to analyzing user interactions to achieve automated workflow composition.

Graphic based workflow uses abstract meaningful icons and edges to generate the workflow. These systems contain a few basic graphical icons that correspond to the workflow nodes and edges. Compared

[4]Jupyter do provide a list of block of coding sections that only allow write python code.

to a script based workflow system, workflow management systems that have graph-based querying user interface usually provide more flexibility than the form-based or text-based paradigm by using node-link diagrams to create arbitrary querying patterns. They dramatically improve usability and reduce the learning curve for users. VisFlow is a graphic based workflow management system. A detailed discussion about the current state-of-the-art in graphic based workflows will be provided in section 2.3.

2.3 Related Work

There have been many visual language projects and workflow management systems created during the past three decades. We are looking into workflow management systems that have not only a graphic interactive user interface but also the capability to analyze different data types and heterogeneous data resources on distributed web services. We explored and surveyed visual language and workflow research conferences and journals to find systems that meet these two requirements like VisFlow.

Kepler [93] is built on the Ptolemy II system[5] designed by University of California, Berkeley. Kepler is an open source workflow engine with contributors from a range of different domains. Kepler is focused on data analysis and modeling, which makes it suitable for various disciplines such as bioinformatics, geoscience, oceanography, astronomy, etc. Kepler inherits the Ptolemy II user interface and refers to components as `actors`. The workflow `actor` is a re-usable independent element that allows operations on data resources or web services. Kepler provides a large variety of `actors` by default. Every `actor` has many input and output ports and `actors` are grouped by mapping output ports to input ports. Kepler also allows users to create modules and plug them into the workflow. Nested workflows are provided by computational controllers called `directors`, which also enable the `actor` to communicate when separated from the overall workflow execution. `Directors` are responsible for deciding the order of actors to be run. This decision is based on the inputs, the operations information and the outputs. In other words, depending on the `director` used in the workflow, the `actors` may be triggered in different orders and are controlled by different directors.

Taverna [149] is an open source workflow system primarily focused on biological experiments using Simple Conceptual Unified Flow Language (SCUFL) [112]. The input in Taverna is named `sources`, while the output in Taverna is labeled as `sinks`. Taverna named the component as service in the workflow. Taverna provides eight default services for users: Local java services, WSDL Web Service - secure and public, RESTful Services, R Processor services (for statistical analyses), Beanshell scripts, XPath scripts, and Spreadsheet import service. Users also are allowed to import their services. There are two types of link relationship among services, `sources`, and `sink`: data links (labels the data

[5]`http://ptolemy.eecs.berkeley.edu/ptolemyII`

flows between node port) and coordination links (control the order of execution separated from data dependency). The idea behind these two types of links is that Taverna plans to take advantage of the execution schedule by requiring explicit ordering constraints. The workflow is executed in a push manner, started from the sources and finished in the sinks. Users also allow partial execution fo the workflow.

Triana [136] is a visual problem-solving environment initially used for the processing of gravitational wave signals, and later it was expanded to signal, image and audio processing in addition to statistical analysis. The component in Triana is named `unit` and consists of an identifying name, input and output ports, and many optional parameters. The icons used in Triana are simple blue rectangles with different background colors to identify services and jobs. Triana also allows users to write their code in other languages, but they need to provide a wrapper to make it executable from Triana. The middleware layer called Grid Application Prototype (GAP) enables Triana to communicate with web and peer-to-peer (P2P) services which allow Triana distribute sections of a workflow to remote machines. Triana uses both data and control flow to set up the execution of the workflow. The unit executions are triggered when data are arriving at the component or used to operate directly from control commands.

Galaxy [65] is a web-based workflow system for life sciences to facilitate genomics research, which requires massive computational experiments and tools for supporting these procedures. Galaxy has integrated a large number of tools for life science researchers to use in the workflow with detailed documentation. Galaxy emphasizes reproducibility by providing mechanisms that capture and track the metadata, recording the editing history and easily allowing users to share their workflows. In the user interface, only one type of component, called `box`, is used to represent the tools collected in the tool panel. Two `boxes` are allowed to connected if the input/output types are compatible. A tool in Galaxy's toolbox is a piece of code that can be compiled and executed in command line environment. Adding a new tool in the tool box is easy, developers write a configuration file that describes the command line commands, plus detailed specifications of input and output parameters. One of the advanced Galaxy features is that it allows Taverna workflows as part of a Galaxy workflow.

Askalon [52] is a workflow system designed for Austrian grid infrastructure. It uses Unified Modeling Language [125] and an XML-based language named Abstract Grid Workflow Language (AGWL) [51] to represent the workflow components. The workflow components are classified as `initial node`, `activity final node`, `atomic activity`, `merge node`, `fork node` and `join node`. In each workflow, an initial node and a final node are required. A list of `atomic activities` is grouped by a compound activity to reduce the size of a workflow. After these components are placed in the canvas, data flow edges and control flow edges are connected by users. Data flow edges are noted in the trans data path. The control flow edges are complimentary of data flow to help users control the execution of a workflow. The control flow constructs include `sequences`, `DAG`, `for`, `forEach`, `while` and `do-while` loops, and `if` and `switch`, `parallel`, `parallelForEach` and `collection` iterators. Askalon provides workflow fault-tolerance and static and hybrid scheduling by delivering the scheduler and the resource management system (GridARM). GridARM is a data repository that not only provides a schedule for all dataset information, but also provides data discovery techniques based on quality-of-service matching to provision the resources automatically.

WS-PGRADE/gUSE [81] is a generic distributed computing infrastructure (DCI) gateway framework for various disciplines in a Grid environment such as biology, seismology, astronomy, and neuroscience. WS-PGRADE is the web-based user interface, and gUSE is the web container that enables the management, store, and execution of workflows. It also acts as the middleware connect to DCI bridge, which provides flexible and versatile access to all DCIs in Europe. WS-PGRADE/gUSE allow users to test equality, inequality and containing the input files. WS-PGRADE includes a combination of portal components to be connected and submitted to gUSE services to execute. Workflow templates are provided to enable the workflow to publish, share and search. The workflow is represented using XML language. The first step of designing a workflow is generating the abstract workflow using the graph editor. The second step configures each step with concrete jobs, binding data input files, and output files. Each icon is a job, which contains algorithm configuration, resource configuration, and port configuration. It supports web service call, runs a binary program or embeds another workflow. Data-avenue[68] is the manager for data transfer among file storage systems using different network protocols to handle interfacing with data storage systems.

Wings [61] is a workflow system build on Pegasus [41]. Wings is based on semantic technology. The primary component in Wings is named `node` and it consists of user-defined activities. The activity and workflow are represented as semantic objects inherited from the root objects in their domains. An activity is described as a component that carries out a collection of computations. The first step in Wings is creating a workflow template, and the second setup specifies the data bound to the workflow and creates the instance. The last step is to transfer this workflow to Pegasus and execute it. The workflow template is stored as Resource Description Framework (RDF) files that contain a list of semantic objects using Ontology Web Language - Description Logic(OWL-DL) [104] as the representation language. The node in the workflow repository has different names with pre-defined input-output ports. The edges are classified as `inputLink`, `inOutLink`, and `outputLink`. Using semantic web technologies, Wings can validate the workflow by the constraints of the components and the characteristics of the data. Wings uses artificial intelligence planning and semantic reasoners to help users create workflows based on selected template abstract workflow and concert workflow generation algorithm.

VizBuilder [71] focuses on data integration and uses on the LifeDB [15] database management system. It provides an extensive graphical query interface for user input. It contains `read`, `write`, and `process` components with specific configurable actions like `group`, `slice`, `if`. At high levels, the icons in VizBuilder uses `read`, `write` and `process`. Besides these three abstract components, concrete components like `table`, `function`, `input`, and `if`, etc. are provided to allow users to click and configure different fields. The whole workflow is defined using XVML - a markup language designed by XAML [6]. After the definition of the whole workflow is done, it will be translated to the declarative querying language BioFlow [75] and executed.

SCIRun [118] is a workflow system that allows users to interactively create and manipulate complex visualizations and directly edit data flows from the University of Utah. Currently, it is extensively used to simulate and show 3D brain simulations [33] and image processing [59]. It is a problem-solving environment that provides a high-level dataflow workflow to connect different software modules for visualization. The components in SCIRun are defined as a `module`, `port`, `quantities datatypes`, and `connection`. A `module` is an algorithm or operation which is the fundamental component in the workflow. The icon used in SCIRun is a grey rectangle with the name, actions, and input-output symbol. A range of predefined activities is stored in the right-click context menu, allowing users to select and add them to the workflow.

VisTrails [16, 31, 126] is another open source workflow system that aims to support data exploration and generate multiple-view visualization pipelines. They use their XML schema to capture all the information in VisTrails. VisTrails provides a change-based provenance API to trace the record of changes made by different users. The component in VisTrails is called a module. The module element in the VisTrails schema providing functionality like running scripts allows access to web services. It is straightforward to add new module elements if they fit the schema. VisTrails delivers a cache manager to schedule the execution of the modules. The cache manager examines the data dependencies and shares the data resources among different modules.

HyperFlow [14] is a simple approach to program on a workflow system. It takes advantage of the executive management in workflow systems by providing different types of parallel execution supporting and providing a user coding interface for each icon. The component in HyperFlow is called a `process`, which assigns functions to execute and generate outputs for the next operation. Signals are passed between process nodes and trigger function executions. HyperFlow uses JSON[6] objects to store the component information and configuration. There are four process types: `dataflow`, `choice`, `foreach`

[6]`https://www.json.org/`

and `join`. Signals have four categories: `next`, `done`, `count` and `merge`. This workflow emphasizes control of execution programming pieces ordered by the graph structure. HyperFlow only supports Javascript language, and the backend server is Node.js.

The Waikato Environment For Knowledge Analysis (WEKA) [69] provides a uniform workbench for users to access state-of-the-art techniques in machine learning. WEKA has a user interface named Knowledge Flow to allow incremental model building. The components in WEKA knowledge flow are called `dataSources`, `dataSinks`, `filters`, `classifiers`, `clusterers`, `evaluation`, and `visualization`. Each component has a group of individual nodes for a different purpose. For example, in a `dataSources` component, there are icons for CSV file loading, database loader, etc. to import data resources to the workflow. Using these pre-defined icons, users can build a data mining process as needed. `DataSinks` provides methods to save stream data in the workflow. `Filters`, `classifiers`, and `clusters` are collections of machine learning algorithms to process data. `Evaluation` contains a list of evaluation methods. `Visualization` includes predefined visualization methods. The edges between components are data edges to transfer data from one component to another.

The Konstanz Information Miner (KNIME) [19, 17] is a modular environment that is primarily used in data mining research. It is an open source data mining, machine learning, and data integration platform. It supports a variety of analytic algorithms in the domain of data mining and machine learning. It has been used in pharmaceutical research, business intelligence and financial data analysis, etc. KNIME only supports two-dimensional table data. The component in KNIME is an abstract of node model, node dialog and node view. Node model controls the computation of each node. Node dialog provides a dialog for custom settings that affect the execution. Node view field records the different views into the current model. By default, KNIME offers a large variety of nodes, data I/O, manipulation and transformations in the interface. Although the whole system is written in Java, it also supports integrated WEKA toolkits and R environment. Plenty of predefined data integration modules are already included in the system and ready to use.

Orange [42] is another open source machine learning and data mining workflow written in Python. It has a large toolbox in machine learning and data mining. It provides many standard visualization toolboxes like scatter plot, box plot, etc. These visualization toolboxes are intelligent enough to find the best projections with the best class separation to show the data by ranking the combinations. Orange components are called widgets. The provided widgets include `data`, `visualize`, `classify`, `regression`, `evaluate`, and `unsupervised`, which contains groups of icons that support data input, filtering and other standard procedures in the data ming or machine learning field. Users also need to configure the input-output edges among icons. The whole project is written in C++ with wrappers for supporting

Python script.

Rapidminer (formerly YALE) [105] is an open source platform for data mining and knowledge discovery in databases (KDD) process with a rich variety of data mining algorithms. It monitors the KDD process as separated steps of applying data analytics tools against data. It uses an XML document to present the methods used in the system and it can be easily extended. It also provides robust data visualization toolboxes.

In this Chapter, we have investigated fifteen workflow management systems that are widely used by scientists. These workflow systems have been used in a variety of domains, like bioinformatics, geoscience, astronomy, etc. They are capable of dealing with different types of files (XML documents, CSV files, and JSON files, etc.), various data resources (SOAP web services, RESTful web services, and FTP web services. etc.), and multidimensional databases. Some of the workflow systems are running on grid infrastructures or clusters, and some are running on distributed systems, the rest are running on a single machine. They all offer highly reusable components to reduce the time of composing the workflow.

Chapter 3: VisFlow Language

VisFlow is a web-based visual ad hoc workflow management system using distributed resources such as online databases, web services, deep web applications, analytics, custom application codes in SQL, XQuery, Python and R, and data visualization libraries. VisFlow leverages the experiences of successful and popular systems such as Taverna, Galaxy, and Kepler. It tries to balance usability, computability, and coverage. By that we mean we focus on simplicity of application coding using abstractions, offer a mix of tools to be able to express application needs (allowing the use of libraries, online resources, custom codes, and established services), and enable the use of a set of arbitrary resources.

3.1 VisFlow Data Model and Language

The data model of VisFlow is a tuple of the form $\mathcal{M} = \langle \mathcal{O}, 2^{\mathcal{O}}, \kappa, \mu, \omega, \phi \rangle$. $\mathcal{O}$ is a set of objects of the form of comma-separated values, XML elements or a relational tuple. κ, μ, ω and ϕ are key discovery (e.g., GORDIAN [129]), schema matching (e.g., S-Match [63]), wrapper (e.g., NTW [39]) and form filling (e.g., iForm [141]) functions respectively. A VisFlow database over the model $\mathcal{M}$, on the other hand, is a tuple of the form $\mathcal{D} = \langle \Lambda, \Sigma, \alpha, \Delta \rangle$ such that Λ is a set of labels, $\Sigma \subset 2^{\mathcal{O}}$ is a consistent set of data sets called schema, α is a assignment function of the form $\alpha : \Lambda \rightarrow \Sigma$, and Δ is a set of database operations such as specialty operators τ (transform [78]), χ (combine [78]), and λ (link [70]), and traditional operators σ (selection), π (projection), $\times$ (Cartesian product), $\bowtie$ (join), $\setminus$ (difference), $\cup$ (union), $\cap$ (intersection) including $\mathcal{G}$ (group-by) and aggregate functions. In this context, any member of $S \in \Sigma$ is consistent if every object $o \in S$ is also in $\mathcal{O}$ and for every object $o \in S$, they have the same type – i.e., they are all either CSV, XML or tuple types over a single scheme.

The language $\mathcal{L}$ of VisFlow is a tuple of the form $\langle \Sigma, \mathcal{I}, \prec, \Delta, \mathcal{F}, \Psi \rangle$ such that Σ is a consistent database, $\mathcal{I}$ is a set of visual icons, Δ is a set of database operations, $\mathcal{F}$ is a set of predefined library functions, Ψ is a mapping function of the form $\Psi : \mathcal{I} \rightarrow 2^{\Delta \cup \mathcal{F}}$, and finally $\prec$ is a partial order among the elements of $\mathcal{I}$ and $\Delta \cup \mathcal{F}$. Intuitively, a workflow W in $\mathcal{L}$ using the icons in $\mathcal{I}$ denotes a workflow involving the elements in Σ expressed as a partial order over $\prec$, and Ψ transforms the meaning of the elements of the icons into database operations Δ and library functions $\mathcal{F}$ over the scheme preserving the partial order expressed in the workflow W. Let I be the meaning function that maps W over Σ to a set of consistent sets Σ' such that $\Sigma' \subset 2^{\mathcal{O}}$, i.e., $I(W, \Sigma) = \Sigma'$. Then, the mapping Ψ is admissible, if the intended meaning of W and $\Psi(W)$ are identical, i.e., $I(W, \Sigma) = I(\Psi(W), \Sigma) = \Sigma'$.

3.1.1 Elements of VisFlow

VisFlow assembles and adapts several well-known database and Internet functions developed by various research groups, for example, KEGG database, Picard tools, Google Maps, etc., often for unrelated purposes, into a special set of database operations such as transform (τ), combine (χ) and link (λ) to ease end-user workflow programming experience in which users are less aware as well as less concerned about the database details. In particular, significant heterogeneity and conflict problems arise when merging different data resources. For example, schema mismatches in similar databases occur commonly, and often key constraints and relationships among objects are unknown. Combining such sets of objects from disparate databases into seemingly identical data sets give rise to semantic conflicts that need to be systematically dealt with using sound information theoretical principles, placing humans in the loop for confident conflict resolution.

Because we allow ad hoc integration, and we also support importing previously unknown and unexplored relations into the application, the potential exists that the representation of entities (and relationships) may not conform to database view of similar entities, or representations of two imported sets of entities may differ, i.e., have different primary keys or scheme. Therefore, there is no guarantee that the key constraints will hold either. Selection, project or cartesian product operations do not pose any difficulty over such ill-explored imported relations. But a significant difficulty arises when we want to collect two or more sets of entities in one set in a way similar to union operation in relational algebra or want to increase what we know about entities in different sets by extending the scheme as in natural join. These types of operations are common in data integration domain and are usually called *horizontal* and *vertical* integration respectively. Horizontal integration is often known as entity extraction or resolution [116], and vertical integration as data fusion [46] or record linkage [152].

In both types of integration, an important mechanism for identifying objects or entities, called *key discovery* or *entity recognition* is used. Key discovery helps us move from syntactic to semantic integration by allowing us to focus on the conceptual equivalence of objects, not on representational equivalence. For example, the schemes *protein(PID, CommonName, Function)* and *proteins(Gene, EncodesProtein, CellFunction, Organism)* may seem to be completely different and union incompatible in relational sense, but they are likely representing the same set of objects almost entirely differently, i.e., *CommonName* $\equiv$ *EncodesProtein* or *PID* $\equiv$ *EncodesProtein*, and *Function* $\equiv$ *CellFunction*. An object-based horizontal integration, which we call a *combine* operation, will take a full outer union of the two sets only if they agree on the discovered keys for each of the relations whereas a union operation in a classical relational model will yield nothing. Similarly, a natural join in a traditional model will fail, but a vertical integration, which we call *link* operation, will succeed if the discovered key of one relation is contained within the key of the other relation. In order to formalize the VisFlow concepts of the special operations *link* and *combine* we discuss ahead a set of functions they leverage.

3.1.1.1 Key Discovery Function κ

In a value-based system such as relational and XML databases, some of their attributes represent conceptual objects as keys. Relational databases leverage these keys in multiple ways to reconstruct the actual complex object that lay normalized and spread all across the database. While these key constraints are known at design time and maintained throughout the lifetime of the traditional databases, without intimate knowledge, such constraints on the objects gathered online from disparate sources, and their relationships with the database objects are rarely known. Key discovery functions help reconstruct these constraints to help query formulation and object reconstruction.

We have adapted the Gordian key discovery function [129] for VisFlow for several reasons. Gordian is simple, scalable and efficient for automatic discovery of simple and composite keys and is equally suitable for finding key column-groups in relational data, or key leaf-node sets in a collection of XML documents with a common schema. Since we predominantly use XML and relational data sets in VisFlow, it seems to be a fitting choice. With a slight twist, it can also be used to find keys in CSV data sets. Since we perform key discovery always dynamically at every execution, we pay particular attention to efficiency.

3.1.1.2 Schema Matching Function μ

However, for the reconstruction of a complex object, though key discovery helps, keys cannot be appropriately leveraged for the object reconstruction if the attribute names or schema is heterogeneous. Especially when information is assembled from numerous online sources, often with no prior inspection, semantic reconciliation of structural and representational differences becomes extremely difficult. The problem compounds manifold when achieving interoperability automatically becomes the goal. In VisFlow, we resolve structural and semantic heterogeneity at execution time every single time because we do not keep query related details to avoid “view materialization” [34] related drawbacks. Furthermore, schema matching is a repetitive and key operation we perform in almost all database operations since the user view of the application in VisFlow is abstract and usually does not have any relationship with real database schema over which the applications are designed. So, it is essential that we use a scalable, efficient and effective schema matcher in VisFlow.

We have adapted the S-Match [63] schema matching system in VisFlow. S-Match supports an extensible API for new algorithms an envisages the possibility of application-specific plug-ins to tailor its functionality according to domain-specific background knowledge. The base system comes prepackaged with three different algorithms tailored for several data structures such as business catalogs, web directories, conceptual models and web services descriptions into lightweight ontologies and semantic correspondences between them. In VisFlow, however, we adopt a human-in-the-loop approach and support a dominant and matching reassignment feature in the event automatic matching results in unexpected and erroneous schema correspondences.

3.1.1.3 Wrapper Function ω

Wrappers are usually extraction rules that apply to data returned by an online source in response to a query and can be lifted and understood. In modern web sources, most communications are in the form of texts – often they are in the form of semi-structured CSV documents, XML, or HTML, or in pure unstructured texts. Writing a wrapper in XPath, XQuery and so on is not that difficult when the source characteristics are statically known, and the number of sources is finite and small. For example, data integration systems such as Galaxy or BioExtract Server [95] use pre-fabricated wrappers to integrate a handful of known databases and often limit query formulation to limit wrapper variabilities that need to be fabricated to extract data.

However, when the properties of returned pages are only known at query execution time, such rules can only be learned dynamically or extracted by analyzing the returned pages, and can never be prefabricated. While some approaches try to leverage crowd [38] or unsupervised learning [39] and thereby attempt to reduce cost and improve scalability, such techniques are ineffective in dynamic and ad hoc integration frameworks such as VisFlow. In such systems, automatic extractions [117, 11, 106] become necessary and often are the sole choice. In the current version of VisFlow, we adopted a variant of the FastWrap wrapper induction system [11] which leverages a table structure recognition algorithm using repetitive element detection and the Boyer-Moore algorithm. FastWrap is linear in complexity and requires only one repeat of a record pattern, and is capable of inducing wrapper rules for CSV, XML, HTML or plain text document.

3.1.1.4 Form Filling Function ϕ

Communications with external sources usually are carried out through internet ports that open a window into their protected contents. Such windows can be generalized, abstracted and viewed as forms that require a set of appropriate parameters to grant access, and accept and process a request for information in prescribed formats, e.g., FTP and HTTP protocols, and web services. While most such forms take parameters as query conditions to execute prefabricated queries, some services also allow flexible queries in a bid to cater to users' needs. For example, systems such as iForm [141] focus on mostly the former kind and do not support the submission of user queries. But, the scientific databases such as Global Boundary Stratotype Section and Point (GSSP), or Golden Spikes database at RPI [96], and SHARE [143] allow users to submit SPARQL queries directly to interrogate the database by user applications in ways similar to VisFlow.

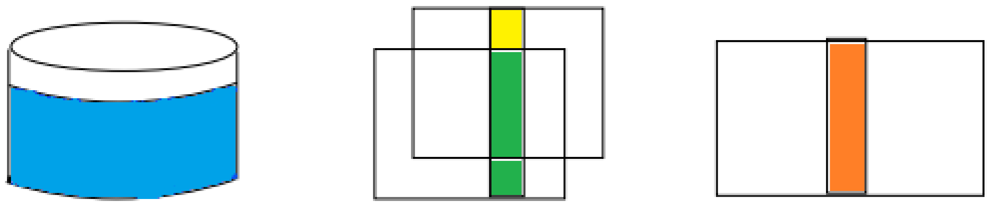
The form filling function ϕ, called *pForm*, that we have developed is an abstraction of popular and current submission protocols. This form function establishes schema correspondence with a data portal using a schema matching function and a capability discovery algorithm to match query needs with the site description. pForm is able to determine the nature of the portal and adjust submission parameters fully automatically leveraging the knowledge in the registry about the site and can be effectively used to communicate with sites using FTP, HTTP, or HTTPS that accept user queries.

3.2 VisFlow Icons

VisFlow provides a web-based user interface to help domain scientists to generate complex workflows. VisFlow icons are classified into data icons, transformer icons, visualization icons, workflow icons, and editor icons. In total there are eighteen meaningful icons to help domain scientists generate a meaningful workflow. Data icons have three icons in the data icon bank - *data*, *combine* and *fusion* icon. Transformer icons contain *adapter*, *analytics* and *library* icons. Visualization icons include *terminal*, *printer* and *general I/O* icons. Workflow icons contain *if*, *repeat*, *connect*, *wait on*, *procedure*, *edge* icon. Editor icons hold the *select*, *delete* and *undelete* icon.

3.2.1 Data Icons

A *data* icon shown in Figure 3.1a represents a stored collection of data from which data can be read or to which data can be stored. It abstracts any data resources including web services imported by the workflow. It can have unlimited incoming and outgoing arrows to show the flow of data. It must have at least one incoming or outgoing arrow. Each arrow may have a distinct type of data items. VisFlow supports all three types of Internet-accessible data - tabular, XML or semi-structured, and text - the abstract view is still tabular, by recognizing that XML and text data are mainly nested tables. Each data item is enriched - by that we mean each data item accompanies rich metadata that adequately describes it. In order to be able to access data items using a `data` icon, VisFlow requires that those data items must be enriched, indexed or cataloged by VisFlow in advance. The resource management



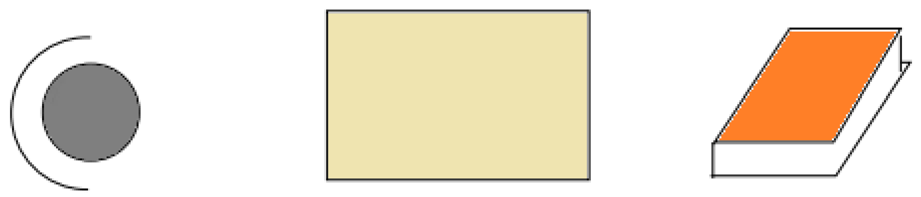
(a) Data Icon (b) Combine Icon (c) Fusion Icon

Figure 3.1: Data Icons

(section 5.3.1) provides the meaning feature to help users select and import data resources into their workflow. It has the register page for new data resource registration, and it also has a resource page to browse registered data resources. The *data* icon uses the top-k matching algorithm to help users select best-matched resources. While ground data resources can be a collection type, a data resource in VisFlow can also be virtual. Meaning, while they are registered and described in VisFlow resource repository, they are in reality stored in online sources, called *deep web* databases. These databases can be accessed using HTML forms, web services or FTP protocols, the contents of which are always dynamically retrieved. Since such access protocols may require input values, they are generally modeled as queries in VisFlow.

A *combine* icon aims to help data integration and data heterogeneity in two multidimensional resources. Two objects or entities in two tables can be merged if they are identical objects. The identities of objects are discovered using a key identifier algorithm like Gordian [129] and Twiner [90]. Once the possible composite key pairs from two tables have been returned, schema matching algorithms are applied to discover the best-matched pairs. It supports matching algorithms such as S-Match [64] , PruSM [110] , ontoBuilder [58] and Cupid [98] to match composite keys in two different tables. The selected schema matching technique should be appropriate to resolve possible heterogeneities in these tables. The best match key pairs have been selected based on the number of keys in each table and their similarities. These pairs are used as the identity of these two tables when merging them into one table. Unmatched attributes are added to the object resulting in schema expansion [127]. As a result, enriched distinct objects are attached to a table in a manner that is similar to a relational set union. But differently from a set union, the identity of objects are not based on the properties; instead, it is based on identifying properties or keys. Once the keys are found to be identical, distinct properties are unioned to make up a new entity. For *combine* icon, there must be at least two incoming arrows, and at least one outgoing arrow.

Similarly, the *fusion* icon also merges two or more tables based on key identification in ways similar



(a) Adapter Icon (b) Analytics Icon (c) Library Icon

Figure 3.2: Transformer Icons

to join in a relational model. It also has all the schema matchers and key identifiers that the *combine* icon has. But, different from SQL natural join operator, VisFlow's fusion links objects based on entity identifying keys and common attributes. In other words, to fuse or aggregate two objects, the objects must have a set of common attributes after schema heterogeneity is resolved and those common attributes must include the "object key." Like the *combine* icon, there must be at least two incoming arrows, and at least one outgoing arrow.

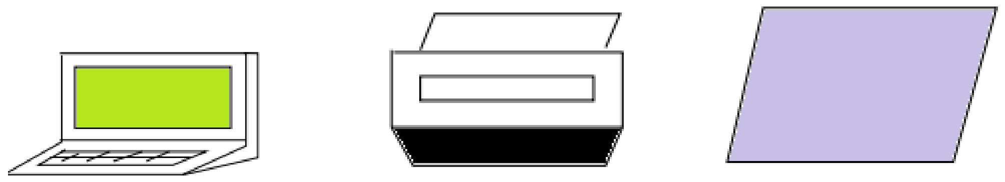
3.2.2 Transformer Icons

Transformers in Figure 3.2 basically change data items and are similar in concept to queries and updates. In the case of VisFlow, they also do a format or representation modification of data items. The icon bank includes three specific icons: *adapter*, *analytics*, and *library* icon.

Adapters perform three types of data transformations - format conversion, attribute generation and data manipulation. When a *library* icon is connected to an *adapter*, a stored and named transformation procedure is expected. Otherwise, a code for transformation within the icon is expected which users must supply. The adapter offers a way to transfer the attribute types in the dataset. String, int, double are three primitive types that VisFlow supports. VisFlow supports direct compilation and execution of users' provided SQL, XQuery, R, and Python codes in the *adapter* icon (section 4.4). It also allows users to write custom code or use the visual query builder (section 5.3.7) to generate a SQL script or XQuery script, or create queries to web services and extract data back. Since deep web databases are also part of the named data resources, they can be used in the same manner as stored ground data resources in queries. However, since deep web data resources are dynamically computed using the BioFlow extract statement in Appendix A, it is treated as a relational *view* and must be defined in the query as such, i.e., as a view table in the from clause of a select query using an extract statement. It is, however, possible to express a query that only involves data extraction from a deep web resource using the *extract* statement. An *adapter* icon can have exactly one incoming arrow from a *data* icon, and at most one incoming arrow from a *library* icon, but it can have any number of outgoing arrows.

An *analytics* icon offers unlimited ways to manipulate the datasets. Some applications require custom programming even beyond the available library functions and all database operations supported by VisFlow back-end data management systems eXist and MySQL. Users are allowed to use stored analytics in the library by connecting an *analytics* icon. Alternately, a user-supplied code in one of the allowed languages can also be used similarly to the *adapter* icon. The *analytics* icon also provides a module that helps users submit the query to web services. Similar to adapters, it automatically batches submission and makes suggestions for users to set up the correct workflow. An *analytics* icon can have any number of incoming arrows and outgoing arrows. An *analytics* icon can be an arbitrary segment of code, an entire workflow or a simple operation. Users are allowed to modify a stored analytics source code in the icon locally to alter its behavior so long it is written in one of the supported languages. Unless stored, the modifications made are transient and do not affect the stored analytics.

The library is a collection of executable functions installed in the web server or any custom code and libraries that can be compiled and executed through Windows command line. As with similar resources in VisFlow, all cataloged library analytics are indexed and enriched. This requires external analytics such as online analysis tools, and executable files can also be used, but they should be cataloged in the library first to be consistent although a direct use is possible. In scientific applications, traditional database queries are often insufficient, and specialized analytics are needed for various analysis. For example, GSEA [134], GWAS [91], and ClustalW [140] type of analyses cannot be done using basic database operations. Most tools also require data to be in specific formats, and for each analysis, there are numerous tools to choose from. These tools are the prime candidates to be described in VisFlow as stored library functions, and these functions can be selected using the *library* icon and used in the *analytics* icons. Technically, all analytics Φ are functions of the form $\Phi X = Y$ where X and Y are sets of data items in one of the supported formats, and X satisfies the format and input requirements of Φ. Finally, data items not in one of the three supported formats can only be processed or analyzed using stored, user-supplied or external analytics manually. This implies that VisFlow is capable of automatically type checking data items needed in analytics and recommend adapters if they are in the supported formats. The *library* icon also has top-k matching algorithms for function recommendation similar to the *data* icon. A *library* icon can have any number of outgoing arrows and no incoming arrows.



(a) Terminal Icon (b) Printer Icon (c) General I/O Icon

Figure 3.3: Visualization Icons

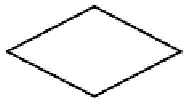
3.2.3 Visualization Icons

This bank in Figure 3.3 has three icons to capture the I/O behavior at any stage of the analysis. The two icons *terminal* and *printer* are obvious: *terminal* icon contains predefined data visualization tools to help users explore the results, while the *printer* icon connects to the local printers and sends the data resource for printing. We want to note here that the visualization is context sensitive and depends on the type of data being visualized and the device used, and the display can adjust if a proper determination of the format can be made. For a proper rendition, the data needs to be well understood, and the modalities of the display need to be appropriately cataloged in the library where an appropriate function of the presentation will be available to facilitate the visualization. The *terminal* and *printer* icon both have any number of incoming arrows but do not have outgoing arrows.

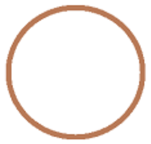
A *general I/O* icon is a parallelogram in which users can describe general input-output needs. When a *general I/O* icon is running, it allows users manipulate the data resources in their local computer by applying custom stored procedures and upload the results back to the server and continue to execute the workflow. A *general I/O* icon has at least one incoming and one outgoing arrow, while the other two icons have no outgoing arrows and are thus *terminal* icons with any number of input arrows.

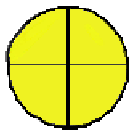
3.2.4 Workflow Icons

In the workflow bank (shown in Figure 3.4), there are five basic icons -*if*, *repeat*, *connect*, *wait on* and *procedure* icon. An *if* icon has one input and at least one outgoing arrows - one “yes” is required and one “no” is optional that indicates the flow of control depending on the success of the test included in the decision icon. All flow or sequences are directed in the direction of the arrow. The `if` conditions are based on attributes’ statistics in the datasets. Attributes’ statistics is defined as a collection of functions that can apply to a list of values. In current version of VisFlow, only maximum value of an array, minimum value of an array, average value of an array, if an array contains an element, and number of rows in the datasets total five functions have been implemented in the if component. We are


(a) If Icon

(b) Repeat Icon


(c) Connect Icon


(d) Wait on Icon

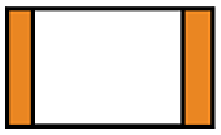

(e) Procedure Icon

Figure 3.4: Workflow Icons

planning to add more functions in if component in future.

A *repeat* icon in VisFlow controls the number of times the workflow executes. It is also based on attributes' statistics in the datasets. It has at least one incoming arrow and two outgoing arrows, one pointing to the repeat icon, the other pointing to the next icon after the loop finishes. Both *if* and *repeat* icons denote the fundamental concept in programming languages and also provide a way to control the branches when we are programming in a visual language.

A *connect* icon helps connect multiple arrows in one icon and serves as an identity function, i.e., it moves data across the icon without any transformation. In other words, a *connect* icon serves no useful purpose other than to help eliminate clutter in the workflow. It can have any number of incoming arrows and usually has one outgoing arrow.

A *wait on* icon provides an explicit way to control the execution order of the workflow. When the execution reached the *wait on* icon, all operations halt at this icon and resume when the processing at the previous icon at the end of each of the incoming arrows has completed. Thus, this icon helps achieve synchronization when needed, i.e., when the input to analytics is expected from two upstream computations or sources. The *wait on* icon has at least two incoming arrows and at least one outgoing arrow.

A *procedure* icon is a place that allows users to introduce nested workflows into the current workflow. A *procedure* icon has any number of incoming arrows and at least one outgoing arrow. When a procedure icon has no incoming arrows, it uses the default datasets as inputs. When it has incoming arrows, the matching data resources are replaced. Users are allowed to assign the data resources generated by *adapter* and *analytics* icons in the nested workflow as output resources. By allowing nested workflows, VisFlow reduces the size of the workflow composition and improves the usabilities of workflow

(a) Edge Icon (b) Select Icon (c) Delete Icon (d) Undelete Icon

Figure 3.5: Editing Icons

components.

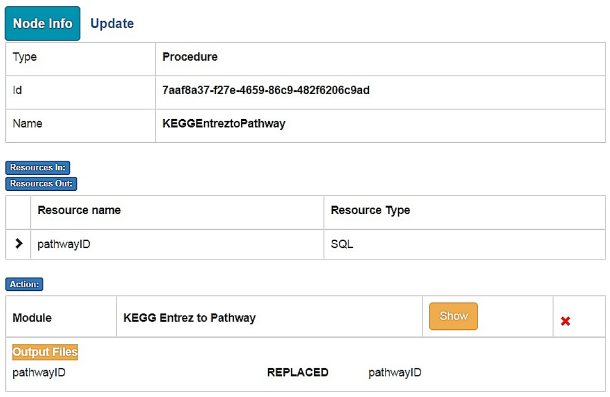
3.2.5 Editing Icons

Finally, the editing icons show in Figure 3.5 - *edge*, *select*, *delete*, and *undelete*, allow editing functions of existing icons. It connects two icons, controlling resources that flow through the workflow. The *select* icon let users choose an icon to be inspected and described. Once selected by clicking on an icon, the node panel on the right opens and displays all the details thus far regarding the icon. Any icon in a workflow can be deleted by selecting the trash icon, and a deleted icon can be recovered by clicking the recover button. We must note that only the recently deleted icon can be recovered and no more than one icon can be recovered as no history is maintained. This is included to help recover accidental deletes, not a whole scale recovery using version management and history.

3.3 VisFlow Data Flow

When users construct the workflow, data resources and libraries transfer from one icon to another icon. VisFlow Constraints are applied when these events occur, in order to validate the workflow on each step. Table 3.1 shows the constraints among all icons in VisFlow language. These rules are generated based on the semantic meanings behind these icons. We also need a mechanism to show the node description and provide possible configuration information for users after different resources or functions have been passed into each icon as defined in the previous section. To achieve these constraints and show the possible configuration and install different parts in VisFlow, we define three matching functions: $\langle \Upsilon, \Theta, \Xi \rangle$ to help us show and validate the correct information, which is critical in VisFlow. Υ is the node/edge constraint function, Θ is the node mapping function, and Ξ is the edge mapping function.

The starting point of a workflow in VisFlow is always a *library* and *data* icon, and this is where the library functions and data resources are imported into the workflow system. Then users are allowed to use the meaningful icons to construct the workflow and apply functions from libraries to datasets or submit queries to web services. Action, function, and resource fields are the key objects in VisFlow, and every icon is based on these three pieces of information. The workflow is based on a list of configured



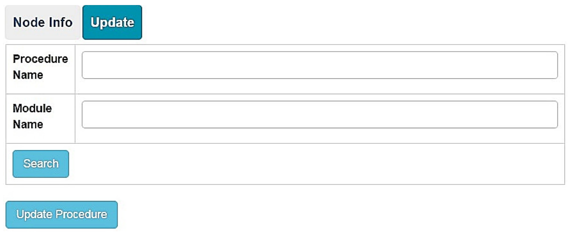
Type	Procedure
Id	7aaf8a37-f27e-4659-86c9-482f6206c9ad
Name	KEGGEntreztoPathway

	Resource name	Resource Type
›	pathwayID	SQL

Module	KEGG Entrez to Pathway	Show	✗

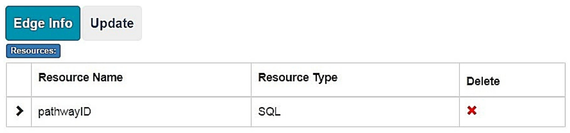
pathwayID REPLACED pathwayID

(a) Node Info



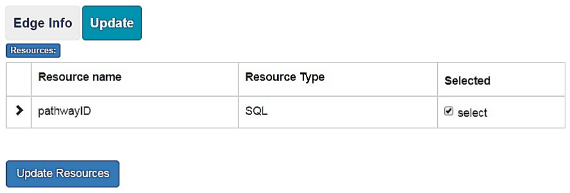
Procedure Name	
Module Name	

(b) Node Update



	Resource Name	Resource Type	Delete
›	pathwayID	SQL	✗

(c) Edge Info



	Resource name	Resource Type	Selected
›	pathwayID	SQL	☑ select

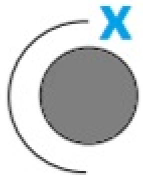
(d) Edge Update

Figure 3.6: Data Flow Information Generation

action icons. $\Upsilon(n, n')$ validates the connect comparability of two nodes when users attempt to connect and returns an error message when two nodes are not allowed to connect. For example, if a *data* icon is planned to connect to another *data* icon, or itself, it will generate error messages for the user.

When a list of resources is passed via an edge and reaches a node, Θ is called to generate the node information, description, proper input fields for users to type in information and show current configurations of the node as shown in Figure 3.6a. It also generates the update information as shown in Figure 3.6b. For example, if we only have resources as functions passed in, Θ generates a form for submitting arguments for this function. Or if we only have resources as remote web services, Θ generates a remote web service submit the form. When a list of resources/functions are passed to the edges, Ξ generates the description information for showing. For example, Figure 3.6c shows the edge information with only one data resource. Ξ also generates the update information and proper input fields for different actions (as shown in Figure 3.6d).

With these mapping and constraint functions, the syntax of VisFlow is complete, and it provides a paradigm where every resource and function is treated as a plugin, and these plugins are passed and installed in each abstract icon defined in VisFlow. This kind of design pattern that will help users compose their workflow quickly by supporting flexible, reusable resources and functions. Once users need to run something, users first use a *data* icon or *library* icon to import data resources or library


(a) Adapter Type Error Icon

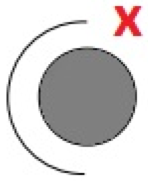

(b) Adapter Syntax Error Icon

Figure 3.7: Examples of Error Messages

functions into the workflow and assign it to proper components in the workflow and configure the nodes, and then everything is done.

3.4 VisFlow Validation

Two types of error messages are defined in VisFlow. One is named type error, and the other is called syntax error. The type error indicates that VisFlow can't match the required attribute from the imported libraries or resources. The syntax error suggests that a mistake happened when users constructed the workflow. We have restrictions on the relationship between these icons. For example, an *adapter* can't be an ending node; it requires an outgoing arrow. Otherwise, this workflow is not valid, and it could not be executed. Figure 3.7 shows the sample adapter icon in the three states. A blue error will pop up on the right top side if the icon has type error; a red error will pop up on the location if the icon has syntax errors.

In practice, Υ function will be called to check the syntax whenever a new node is added in a workflow, or users try to connect two existing nodes, or users delete edges or nodes in the workflow. The Θ and Ξ functions are triggered when users update, remove, or change resources in a workflow, or functions are updated, deleted, or modified to check for a type error. If one of these mistakes occurs, the icon is appropriately changed. If an icon has both a type error and a syntax error, only a syntax error is shown. In VisFlow, fixing the syntax error is the first step to generate a working workflow. Without fixing the syntax error, just correcting the type error is meaningless in the context of the workflow.

3.5 VisFlow Presentation

When implementing VisFlow, the workflow node is in the form of $\langle type, module, resourcesIn,$ $resourceOut, action\rangle$, where *type* is the node type from fifteen icons, *module* is used for *procedure* icon, $resourcesIn$ and $resourcesOut$ are lists of resources from previous nodes and those resources generated by this node, *action* is an object of $\langle type, condition, tBranch, fBranch, repeatprocess, targetRes,$

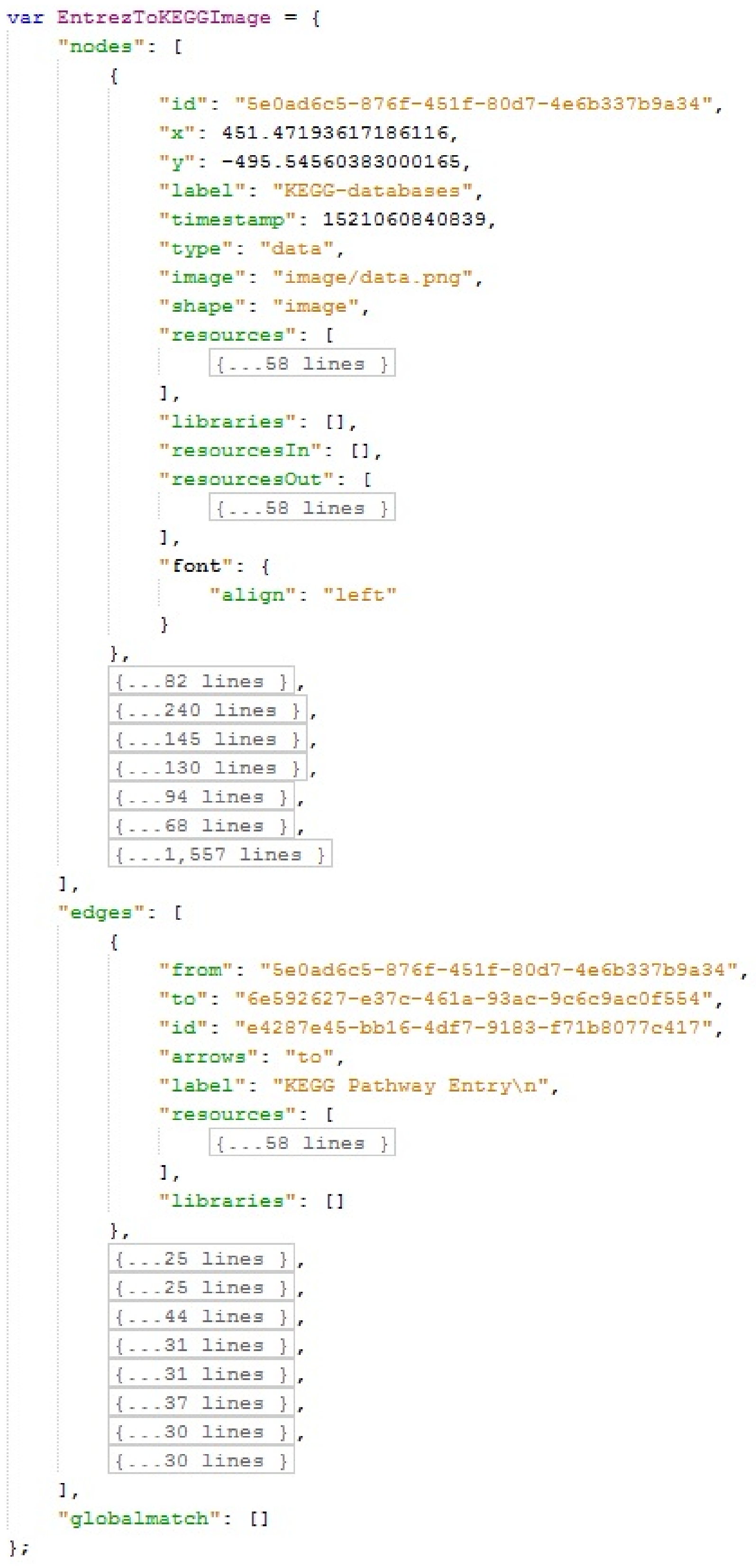
Icon Name	Number of Incoming Arrows	Number of Outgoing Arrows	Can't connected to	Can't connected from
Data Icon	0	1+	Data, Library	All of them
Combine Icon	2+	1+	Data, Library	Library, Terminal, Printer
Fusion Icon	2+	1+	Data, Library	Library, Terminal, Printer
Adapter Icon	1+	1+	Data, Library	Terminal, Printer
Analytics Icon	1+	1+	Data, Library	Terminal, Printer
Library Icon	0	1+	Data, Library, Terminal, Printer	All of them
Terminal Icon	1+	0	Don't Allow	Library, Terminal, Printer
Printer Icon	1+	0	Don't Allow	Library, Terminal, Printer
General I/O Icon	1+	1+	Data, Library	Library, Terminal, Printer
if Icon	1	1-2	Data, Library	Library, Library, Terminal, Printer
Repeat Icon	2	1	Data, Library	Library, Terminal, Printer
Connect Icon	1+	1	Data, Library	Library, Terminal, Printer
Wait On Icon	2+	1+	Data, Library	Library, Terminal, Printer
Procedure Icon	0+	1+	Data, Library	Library, Terminal, Printer

Table 3.1: VisFlow Icon Constraints

$code, funcs, inputs, resources\rangle$. Action type is used to a denote web service call, custom code or function calls. The *condition*, $tBranch$, $fBranch$, and *repeatprocess* are used for *if* and *repeat* icons for their conditions and outgoing icons for each branch. $TargetRes$ is a form of $\langle outputAttrs, url\rangle$ for a selected web service resource that contains the output attributes and URL. The code field stores custom code. The last three fields of an action are $funcs$, *inputs* and $resources$. These fields are used to store the action applied imported functions, required inputs values and resources for transformation. The workflow edge is in the form $\langle from, to, resources, libraries\rangle$ where $from$ points to the previous node, *to* points to the current node, $resources$ are a list of passed resources and *libraries* are a list of functions passed by a *library* icon.

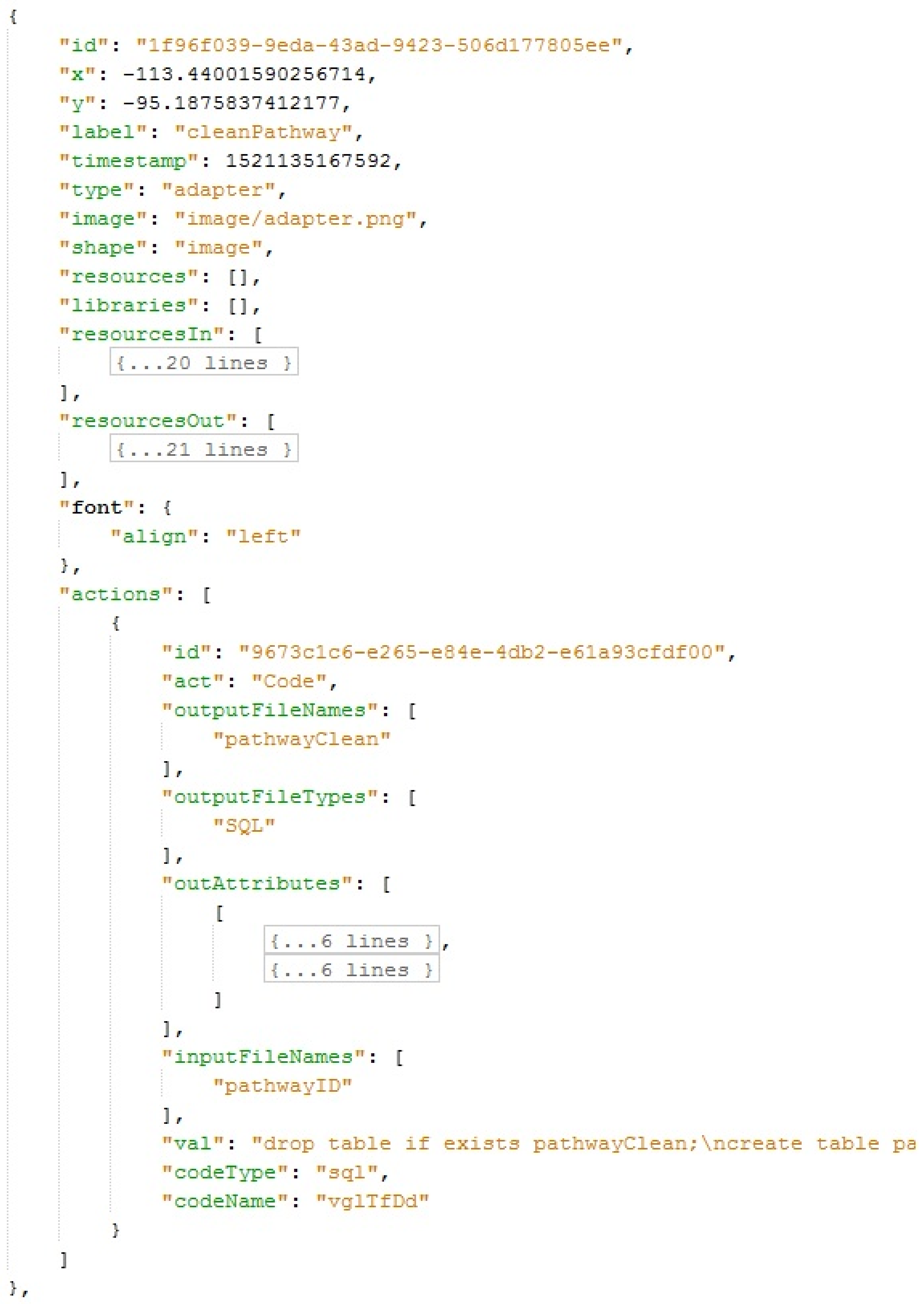
The whole VisFlow workflow graph is stored as a JSON object file. JSON is a lightweight data-interchange format which is widely supported in a variety of languages. It is easy to manipulate and store in the web browser, or on the web services. We save the node and edge information in a prominent JSON object.

Figure 3.8 shows the JSON object as the workflow of the Entrez to KEGG Image example in section 4.1, where it contains eight nodes and nine edges. The whole JSON object has two attributes named nodes and edges, which are lists of node objects and edge objects. The expanded node is the KEGG-database *data* icon. The edge is from this *data* icon to the retrieveDescription *adapter* icon. Figure 3.9 shows an example JSON node named cleanPathway in the same example after the users' configuration. The attributes in these fields are more than the language we defined, and this is because we need extra information to store visualization on the user interface canvas and other systems auto-generated information to speed up the composition process.



```
var EntrezToKEGGImage = {
    "nodes": [
        {
            "id": "5e0ad6c5-876f-451f-80d7-4e6b337b9a34",
            "x": 451.47193617186116,
            "y": -495.54560383000165,
            "label": "KEGG-databases",
            "timestamp": 1521060840839,
            "type": "data",
            "image": "image/data.png",
            "shape": "image",
            "resources": [
                {...58 lines }
            ],
            "libraries": [],
            "resourcesIn": [],
            "resourcesOut": [
                {...58 lines }
            ],
            "font": {
                "align": "left"
            }
        },
        {...82 lines },
        {...240 lines },
        {...145 lines },
        {...130 lines },
        {...94 lines },
        {...68 lines },
        {...1,557 lines }
    ],
    "edges": [
        {
            "from": "5e0ad6c5-876f-451f-80d7-4e6b337b9a34",
            "to": "6e592627-e37c-461a-93ac-9c6c9ac0f554",
            "id": "e4287e45-bb16-4df7-9183-f71b8077c417",
            "arrows": "to",
            "label": "KEGG Pathway Entry\n",
            "resources": [
                {...58 lines }
            ],
            "libraries": []
        },
        {...25 lines },
        {...25 lines },
        {...44 lines },
        {...31 lines },
        {...31 lines },
        {...37 lines },
        {...30 lines },
        {...30 lines }
    ],
    "globalmatch": []
};
```

Figure 3.8: VisFlow JSON Presentation



```
{
    "id": "1f96f039-9eda-43ad-9423-506d177805ee",
    "x": -113.44001590256714,
    "y": -95.1875837412177,
    "label": "cleanPathway",
    "timestamp": 1521135167592,
    "type": "adapter",
    "image": "image/adapter.png",
    "shape": "image",
    "resources": [],
    "libraries": [],
    "resourcesIn": [
        {...20 lines }
    ],
    "resourcesOut": [
        {...21 lines }
    ],
    "font": {
        "align": "left"
    },
    "actions": [
        {
            "id": "9673c1c6-e265-e84e-4db2-e61a93cfdf00",
            "act": "Code",
            "outputFileNames": [
                "pathwayClean"
            ],
            "outputFileTypes": [
                "SQL"
            ],
            "outAttributes": [
                [
                    {...6 lines },
                    {...6 lines }
                ]
            ],
            "inputFileNames": [
                "pathwayID"
            ],
            "val": "drop table if exists pathwayClean;\ncreate table pa
            "codeType": "sql",
            "codeName": "vglTfDd"
        }
    ]
},
```

Figure 3.9: Node 3: CleanPathway Configuration in JSON Format

Chapter 4: VisFlow Execution

After the workflow has been defined, it needs to execute on the server. There are three approaches implemented to transfer workflow icons into computer understandable code. First, transfer it to another workflow system by accommodating the workflow paradigm. This philosophy of extending an existing language is not new [56]. Wings [61] is an similar workflow system that translated their graph to Pegasus [41] and executed. The second way is to map the workflow directly that users does to executable code. HyperFlow [14] is in this catalog which provides users direct benefit of workflow and Javascript language.

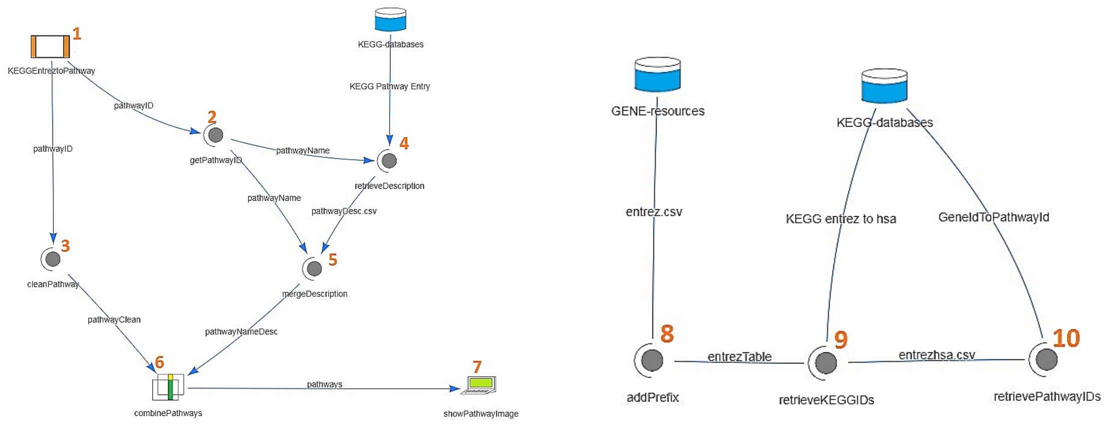
The last method is to translate it into another language, for example, a declarative language. The main advantage of doing this is that users do not have to learn a new language, and the developers do not have to implement the interpreter for this language from scratch. From this standpoint, declarative languages such as SQL and Datalog, and their domain-specific extensions, have been identified as significant artifacts for application development. Though more limited, these languages embody higher abstractions compared to procedural languages such as C and Java to ease querying by end users. However, though simpler, declarative languages can still be intimidating to master before they can be used to develop the users' applications. Most workflow systems are following this procedure to execute the workflow that directly translated visual languages to script language and then execute it.

The third approach, which is the way VisFlow followed, is to translate it into another language named BioFlow [74], which executes it. For more information about BioFlow, please refer to Appendix A. This method is not only better for abstracting the workflow and providing more meaningful icons, but also reduces the size of code. It is also easy to maintain and reuse the code.

4.1 An Illustrative Example Using VisFlow

For illustrating the features of VisFlow, we selected a relatively complex workflow example named entreztoKeggImage[1] from the myExperiment repository as a computational pipeline in Bioinformatics domain and generated the workflow using VisFlow as shown in Figure 4.1. This workflow first submits a list of Entrez Gene IDs through the KEGG database and gets the corresponding KEGG IDs back, then sends these KEGG IDs to the KEGG web service again to retrieve the pathways that involve each gene and displays the Pathway diagram image with the description. Node 1 is a *procedure* icon that allows users to import stored workflows. Please refer to section 6.2 for how to construct this example in details. In this example, node 1 contains the nested workflow as shown in Figure 4.1b where it includes two *data* icons and three *adapter* icons.

[1]`https://www.myexperiment.org/workflows/4453.html`



(a) Workflow graph. (b) Entrez IDs to KEGG Images in VisFlow.

Figure 4.1: Entrez IDs Workflows

Node 8 in figure 4.1b is an adapter node that adds the prefix to Entrez IDs using a SQL script generated by the visual query builder. Node 9 and node 10 are two web services calls that submit to KEGG databases. Node 9 retrieves a list of KEGG IDs, while node 10 retrieves a list of pathways based on the KEGG IDs. Figure 4.1a in Node 1 imported the output file named pathwayID from the sub workflow in Figure 4.1b. This file is passed to node 2 and node 3 to extract HSA corresponding Pathway names and remove pre-string 'path:,' respectively. These two nodes also use the visual query builder to generate the SQL queries. Node 4 is another web service call to get the Pathway descriptions. Node 5 merged the input Pathway IDs with the returned descriptions. Node 6 is a *combine* icon that automatically merges two tables based on their key pairs to generate the final result. The *combine* icon has a generate key pairs button to suggest the possible key pairs in two tables, and users are allowed to overwrite the recommended key pairs in the returned table. Node 7 is the *terminal* icon that generates vivid results by offering predefined printing rules. In this example, pathways are viewed as an image and VisFlow has built-in functions go to the KEGG database to retrieve the images back to local and show it to users.

4.2 Transformation from VisFlow to BioFlow

VisFlow workflows potentially contain a visual representation of BioFlow statements or a combination of imperative program codes in R and Python both in executable form or in source code form, and SQL and XQuery to manipulate VisFlow database information. VisFlow icons, which are stored as JSON files are automatically translated to Java objects using the Google GSON library and mapped to BioFlow statements, then BioFlow statements are compiled down to Java code and executed. Code segments written directly as operations of VisFlow are compiled or executed using the command line in Windows 7 operating system which may include Python scripts, R Scripts, Windows bash files, etc., or MySQL server for SQL script, or eXists-db for XQuery script. In user-supplied codes and libraries, HTTP, and FTP protocol based web services are supported, including RESTful based web services.

The idea of transformation from VisFlow to BioFlow is very straightforward. For each icon, it is a process statement if the icon is *combine*, *fusion*, *adapter*, and *analytics*. These statements are connected using perform statement together to decide the execution order. Algorithm 1 lists the steps that show how to convert a VisFlow Graph to a BioFlow script in detail. This algorithm contains two steps. The first step generates the execution control statements, which includes the parallel execution, pause and wait on statements. The second phase creates BioFlow process statements for each node in the graph. The input arguments are the VisFlow graph, a list of possible schema matchers, list of possible wrappers, list of identifiers and list of filters. In line 2, it generates the dependence graph which stores the dependencies among nodes in pairs named $from$ and *to* in this graph. Lines 4 to 5, loop through all the nodes and get its parents and successors from the two maps and generate the PERFORM PARALLEL statements. After generating the PERFORM PARALLEL AFTER statements, we need to take care of the workflow nodes that allow changing the execution of the workflow order. From line 6 to 23, the algorithm checks whether a node is a type of if, repeat, general I/O or wait on. If it is one of these three, it generates their corresponding statements, and the first step is done after all these node statements have been added. The next step is generating the process for each node using Algorithm 2. We loop through all the nodes and for each different node type generate the corresponding BioFlow process script. This algorithm has three different conditions related to the *procedure* icon, *combine* and *fusion* icon, and lastly *adapter* and *analytics* icon. From line 6 to line 8, if the node is a procedure node, the generateBioFlow function in Algorithm 1 is called again to generate the script for the nested workflow. From line 9 to line 13, the combine and fusion statements are generated. Lines 14 to 29 deal with the *adapter* and *analytics* icon, which supports web service call, transformation, executable function, and custom coding.

VisFlow icons have one to one mapping to BioFlow statements, VisFlow interface has provided a Get BioFlow script button (shown in section 5.1) to automatically generate the matching script for current workflow on the canvas. The list of all BioFlow scripts of all the examples in this dissertation is present in Appendix A.

Listing 4.1 is the BioFlow script transferred from example 4.1 for each node process. The transformer algorithm assigns random file names after `call` statement (shown in line 2, 14 and 23 in this example). These are used to reference the script that the system will execute (Please refer to Section 4.4 for more details). Process in line 1 is node 8 in Figure 4.1b. Processes in line 3 and line 8 are node 9 and node 10 for KEGG web service calls, respectively. These three processes are from the nested workflow node 1 in Figure 4.1a. From node 2 through node 6 they match the processes one by one in order in the BioFlow script.

Algorithm 1: GenBioFlow Function

```
input  :
g: VisFlow Graph
M: a list of matchers
Ω: a list of wrappers
K: a list of key identifiers
Φ: a list of fillers
output: A script of BioFlow s
Function genBioFlow (g, M, Ω, K, Φ)
    /* Dependence graph                                          */
    depGraph = generateDependenceGraph (g)
    s ← {}
    foreach {fromNode, toNode} ∈ depGraph do
        s ← "PERFORM PARALLEL " + toNode + " AFTER " + fromNode;
    foreach node ∈ g do
        switch node.type do
            case IF do
                θ ← node.action.condition
                tBranch ← node.action.tBranch
                fBranch ← node.action.fBranch
                s ← "IF θ THEN tBranch
                ELSE fBranch"
            case Repeat do
                θ ← node.action.condition
                repeatProcess ← repeated processes from graph
                s ← "REPEAT processes UNTIL θ"
            case General I/O do
                r ← node.action.resources
                s ← "PAUSE r"
            case Wait on Icon do
                parents ← node's parents from graph
                s ← "WAIT ON parents"
    s ← genBioFlowProcess(g, M, Ω, K, Φ)
    return s
```

Algorithm 2: GenBioFlowProcess Function

```
input :
$g$: VisFlow Graph
$M$: a list of matchers
$\Omega$: a list of wrappers
$K$: a list of key identifiers
$\Phi$: a list of fillers
output: s
1  Function genBioFlowProcess ($g, M, \Omega, K, \Phi$)
2      s ← {}
3      foreach $node \in g$ do
4          s ← "PROCESS " + $node.label$
5          switch $node.type$ do
6              case Procedure do
7                  $module \leftarrow node.module$
8                  s ← genBioFlow($module, M, \Omega, K, \Phi$)
9              case Combine or Fusion do
10                 $r, s \leftarrow$ resources in $node.resourcesIn$
11                 $\mu \leftarrow m(M, node.resourcesIn)$
12                 $\kappa \leftarrow k(K, node.resourcesIn)$
13                 s ← "[$COMBINE||FUSION$] $r$, $s$ USING MATCHER $\mu$ IDENTIFIER $\kappa$"
14             case Adapter or Analytics do
15                 switch $node.action.type$ do
16                     case Web Access do
17                         $\alpha \leftarrow node.action.targetRes.outputAttrs$
18                         $url \leftarrow node.action.targetRes.url$
19                         $r \leftarrow node.action.inputAttrs$
20                         $\mu \leftarrow m(M, node.action.targetRes)$
21                         $\omega \leftarrow o(\Omega, node.action.targetRes)$
22                         $\phi \leftarrow p(\Phi, node.action.targetRes)$
23                         s ← " EXTRACT $\alpha$ USING MATCHER $\mu$ WRAPPER $\omega$ FILLER $\phi$
                             FROM $url$ SUBMIT $r$
24                     case Code or Function do
25                         $f \leftarrow node.action.[funcs||code]$
26                         $r \leftarrow node.action.resources$
27                         s ← " CALL
28                         FUNCTION $f$
29                         WITH $r$
30     return s
```

```
PROCESS add-prefix {
CALL LYrQLtb WITH (entrez); }
PROCESS retrieveKEGGIDs {
EXTRACT ko,hsa
USING MATCHER S-Match WRAPPER FastWrap
FROM  http://rest.kegg.jp/conv/hsa/
SUBMIT entrez (gene); }
PROCESS retrievePathwayIDs {
EXTRACT hsa,pathway
USING MATCHER S-Match WRAPPER FastWrap
FROM http://rest.kegg.jp/link/pathway/
SUBMIT entrezTable (hsa); }
PROCESS cleanPathway {
CALL vglTfDd WITH (pathwayID); }
PROCESS getPathwayID {
CALL rEHJwIN WITH (pathwayID); }
PROCESS retrieveDescription {
EXTRACT pathway,description
USING MATCHER S-Match WRAPPER FastWrap
FROM  http://rest.kegg.jp/get/
SUBMIT pathwayName (pathway); }
PROCESS mergeDescription {
CALL XDPhGCO WITH (pathwayName,pathwayDesc.csv); }
PROCESS combinePathways {
COMBINE pathwayClean, pathwayNameDesc
USING MATCHER S-Match IDENTIFIER Gordian; }

```

Listing 4.1: BioFlow Processes for the Entrez Workflow in Figure 4.1.

4.3 VisFlow Orchestration

```
PERFORM retrieveKEGGIDs AFTER add-prefix
PERFORM retrievePathwayIDs AFTER retrieveKEGGIDs
PERFORM getPathwayID, cleanPathway AFTER retrievePathwayIDs
PERFORM mergeDescription AFTER retrieveDescription
PERFORM combinePathways AFTER mergeDescription

```

Listing 4.2: BioFlow Executions

In the scientific workflow management system domain, workflow execution or orchestration are allowed to be run continuously or parallel. For parallel execution in workflow systems, there are two types: data-driven based and event-driven based. Data-driven based algorithms focus on running analytics on data resources in parallel within a component, while in event-driven algorithms, the next component is executed if and only if all the inputs from previous components are finished. The execution of VisFlow is based on an event-driven algorithm. Unlike most of the workflow systems that require users to understand the parallelism in detail and then configure the workflow as needed, VisFlow is fully automatically executed that, if possible, in parallel.

The idea behind this type of execution is we are trying to take advantage of parallel execution to speed up the running time of execution. If a process has nothing to do with other actions, we can execute them at the same time without a break the consistency in the workflow. From line 1 to 5 in Listing 4.2 shows one of many ways to execute the whole workflow graphs, it depends on the execution time for each node. This is a dilemma in that we can't get the execution time for every node unless we finished execution of the whole workflow.

In order to optimize the execution order of the workflow to reduce the total time, we propose a dynamic parsing algorithm to parse and execute the workflow at runtime. Topological sorting in Graph theory inspires the whole algorithm. It first identifies these nodes which can run at the first time, then put them into the thread pool for parallel execution. When one of the threads finishes, it examines the whole graph and gets a list of ready nodes and throws them into the thread pool. In this type of mechanism, the order of execution is decided at runtime. Although the BioFlow script generated in Listing 4.1 looks like it is executed sequentially, it's not.

Based on this idea, Algorithm 3 shows how VisFlow parallel executes a BioFlow Script. The first step of this algorithm calls the generateGraph() function at line 4 to generate the process graph using the parallel statements, then we calculate the indegrees for each node (line 5 to 6), and add all nodes with indegrees of zero (line 7 to 9) to our queue. From line 10 to 12, we add the BioFlow scripts for nodes in the queue to execute. From line 13 to 23, we wait until one of the threads is finished and get the last finished node information. There are three exceptions we need consider. If the previously executed node is if, wait on or repeat, we call our delete function to delete the nodes that are based on the conditions in the node and then update indegrees in the graph. Then we update the indegree array and add new nodes to our queue. Next is add the first node in the queue to threadPool and start the execution.

In the Entrez IDs to KEGG image example, node 8, node 9 and node 10 are executed one by one. After that, node 2 and node 3 execute simultaneously. After node 3 is done, node 4 starts running. If after node 5 finishes node 3 is still running, node 6 has to wait until node 3 is finished to start itself. Another situation is that node 3 finished before node 5. Node 6 can start right after node 5 finishes. The actual execution order is based on each node's execution time and their topological relations.

Algorithm 3: Parallel Execution

```
   input :
   script: BioFlow Script
   /* Initial Queue                                              */
1  initial queue
   /* Ready List                                                 */
2  initial threadPool
   /* indegree array for each node                               */
3  initial indegree
   /* Generate Graph via parallel statements                     */
4  g ← generateGraph (script)
5  foreach edge ∈ g do
6      indegree[edge.from] ← indegree[edge.from] + 1
7  foreach node ∈ indegree do
8      if indegree[node] == 0 then
9          queue ← node
10 while queue is not empty and threadPool is not full do
11     node ← first node in queue
12     threadPool.run(node.BioProcess)
13 while queue is not empty do
14     Wait until threadPool has space for running
15     finishedNode ← threadPool.getFinishedNode()
16     if finishedNode ∈ {IF, Waiton, Repeat} then
           /* delete indegree, g as needed                       */
17         delete(node, indegree, g)
18     foreach v ∈ g where finishedNode goes to do
19         indegree[v] ← indegree[v] - 1
20         if indegree[node] == 0 then
21             queue ← node
22     node ← first node in queue
23     threadPool.run(node.BioProcess)
```

4.4 BioFlow Execution

In Algorithm 3, the threadPool run function is the BioFlow engine that translates the BioFlow `process` statement to Java code. The BioFlow script to Java Programming language mapping is still a one to one mapping. Algorithm 4 shows the detailed steps. In the context of execution, only combine, fusion, extract, and call statement are passed in. If it is a combine statement, the keyPairs are calculated by applying κ function, then μ function is used to determined the matched keys, the last step is apply `union` function to merge to data resources together. fusion has the same steps except the last step it applies `join` function to the data resources. Lines 10 to 13 shows how to execute the extract statement. The parameters are generated by applying μ function to the input resource and the form filling function

Algorithm 4: BioFlow Execution

```
input :
script: BioFlow Process
1  switch script do
2      case Combine do
3          $keyPairs \leftarrow \kappa(r, s)$
4          $matchedKeys \leftarrow \mu(keyPairs)$
5          $union(r, s, matchedKeys)$
6      case Fusion do
7          $keyPairs \leftarrow \kappa(r, s)$
8          $matchedKeys \leftarrow \mu(keyPairs)$
9          $join(r, s, matchedKeys)$
10     case Extract do
11         $parameters \leftarrow \phi(\mu(r))$
12         $data \leftarrow submit(url, parameters)$
13         $\alpha \leftarrow \omega(data)$
14     case Call do
15         switch f.type do
16             case Windows Bash do
17                 $bash(f(r))$
18             case Python do
19                 $bash(pythonf(r))$
20             case R do
21                 $bash(Rscriptf(r))$
22             case SQL do
23                 $bash(mysql < f(r))$
24             case xQuery do
25                 $xQueryClient(f(r))$
```

ϕ. *submit* function is called to submit the query to the web service and retrieve the data back. The wrapper function ω extracts the results from data. Lines 14 to 25 shows how to execute the call statement. All the scripts are executed by windows execution command line except the XQuery script. For XQuery scripts, they are submitted to the eXist database endpoint in the server for execution using the *xQueryClient*.

Listing 4.3 shows all the depended resources that example 4.1 required for execute the BioFlow statements. It contains all the necessary information to execute the Jave code. For example, the actual resource code for executing a script, or the web service information for generating or submitting queries.

For example, node 9 which matches the code in Listing 4.1 line 8 to 12 uses the BioFlow extract statement. The first step is to apply the S-Match algorithm to our data resource entrez, which is a relational table to extract the attribute column gene as a list. Then apply our *submit* function to generate a list of queries and submit through `http://rest.kegg.jp/conv/hsa`. The BioFlow Execution engine generates the corresponding URLs and using the matching HTTP client to submit the queries based on the resource type and method information that stored in the depended resources. For each query, the result is analyzed by the Fastwrap algorithm to extract the results which are appended one by one to form the final results.

```
{codeName: LYrQLtb
codeType: sql
code: drop table if exists entrezTable;
create table entrezTable as
select  (concat('ncbi-geneid:', entrez.entrez)) gene
from  entrez entrez;}
{codeName: vglTfDd
codeType: sql
code: drop table if exists pathwayClean;
create table pathwayClean as
select   pathwayID.hsa ,  (substring(pathwayID.pathway, 6, 8)) pathway
from   pathwayID;}
{codeName: rEHJwIN
codeType: sql
code: drop table if exists pathwayName;
create table pathwayName as
select  substring(pathwayID.pathway, 6, 8) pathway
from  pathwayID pathwayID
group by  pathwayID.hsa;}
{codeName: XDPhGCO
codeType: sql
code: drop table if exists pathwayNameDesc;
create table pathwayNameDesc as
select   pathwayDesc.description ,  pathwayName.pathway
from   pathwayDesc,  pathwayName;}
{resource type:CSV
resource name:entrez}
{resource type:CSV
resource name:pathwayDesc}
{resource type:REST
resource name:KEGG Pathway Entry
method:REST
url:http://rest.kegg.jp/get/{pathway} }
{resource type:CSV
resource name:entrezhsa}
{resource type:REST
resource name:KEGG entrez to hsa
method:REST
url:http://rest.kegg.jp/conv/hsa/{gene} }
{resource type:REST
resource name:GeneIdToPathwayId
method:REST
url:http://rest.kegg.jp/link/pathway/{hsa} }

```

Listing 4.3: Resources for the Entrez Workflow

Chapter 5: VisFlow Workflow Management System

5.1 Visual User Interface

One of the most powerful features of VisFlow is its visual programming interface in which users are able to express computational needs declaratively using visual icons shown in Figure 5.1. The interface consists of a panel of icons with predefined meanings (shown inside the red ellipse on the left), a set of tabs for resource management at the top (shown within the blue box), and a set of workflow execution and management buttons (shown inside the mustard ellipse) just below the system tabs. The left panel includes five classes of icons - from top to bottom, data (dark green bank), transformer (coral bank), visualization (dark cyan bank), workflow (sienna bank), and editing (teal bank) icons. In the VisFlow interface, the exact name or icon identity can be seen in a tooltip by pointing the mouse on any of the icons in any of the banks. Users choose icons from the icon banks on the left and drop them on the canvas, shown within the middle frame, by clicking anywhere. Two icons can then be connected by selecting the arrow icon and then using a drag operation between the two icons. The arrow is always pointed to the icon to which the connection is made (the second icon). By deselecting the *select* icon, users can click and drag objects on the canvas anywhere during an editing session to re-orient the positions of the objects in a workflow. The zoom in and out buttons can be used to magnify or shrink an entire canvas. The roller button of the mouse can also be used to perform these two operations. If the objects in any of the panels, the canvas, the information panel or the results panel become too large to fit within the respective boundaries, scroll bars appear to bring the objects into full view.

The system menus include Examples, Help, About and sign in with their common functionalities. In VisFlow, users don't need to sign in to play with the interface. However, if users register resources in this system by creating accounts, the VisFlow system creates private folders for each user. All the data resources and libraries users register after they sign in are only visible by current account, and others do not have the permission to view and manipulate the folders. The interface also comes with four resource management tabs - Query, Results, Register, and Resources. The query tab has five subordinate options - Run, Clear, Import, Export and Get BioFlow script workflow buttons for individual workflows. Using run, an active workflow on the canvas can be executed while using clear, it can be removed from its active status, and a new workflow can be loaded or developed. Existing workflows can be loaded into the canvas using the import button and made active, and an active workflow can be saved using the export

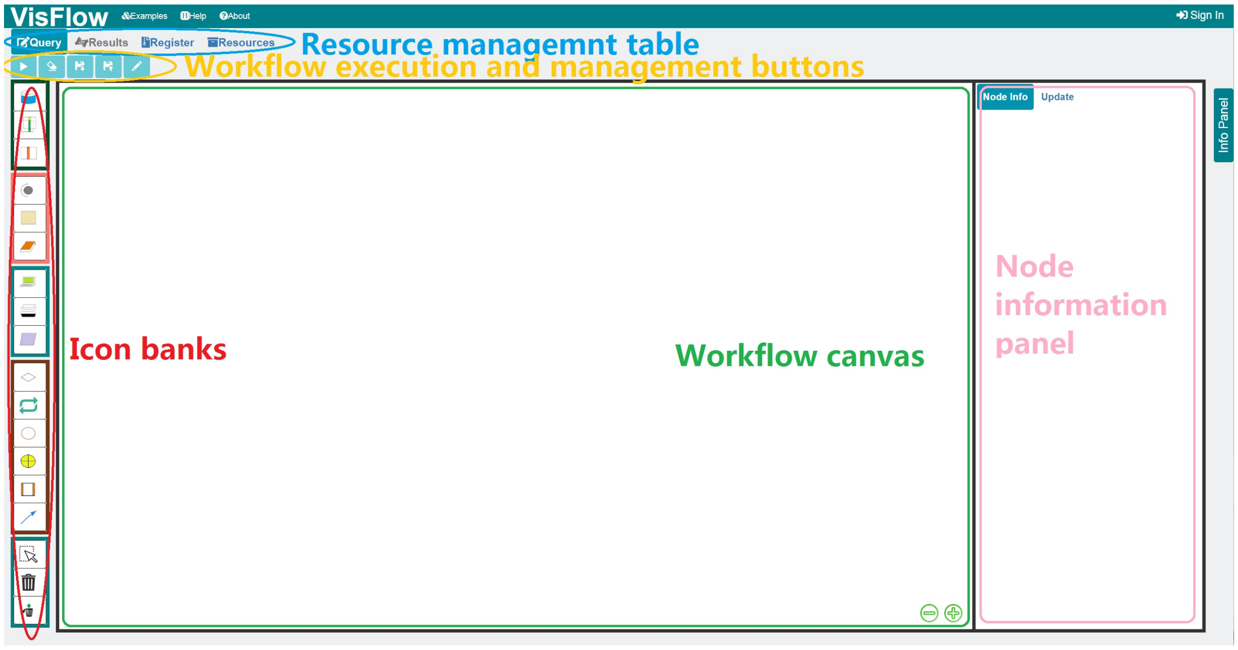
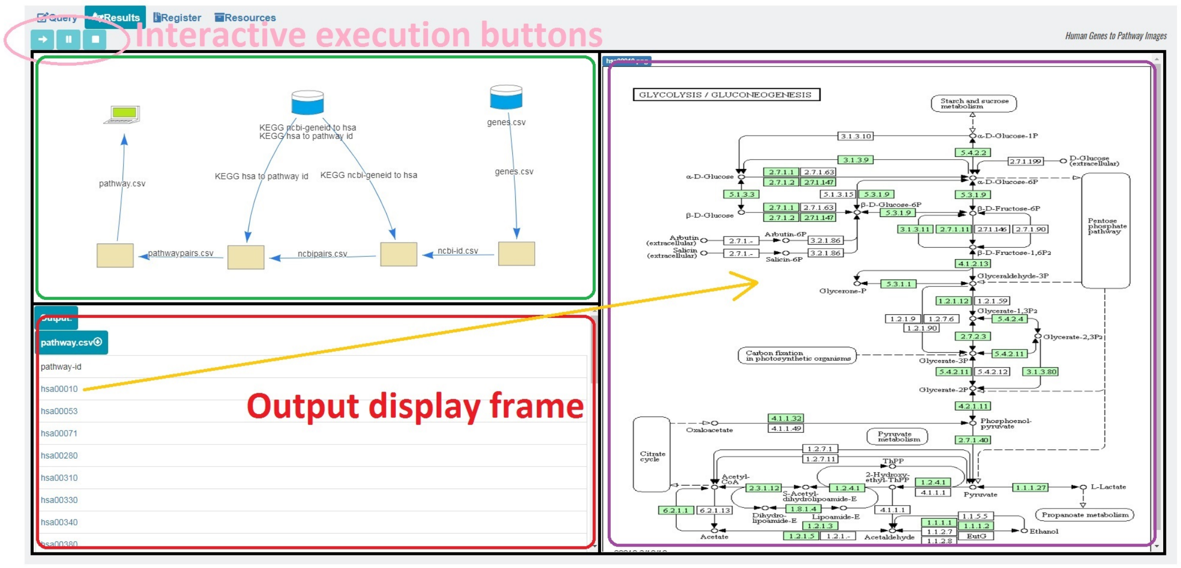


Figure 5.1: VisFlow User Interface

button. The import button provides two ways of saving a working workflow - save to local computer as workflow or save to VisFlow server as a module and reuse it in future workflow composition via a *procedure* icon. The last button is the Get BioFlow script button. When this button is clicked, the BioFlow script generated by the system is popped up on a new window to show matching BioFlow script with the depended resources for current workflow on the canvas. Once run is clicked, and the execution begins in an interpretive mode, the interface switches to the mode shown in Figure 5.2 with three panels and all compilations and executions take place in real time. The green panel shows the active workflow, the magenta panel displays execution information for every node and serves as a viewing window of complex objects in the results such as a picture or a web page, and the red panel displays the computed results. After the execution has finished, the magenta panel is used to display extra information from the results, e.g., images, maps, visualizations. The stages of execution are shown by highlighting the active icon as the control steps through the nodes in the workflow graph. The exact operation being performed within the icon is displayed so that users can follow the execution. Users are optionally able to halt, continue after a halt, inspector stop execution at any of the icons using the interactive execution buttons in the pink ellipse shown in Figure 5.2 to debug a workflow.

The next tab is a registry page that allows users to register new resources or libraries in Figure 5.3. In the resource form, users need to provide the resource type, either CSV, XML, HTTP, REST or Other, which is a flat table, XML document, web service based on HTTP Protocol, web service based



Output:

pathway.csv

pathway-id
hsa00010
hsa00053
hsa00071
hsa00280
hsa00310
hsa00330
hsa00340
hsa00380

Figure 5.2: VisFlow Result Page

on REST protocol, or other types of documents. For each resource uploaded to VisFlow, an aggregate name is needed to identify the resources relationship. If users used other types of documents, they still need to provide the schema in the schema field, as VisFlow doesn't have a proper wrapper and matcher to extract the unknown type of document. Adding web services as resources requires more information than local datasets. The URL and attributes are needed; they contain the web service parameters that are submitted as shown in Figure 5.3a which are well-documented. The Method Return File type, Return File type, and Suggested Output Name are three fields used in VisFlow for a wrapper suggestion for a response from the web pages. The Method Return File type denotes the endpoint return file types. The Return File type records the registered resources return type. VisFlow has integrated wrappers to transfer different method return data formats from CSV, XML, JSON, and HTML to CSV or XML format. By providing this information together, a configuration of web service call in VisFlow is straightforward. After every field has been filled in by users, they have to click the add button to finalize the registration.

Adding libraries in VisFlow is similar to adding resources as shown in figure 5.3b, users need to provide the executable file, and a description of this executable file includes attributes that are required in this function. The attributes rows show the detailed required information, the attribute name, the attribute value, if it is required to be shown by users, the attribute type, an example for the value, and a description of what this attribute does. The current version of VisFlow supports Windows Executable, Jar File, Windows Bash, Python script, and R script which can be called and run using command line parameters. Using these attribute actions combined, users are allowed to generate and involve essential reusable functions in the workflow.

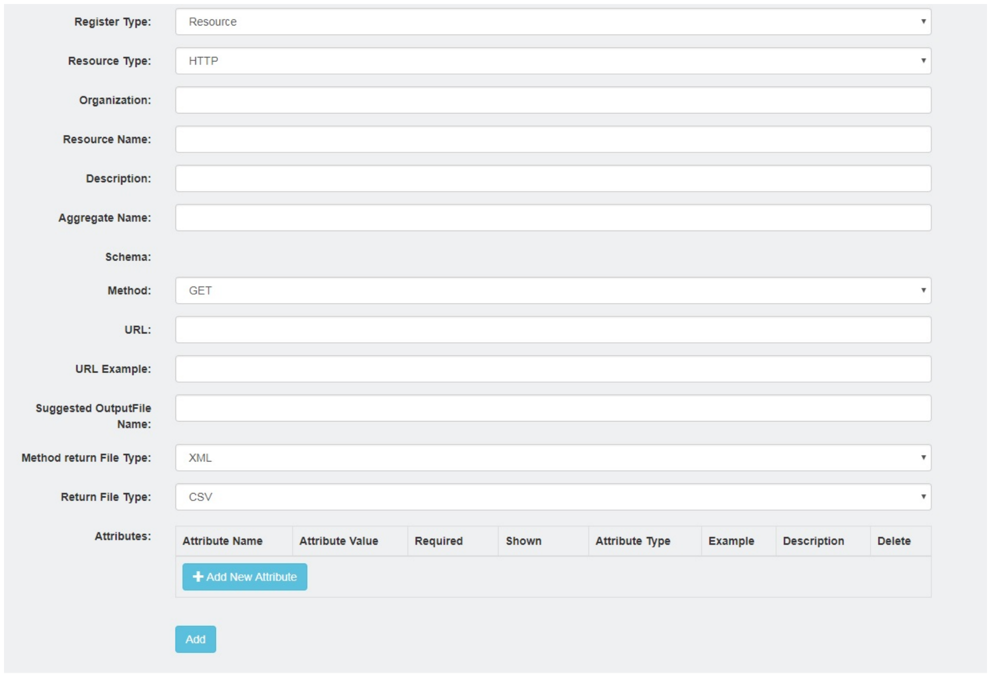
The next page is a resource and library browser page shown in Figure 5.4 where users are allowed to view registered resources and libraries in the VisFlow system. After users click the resources, the detailed information pops up to the right window, and they can click review to dive into the details. If the resources are too big to view, VisFlow downloads it to the local computer. It has a free text search button for users to filter out the resources' names. The library viewer is similar to resources, and can easily access via clicking the libraries tab.

The last page is the VisFlow example page in Figure 5.5. The following seven comprehensive examples are provided and are classified into three different catalogs: bioinformatics, geoscience, and Feature examples to illustrate how useful and efficient our workflow is. For the detailed discussion about each example, please refer to Chapter 7.

5.2 VisFlow System Architecture

VisFlow leverages the experiences of successful and popular systems such as Taverna, Galaxy, Kepler, and SADI, and balances usability, computability, and coverage. We focus on simplicity of application coding using abstractions, offer a mix of tools to be able to express application needs (allowing the use of libraries, online resources, custom codes, and established services) over a set of arbitrary resources. The design of VisFlow is also based on the observation that while important, factors such as computational efficiency and theoretical superiority of the workflow language usually are lesser concerns [54], and with the belief that to improve usability and acceptance, VisFlow must be general enough and have wider coverage of application domains.

The basic design of the VisFlow interface closely parallels the design of the systems such as PhyloBase [77] and is a departure from a pure form-based workflow articulation in VizBuilder [71]. Technically, VisFlow is an improved version of VizBuilder and is superior in a number of ways. First of all, VisFlow uses a set of icons that can be used to draw a workflow on a canvas on a web browser as a process graph using the language of VisFlow, while VizBuilder is a desktop application. VisFlow includes a resource registry using which VisFlow offers active support in the design process, eliminating possible design errors and offering semantic suggestions specific to intended applications, which VizBuilder did not. While both systems are based on the declarative workflow language BioFlow, VizBuilder supported far fewer abstractions than does VisFlow, and thus required more imperative coding. Finally, VisFlow

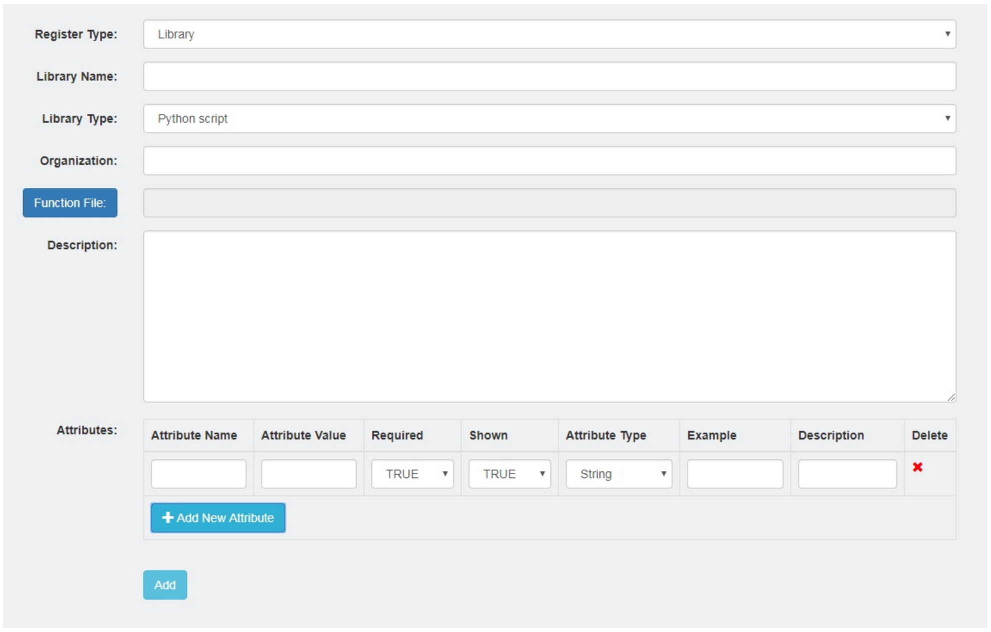


Register Type:	Resource
Resource Type:	HTTP
Organization:	
Resource Name:	
Description:	
Aggregate Name:	
Schema:	
Method:	GET
URL:	
URL Example:	
Suggested OutputFile Name:	
Method return File Type:	XML
Return File Type:	CSV

Attributes:

Attribute Name	Attribute Value	Required	Shown	Attribute Type	Example	Description	Delete

(a) VisFlow Register Resource Page



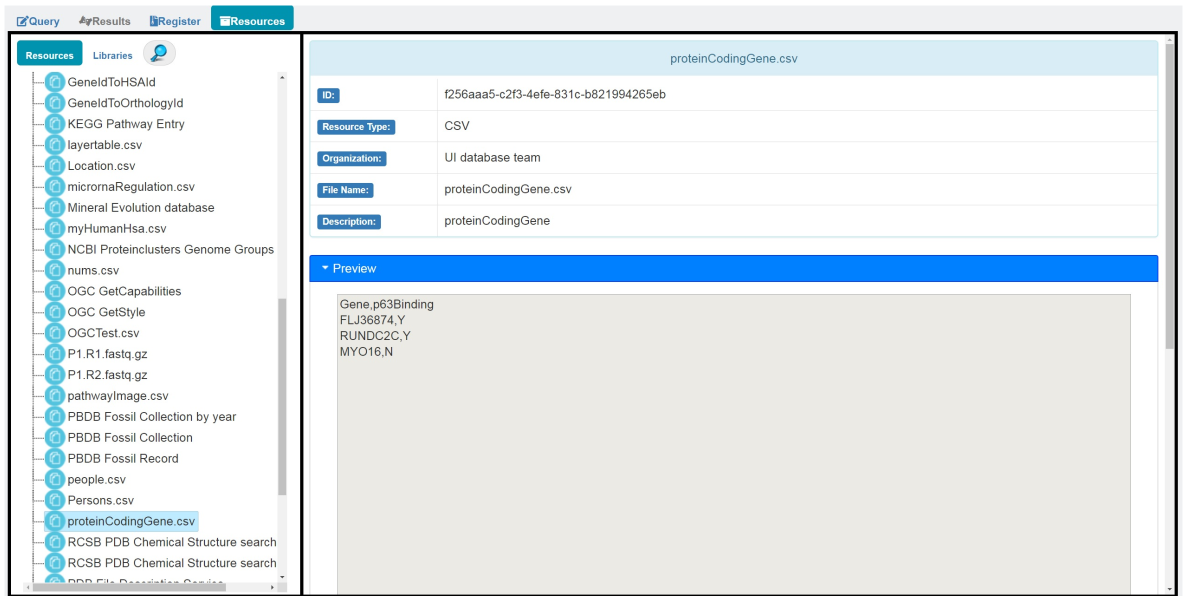
Register Type:	Library
Library Name:	
Library Type:	Python script
Organization:	
Function File:	
Description:	

Attributes:

Attribute Name	Attribute Value	Required	Shown	Attribute Type	Example	Description	Delete
		TRUE	TRUE	String			✖

(b) VisFlow Register Library Page

Figure 5.3: Examples of VisFlow Registry Page



GeneIdToHSAId
GeneIdToOrthologyId
KEGG Pathway Entry
layertable.csv
Location.csv
micrornaRegulation.csv
Mineral Evolution database
myHumanHsa.csv
NCBI Proteinclusters Genome Groups
nums.csv
OGC GetCapabilities
OGC GetStyle
OGCTest.csv
P1.R1.fastq.gz
P1.R2.fastq.gz
pathwayImage.csv
PBDB Fossil Collection by year
PBDB Fossil Collection
PBDB Fossil Record
people.csv
Persons.csv
proteinCodingGene.csv
RCSB PDB Chemical Structure search
RCSB PDB Chemical Structure search

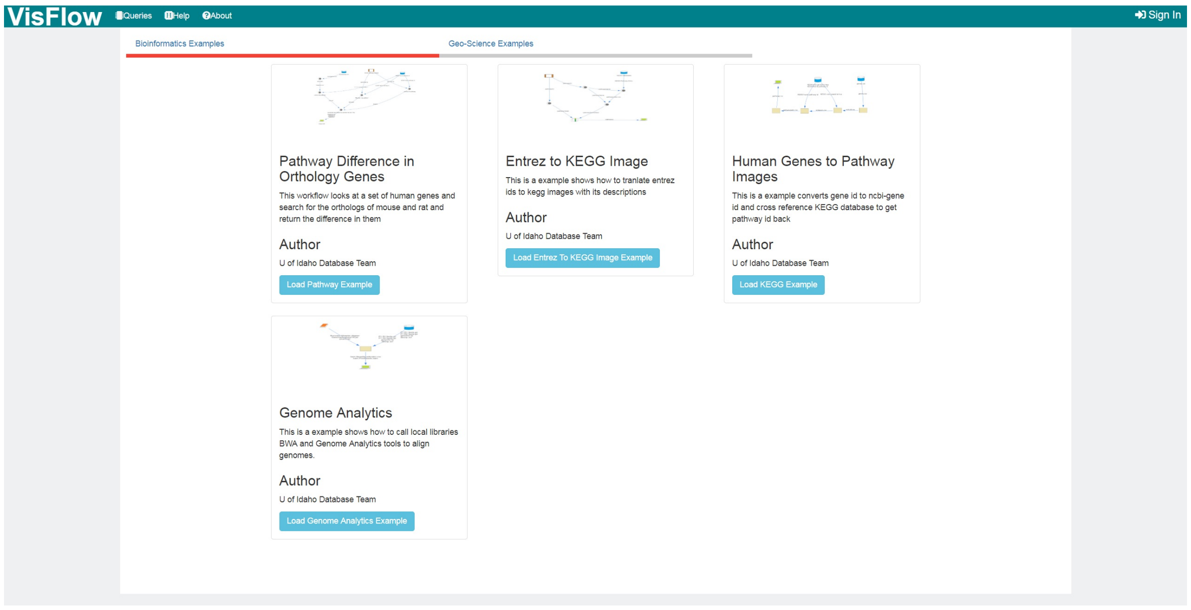
proteinCodingGene.csv

ID:	f256aaa5-c2f3-4efe-831c-b821994265eb
Resource Type:	CSV
Organization:	UI database team
File Name:	proteinCodingGene.csv
Description:	proteinCodingGene

Preview

```
Gene,p63Binding
FLJ36874,Y
RUNDC2C,Y
MYO16,N
```

Figure 5.4: VisFlow Resources Page



Pathway Difference in Orthology Genes

This workflow looks at a set of human genes and search for the orthologs of mouse and rat and return the difference in them

Author

U of Idaho Database Team

Load Pathway Example

Entrez to KEGG Image

This is a example shows how to tranlate entrez ids to kegg images with its descriptions

Author

U of Idaho Database Team

Load Entrez To KEGG Image Example

Human Genes to Pathway Images

This is a example converts gene id to ncbi-gene id and cross reference KEGG database to get pathway id back

Author

U of Idaho Database Team

Load KEGG Example

Genome Analytics

This is a example shows how to call local libraries BWA and Genome Analytics tools to align genomes.

Author

U of Idaho Database Team

Load Genome Analytics Example

Figure 5.5: VisFlow Examples Page

supports three data formats (CSV, XML, and tables), library functions, and users' codes (in Python, R, XQuery, and SQL), and flexible data visualization, which VizBuilder did not. However, the semantics of VizBuilder icons could be redefined by changing their operational semantics as a distinct process, which requires overhauling of the system in VisFlow. In the sections below, we discuss the architecture of VisFlow as shown in Figure 5.6 and its three major functional components – the *User Interaction Module* (UIM), the *Query Rewriting Engine* (QRE) and the *Query Processing Engine* (QPE), below.

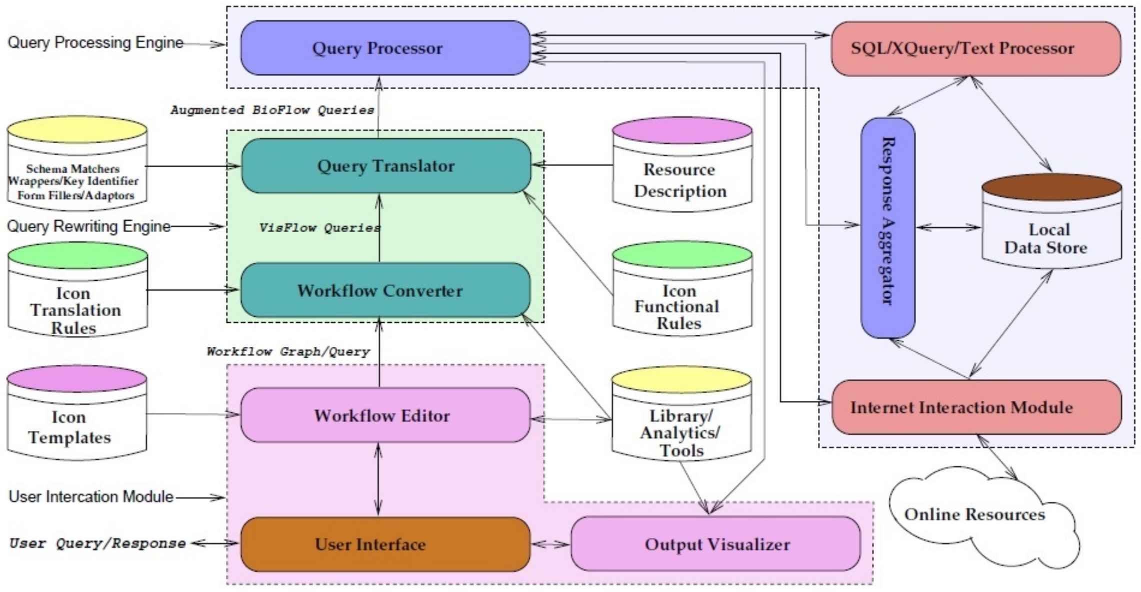


Figure 5.6: VisFlow Architecture.

5.2.1 User Interaction Module

Users interact with VisFlow through the web-based front-end UIM. The scripting of this front-end uses HTML 5, JavaScript ECMAScript 6, and JQuery 1.12.4. The workflow graph drawing is implemented using vis.js 4.16.1 graphics package that supports flexible editing and manipulation of graphics objects, while HTML 5 is used to render the interface overall. JavaScript with JQuery together is used to control and manage user interactions, show useful information, and navigate throughout the interface pages. They are also used to submit the composed workflows to the VisFlow server and display and visualize returned results in the *Results* tab. The current version of VisFlow front-end is only supported on Chrome web browsers, version 40.0.2214 or higher.

5.2.2 Semantics Preserving Query Mapping

In VisFlow, the workflow graphs are converted into a possible parallel sequence of tasks for the back-end execution engine. VisFlow workflows potentially contain a visual representation of BioFlow statements, or a combination of imperative program codes in Java, R, or Python both in executable form or in source code form, and SQL and XQuery to manipulate VisFlow database information or both. BioFlow visual abstractions in VisFlow, which are stored as JSON files are automatically translated to Java objects using Google GSON library and executed as Java code. Code segments written directly as operations of VisFlow steps are compiled or executed using the command line in Windows 7 OS which may include Java Jar files, C++ compiled files, Python scripts, R Scripts, windows executable files, windows bash files, etc. In both BioFlow scripts and user-supplied codes and libraries, HTTP, and FTP protocol based internet services are supported, including RESTful based web services. The translator from VisFlow to BioFlow is written in Java 8. Intuitively, the QRE system delivers an executable script of VisFlow to the query processing engine written as a sequence of BioFlow, SQL, and XQuery statements, and a set of executable imperative codes written in Python or R as a partial order.

5.2.3 Query Processing Engine and Query Execution

The translated workflow scripts are executed by the back-end QPE implemented in Java 8 as a Java Servlet hosted on Tomcat server 8.5.11 using RESTful Jersey web service 2.23.1 as the gateway under Windows 7 Ultimate 64bit OS. Translated scripts are processed using Maven 3.3.9 to manage third-party packages such as Google GSON 2.5 to manipulate JSON objects, JDBC package for database connections, and Selenium 2.0.0 for our wrapper implementation. Google Guice 4.0 and Hexokinase 2 (HK2) 2.4.0 are used for dependency injections in Java Servlets for efficient handling of Java objects.

We use eXist 2.2 and MySQL community server 5.5 as the native and primary databases to support XQuery and SQL queries, and to store XML and relational data. We also use them for secondary data processing when appropriate. While we support mixed mode XQuery and SQL queries separately, we do not yet support them together in one single query, i.e., a table and an XML document in a query. Doing so requires a conversion of one into another to use a single query platform.

5.3 VisFlow System Features

Besides the fact that VisFlow is a declarative language and has capabilities of parallel execution whenever needed, VisFlow also has a resource and library management system, auto data integration, auto file type transformation, auto suggestions, ad hoc debugging, data visualization, and visual query builder built-in features.

5.3.1 Resource and Library Management

Resources are primarily data stored in local and remote repositories. Among these operators, only the *data* icon is a terminal icon, and the remaining two - *combine* and *fusion* - do not require description as they accept inputs from other operators that produce data. Therefore, we only discuss how *data* icons are described. Resource description has two major parts - resource identification and resources submitting. Identification may be described in two principal ways - by qualitatively expressing what to match, and by explicit identification. Qualitative description is a mixed list of features of a data collection using ontological terms or attribute names regardless of the type of resource - table, text, or XML document. An explicit identification, on the other hand, is a specific name of a resource and its Internet location. To describe the subset to be used, additional features are listed. The list of features in the identification are inherited in the subset list and are overridden if some of it is not intended to be in submitted. In either case, the resource is named eventually. A resource icon may describe a set of resources, not just one.

5.3.1.1 Resource Identification

Indexed resources are identified using a type-ahead stepwise method described earlier. Given any subset of a set of attribute names, an URL, a table name, a mapping function τ returns at most the top-k objects matching the identifying properties. Users can set the value of k in advance. The exciting aspect is that these identifying strings need not be identical to the catalog description. A suitable schema matching function δ establishes the match in a user transparent manner. The matching and mapping continue until a unique match is found or users select one from the set of k matches returned by the system. A system-wide map table then is maintained to preserve the user view of the workflow scheme. Resource matching can be accomplished in two ways: active matching and direct matching.

5.3.1.2 Top-k Matching Algorithm

Active matching

The icons are pre-filled from the meta-data library using a match function μ that uses the supplied information in the description form for each icon and suggests resources that are a potential match. These matches are qualitative and are solely based on an abstract description of the icons and the resource properties, and their locations play no role in the matching process. Users can select between two options -accuracy or efficiency. Inaccuracy based suggestions, the mapping, and appropriateness are the only criteria considered in the matching process. On the other hand, inefficiency based matching accuracy is compromised to the extent possible to prefer the most efficient execution. That essentially means the location of the resource in terms of network communication cost, hardware and total response time are primary concerns. Users can preselect how many suggestions per icon they prefer at most. Based on the top-k match, the interface shows possible choices for each icon and each edge, including possible edges introducing possible adapters for edges between resource and analytics, and resource and resource pairs. Users will be able to select (using a bank of radio buttons that selects and deselects a suggestion) and finalize any of the suggestions to make the remaining suggestions disappear, or deselect one to make the suggestions reappear. Identifying information is displayed in green.

Direct matching

Users can make substring matching with a resource by specifically supplying all the identifying information to the system by describing an icon. In such instances, no suggestions for these icons are made. The whole algorithm is based on substring matching. The radio button "select" will be active in the radio button bank for each icon.

S-Match Algorithm

The matching algorithm is based on the S-Match [63] algorithm to support active matching and direct matching. We have extended this idea to fit our matching usage. The basic idea is still to compute the relations between the concepts at a node. To do that, first the intended meaning is extracted using WordNet[1], then we calculate the meaning of the node based on their position. During the third step, the semantic and syntactic relations between the concepts of a label of two input trees are generated, and lastly, the semantic relations between the concepts are computed. For direct matching, we filter out three other relationships and return equivalence nodes if there is one exists in the data resources. For active matching, we return equivalence, less general and more general nodes if it exists a relationship like this. Figure 5.7 shows an example of this process in the *data* icon. It shows the input panel where users are allowed to submit a tree structure to VisFlow resource management systems. In this example, a tree structure protein with its child gene keywords is filled. After users click the submit function, the result is generated where the matched field is highlighted the same color as in the query field. In this case, there are three data resources that are returned to let users choose the best one. The matching resources description help users to select the proper resources. This is very helpful when there

[1]`https://wordnet.princeton.edu/`

are a large number of datasets, and later when users are allowed to share their datasets, this process helps them to reduce the search time when they need to find the proper datasets in their workflow.

The library management in VisFlow deals with the functions used in the workflow. Like the *data* icon, the *library* icon is a terminal icon, and function description also has two major parts - function identification and function submitting. An XML document is used to store this information for each function, with its input attribute, the commands to call this function, plus descriptions of what this function does. The *library* icon also provides a top-k matching function to help users select the functions. The user interface is similar to data icon, and it also allow the user to provide a tree structure query to search among the registered library functions. The process of this top-k search is similar to resources top-k search. The data is just on different domains.

5.3.2 Auto Data Integration

Data Integration during the workflow composition is painful for domain scientists to deal with. VisFlow has provided two auto data integration procedures to help reduce the integration work. What's more, the extract and merge functions provided in the *analytics* icon also help users to extract attributes from datasets and merge results from web services. The merge function is used a lot during web service calls because most of the time the submitted parameters information is lost in the return datasets, and it's difficult to collect this information.

5.3.2.1 Collating Objects: Combine Operation

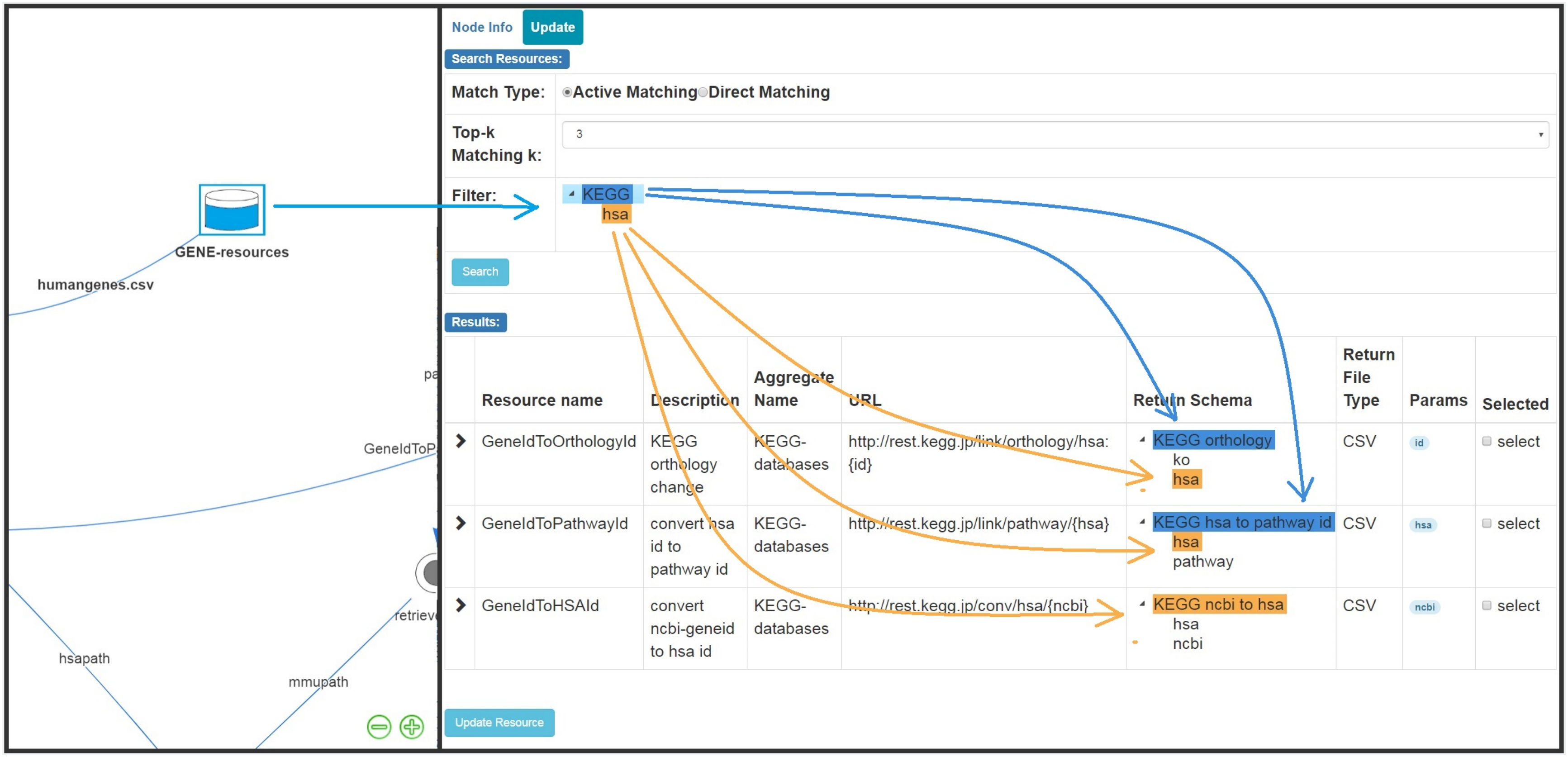
The combine operation directly matches the *combine* icon in VisFlow. In our definition, two relations are combined compatible if both share at least one candidate key.

Definition 5.3.1. Let r and s be two relations and K_r and K_s be two corresponding candidate keys. r and s are combine compatible if the subrelations with attributes in K_r and K_s are union compatible in the classical sense. □

The above definition implies that the disagreement of the remainder of the schemes does not matter so long as the candidate keys are compatible. The output scheme has an outer union semantics for the attributes not in the candidate keys that make the relations combine compatible.

5.3.2.2 Aggregating Objects: Link Operation

In value-based systems such as the relational model, an object represented differently in two different tables are considered different. For example, consider the two table instances (Table 5.1a and Table



Match Type: ◉ Active Matching ○ Direct Matching

Top-k Matching k: 3

Filter: KEGG / hsa

	Resource name	Description	Aggregate Name	URL	Return Schema	Return File Type	Params	Selected
›	GeneIdToOrthologyId	KEGG orthology change	KEGG-databases	http://rest.kegg.jp/link/orthology/hsa:{id}	KEGG orthology / ko / hsa	CSV	id	☐ select
›	GeneIdToPathwayId	convert hsa id to pathway id	KEGG-databases	http://rest.kegg.jp/link/pathway/{hsa}	KEGG hsa to pathway id / hsa / pathway	CSV	hsa	☐ select
›	GeneIdToHSAId	convert ncbi-geneid to hsa id	KEGG-databases	http://rest.kegg.jp/conv/hsa/{ncbi}	KEGG ncbi to hsa / hsa / ncbi	CSV	ncbi	☐ select

Figure 5.7: VisFlow Top-k Matching.

GeneName	**Organism**	**Function**
repA1	Greenbug	plasmid maint.
UQCC	Human	growth control

(a) Table function.

GeneID	**GOTerm**
1246500	GO:0006276
55245	GO:0001558

(b) Table Gene.

GeneName	**Organism**	**Function**	**GeneID**	**GOTerm**
repA1	Greenbug	plasmid maint.	1246500	GO:0006276
UQCC	Human	growth control	55245	GO:0001558

(c) Linked table.

Table 5.1: Gene Information Tables

5.1b) with their primary key underlined. Although it is difficult to observe from the representation, the tuples in both relations are conceptually similar, i.e., *repA* $\equiv$ *1246500*, and *plasmid maintenance* $\equiv$ *GO:0006276*, and so on. By joining them vertically, the only truly new information we learn is the *Organism* name. This vertical conjoining is called *linking* in BioFlow.

In our quest to stay as close as possible to the relational model, we choose a value based definition for the link operation and leave entity resolution to the key discovery function such as GORDIAN. But such a choice will mean that once the keys or entities are identified, linking requires testing equality of values, not equivalence (perhaps using ID mapping [45, 76], or Blast search). In other words, in the current version of VisFlow, the link operation requires that column values of the two tables be identical (i.e., for *GeneName* = *GeneID* to be successful, both columns must have *repA* as values, and not *repA* and 1246500) even though the column names are entirely different which potentially cannot be mapped using a schema matcher such as the S-Match system we have used. As such, similar to natural join, we will remove the duplicate columns[2]. We require that identified objects share a candidate key at least partially, regardless of their attribute names, i.e., one key must be an improper superset of the other key that allows for multiple objects to join with the object (supporting 1-M relationship). This operation becomes identical to the intersection operation when the two relations have identical keys.

Thus, fusion also combines two or more tables based on key identification in ways similar to join in relational model. But, differently from join, it links objects based on entity identifying keys and common attributes. In other words, to fuse or aggregate two objects, the objects must have a set of common attributes after schema heterogeneity is resolved and those common attributes must include the "object key". For both icons, there must be at least two incoming arrows, and at least one outgoing arrow – combine and fusion cannot be a terminal node in a workflow graph.

[2]A more interesting definition will join the relations based on object identities and retain all the columns as in the tables in figure 5.1c.

5.3.3 Auto File Type Transformation

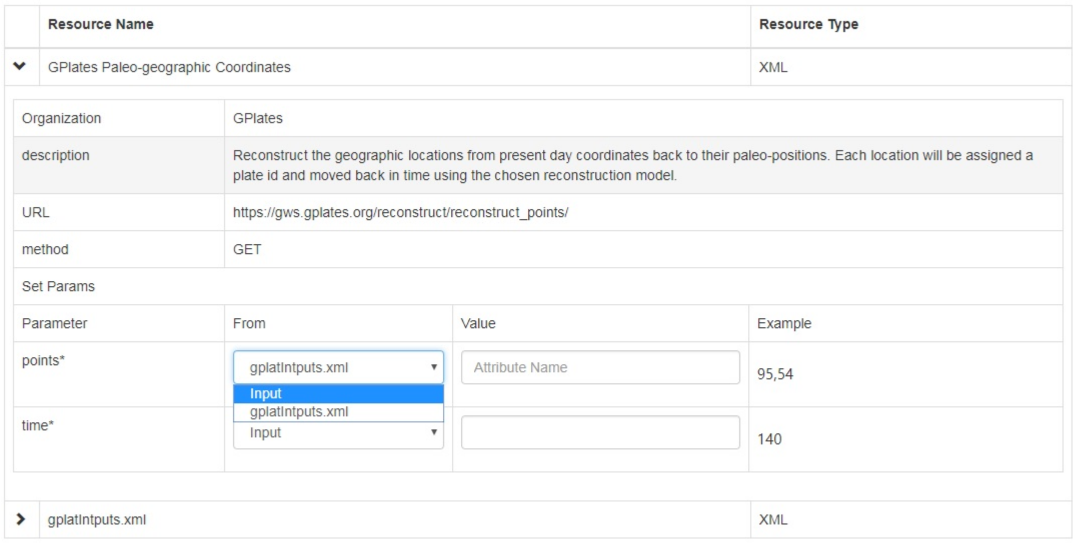
In VisFlow, it allows transform file types among the relational table, CSV file and XML document. When converting from CSV file to relational table, the first step generates the create statement based on the metadata stored in the system, then use MySQL built-in statement "load data local infile" for users uploading data to MySQL server. When converting from relational table to CSV file, `select` statement has been called, and the results are parsed by comma separator. When converting from CSV file to XML document, the root element is named "resultsets" and for each row of the value is named "row". The attribute of each row is mapped as the name, and the value is mapped as the content in the element. When converting from XML document to CSV file, only the deepest element is extracted, we require all the values in the XML file must at the same level. The converting algorithms from a relational table to XML document or XML document to relational table use the CSV file as the middle type. They all first convert them to CSV file, then apply the algorithms we discussed before to the corresponding files.

5.3.4 Auto Suggestions

The auto suggestions function is another feature for users to generate a query to submit to the web services and retrieve the data back. Thanks to the meta-data information stored in VisFlow, the schema of the input dataset is known, as well as the required input attribute, and a schema matching algorithm is applied against these attributes with the required one. Auto suggestions is super helpful when the required inputs are large, and users don't need to fill out all the fields and then submit it to the server. Auto suggestions also reduce the total time composing the workflow. Figure 5.8 shows an example in which the suggestion algorithm successfully selects the matching field from the input resource and puts in the value part.

5.3.5 Ad Hoc Debugging

The Ad Hoc Debugging feature is another important feature to help domain scientists generate the workflow. It allows part of the workflow to be submitted to the server and executes and stops exactly at the node to provide detailed information about current data resources. Users only need to configure the workflow as needed, even if some type error or syntax error happened during the workflow execution. If the path from the data resources to the current node is executable, the results will be generated. The red and green rectangle in Figure 5.9 shows the data resources passed in this adapter and the data resources generated using this *adapter* icon. The content of data resources are shown under the

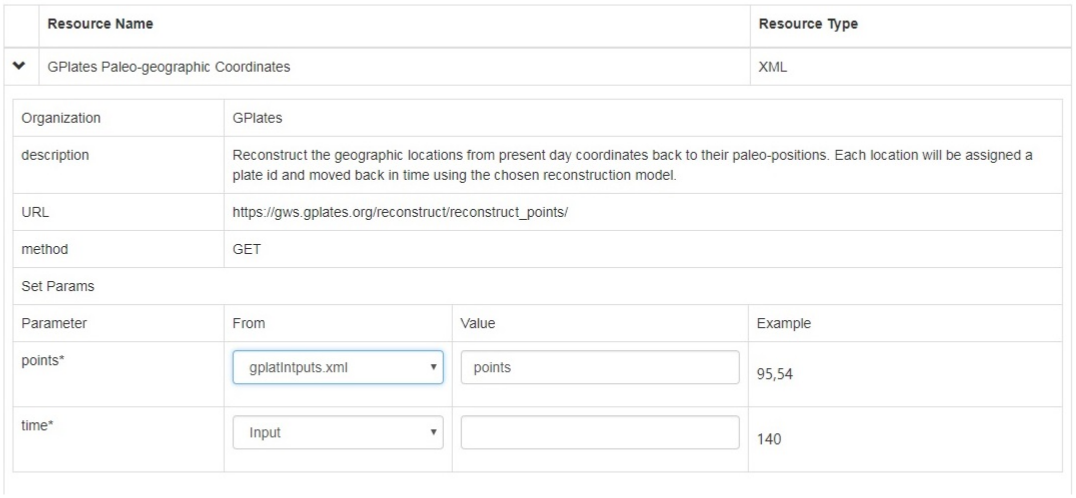


	Resource Name	Resource Type
⌄	GPlates Paleo-geographic Coordinates	XML

Organization	GPlates		
description	Reconstruct the geographic locations from present day coordinates back to their paleo-positions. Each location will be assigned a plate id and moved back in time using the chosen reconstruction model.		
URL	https://gws.gplates.org/reconstruct/reconstruct_points/		
method	GET		
Set Params			
Parameter	From	Value	Example
points*	gplatIntputs.xml (Input / gplatIntputs.xml)	Attribute Name	95,54
time*	Input		140

›	gplatIntputs.xml	XML

(a) VisFlow Auto Suggestions: before



	Resource Name	Resource Type
⌄	GPlates Paleo-geographic Coordinates	XML

Organization	GPlates		
description	Reconstruct the geographic locations from present day coordinates back to their paleo-positions. Each location will be assigned a plate id and moved back in time using the chosen reconstruction model.		
URL	https://gws.gplates.org/reconstruct/reconstruct_points/		
method	GET		
Set Params			
Parameter	From	Value	Example
points*	gplatIntputs.xml	points	95,54
time*	Input		140

(b) VisFlow Auto Suggestions: after

Figure 5.8: Examples of VisFlow Web Service Call Parameter Suggestion

window. Unlike most of the workflow management systems, with this feature users have capabilities of composition workflows by step validation, to make sure every step is correct and when the whole workflow is finished, no debug is needed for the whole workflow.

5.3.6 Data Visualization

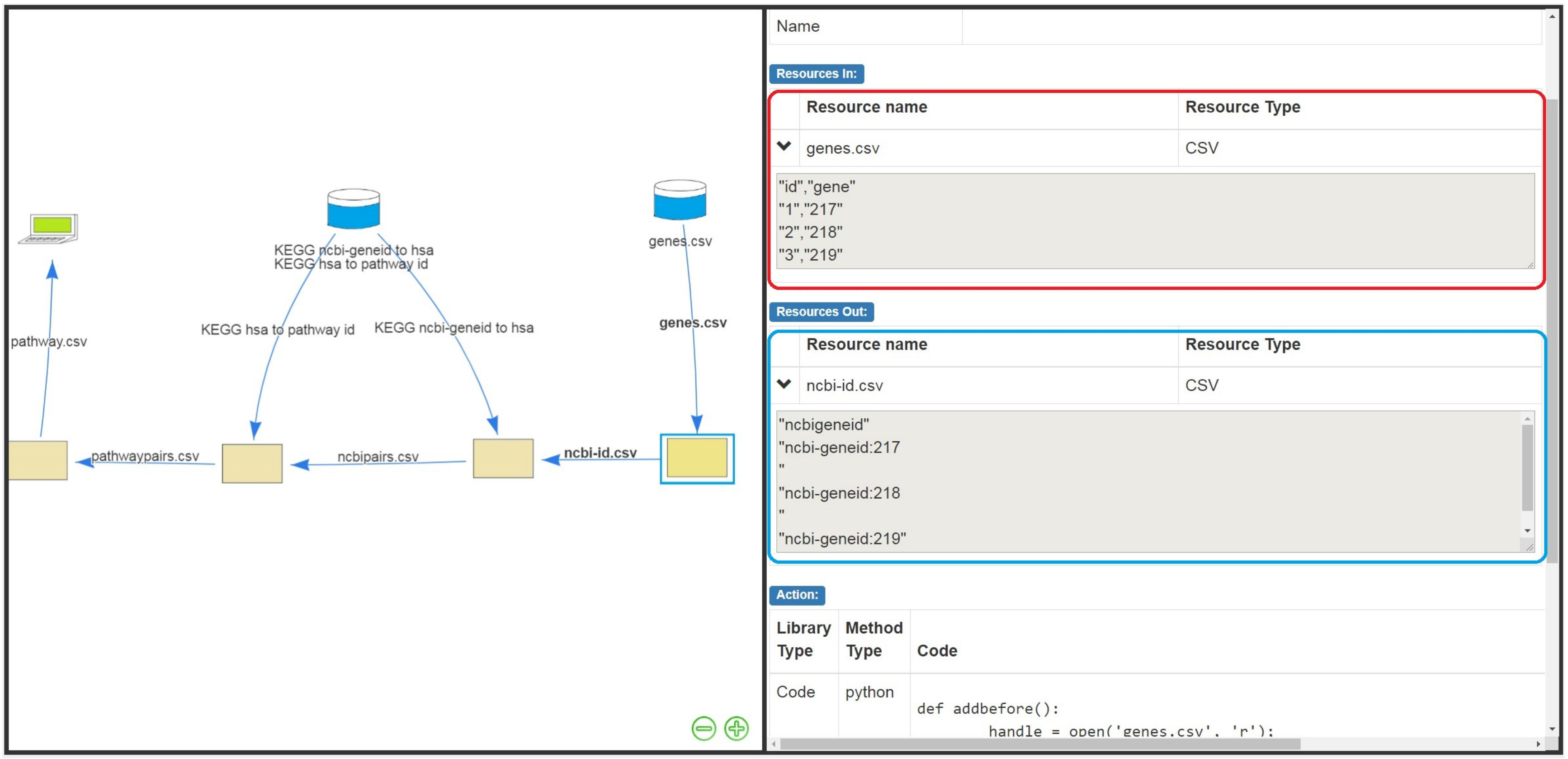
The *Terminal* icon controls the data visualization in VisFlow. A *terminal* icon provides a list of data visualization tools to help users to view the data. In VisFlow, the return file type only supports a flat table, file, or an image type. There are tools to extract table attributes and show them as coordinates in a map like the Google Map tool, and there are tools to help users to show Pathway images by merely providing the Pathway IDs in the return file and selecting the image type. These tools are pre-installed in the VisFlow system. Figure 5.10 shows the *terminal* icon update panel (the red rectangle) and some examples rendered by visualization tools based on the rules configured in this panel. These rules can apply to entire datasets or single columns. For tools applied on the datasets, users need to specify the parameters these tools required, and for tools applied on a single column, column names are required. VisFlow supports split the frames into small windows for visualization. Users are allowed to bind different tools to these windows. Currently, this version of VisFlow is not allowed to install custom visualization tools into a *terminal* icon. In a future release, we are planning to support custom visualization tools introduced by users and installed into the *terminal* icon.

5.3.7 Visual Query Builder

In order to take advantage of SQL and XQuery, VisFlow provides a visual query builder to help users generate these two scripts without knowing the details of these two languages. It also benefits from all the built-in functions and declarative syntax which is easy to understand by users without any programming knowledge. Visual query builder supports three essential functions: create a new table, remove rows and modify columns which allow users to create a new table and remove rows based on the condition and modify column values.

Visual query builder provides a simple way to manipulate the data resources by taking advantage of these declarative languages, and the visual query builder has limited supported statements for the basic functionalities. If users want more sophisticated requirements for the analytics purpose, they have to learn these languages to manipulate it.

There are papers that proposed algorithms to translate SQL to XQuery [114, 109]. They are focus on the integration of SQL and XQuery in Oracle database. Only a couple of papers [80, 53] talks about the translation from SQL to XQuery using `select` and `join`. Most of these papers deal with flat XML



Name

Resources In:

	Resource name	Resource Type
❯	genes.csv	CSV

```
"id","gene"
"1","217"
"2","218"
"3","219"
```

Resources Out:

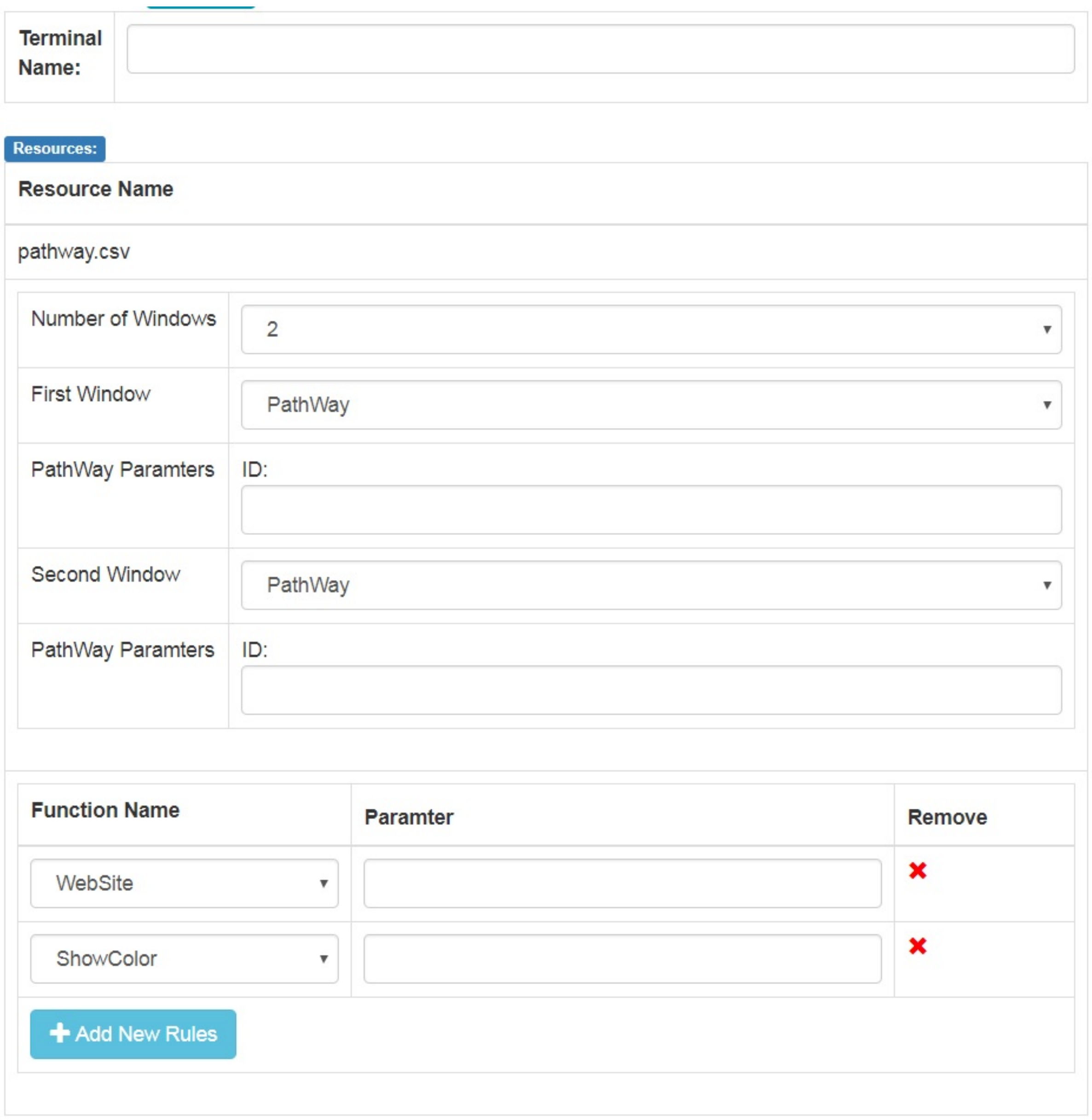
	Resource name	Resource Type
❯	ncbi-id.csv	CSV

```
"ncbigeneid"
"ncbi-geneid:217
"
"ncbi-geneid:218
"
"ncbi-geneid:219"
```

Action:

Library Type	Method Type	Code
Code	python	`def addbefore():` `handle = open('genes.csv', 'r'):`

Figure 5.9: VisFlow Debugging Example



Terminal Name:	

Resources:

Resource Name
pathway.csv

Number of Windows	2 ▾
First Window	PathWay ▾
PathWay Paramters	ID:
Second Window	PathWay ▾
PathWay Paramters	ID:

Function Name	**Paramter**	**Remove**
WebSite ▾		✖
ShowColor ▾		✖

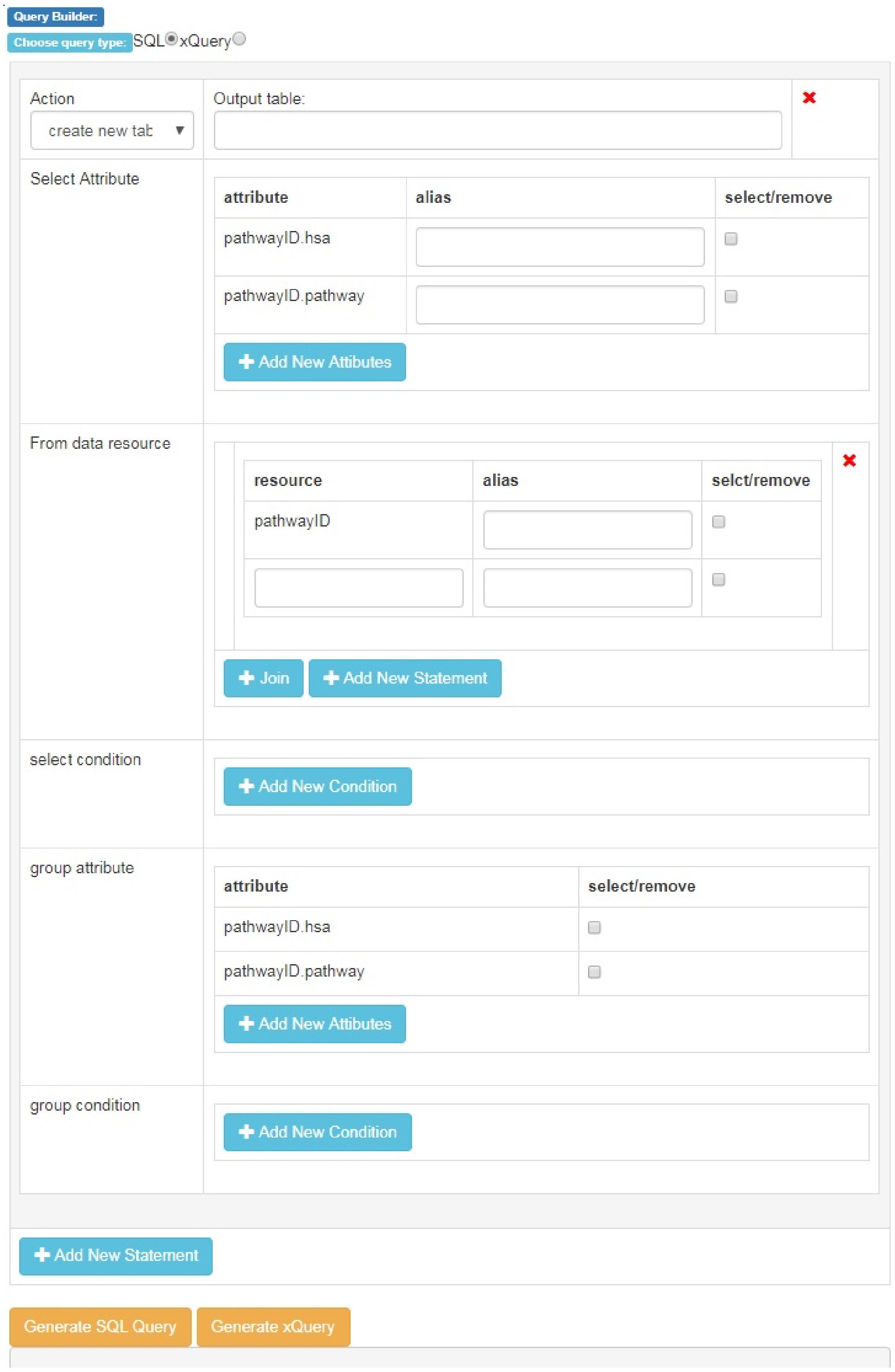
Figure 5.10: VisFlow Visualization

documents, not for nested XML documents. Unlike these systems, visual query builder supports SQL `select`, `delete` and `update` and keeps the structure information as much as possible.

5.3.7.1 Create New Tables

The creating new table interface is used for users to select attributes from multiple data resources and merge them based on predefined conditions. The query generation interface for creating a new table is shown in Figure 5.11. The SQL `select` statement inspires the design of this interface. The first field is for selecting attributes from the data resources, or you can create your statement using the add new attributes button, which allows you to use SQL built-in functions as well. The next field is the data resource field, where uses the data resources passed from the previous icons as inputs. The following field is the select condition, where users fill out the conditions for attributes selected in the first field. After that, the group and group condition work together for aggregate attributes information.

Mapping from the visual query builder to SQL script is straightforward; these fields have a one to one mapping to SQL `select`, `from`, `where`, `join`, `on`, `group by` and `having` statements. Mapping from the visual query builder to XQuery is not direct. From the visual query builder, we collect information of the form $\langle resName, Alias, R, W, S, N, B, H, F\rangle$ to generate the XQuery script. Algorithm 5 shows the detailed definition of these elements and steps. The basic idea of translating SQL like script into XQuery is using XQuery `For`, `Let`, `Where`, `Return`, and `Group by` statements. The algorithm has five small steps. In line 1, it generates the select attribute, inner join, on, and select condition fields. In line 2, the algorithm deals with a group by and their conditions. The next step is to call the functions that are applied to these elements. In visual query builder, we only support functions that are originally built-in to the language and used on attributes. In these steps, all the elements are stored under $< row >$ element and the original root to attribute information has been lost. To recover this path information, the algorithm gets the least common ancestor (LCA) nodes by calling the getLCA function, and after that, it removes current information and adds the new nodes back to the original file as shown from line 4 to line 5 using XQuery update insert statement. Lastly, it returns the result after it removes the empty elements. Figure 5.12 is a sample code generated using the visual query builder to select a gene that equals 217 and group the gene from gene.xml file to generate an output.xml file. These five steps match the highlight parts with their numbers in this graph. Please note in the beginning, we have defined the removeEmptyElements function, and the last part of the code is to save it from the XML database to the file system.



Query Builder:

Choose query type: [✓ SQL] xQuery

Action: create new tab

Output table:

Select Attribute

attribute	alias	select/remove
pathwayID.hsa		☐
pathwayID.pathway		☐

From data resource

resource	alias	selct/remove
pathwayID		☐
		☐

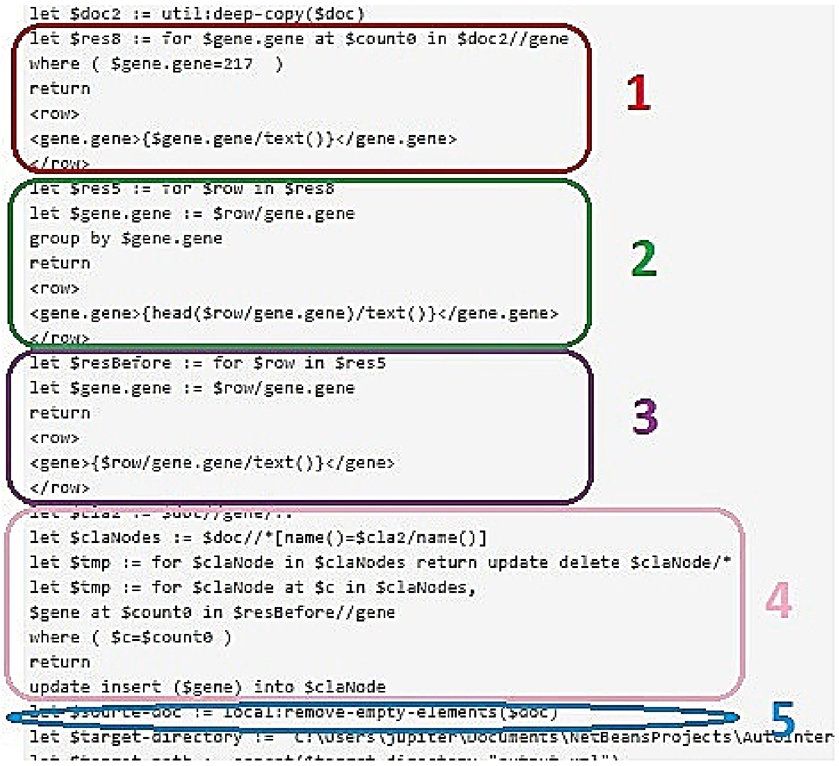
select condition

group attribute

attribute	select/remove
pathwayID.hsa	☐
pathwayID.pathway	☐

group condition

Figure 5.11: Visual Query Builder: Creating a New Table



```
let $doc2 := util:deep-copy($doc)
let $res8 := for $gene.gene at $count0 in $doc2//gene
where ( $gene.gene=217  )
return
<row>
<gene.gene>{$gene.gene/text()}</gene.gene>
</row>
let $res5 := for $row in $res8
let $gene.gene := $row/gene.gene
group by $gene.gene
return
<row>
<gene.gene>{head($row/gene.gene)/text()}</gene.gene>
</row>
let $resBefore := for $row in $res5
let $gene.gene := $row/gene.gene
return
<row>
<gene>{$row/gene.gene/text()}</gene>
</row>
let $cla2 := $doc//gene/..
let $claNodes := $doc//*[name()=$cla2/name()]
let $tmp := for $claNode in $claNodes return update delete $claNode/*
let $tmp := for $claNode at $c in $claNodes,
$gene at $count0 in $resBefore//gene
where ( $c=$count0 )
return
update insert ($gene) into $claNode
let $source-doc := local:remove-empty-elements($doc)
let $target-directory :=  C:\Users\jupiter\Documents\NetBeansProjects\AutoInter
```

Figure 5.12: Visual Query Builder: Creating a New Table Script

Algorithm 5: Create New Tables Statement to XQuery

```
input :
    $resName : resourcename$
    $A$: selected attributes $\alpha_1 \dots \alpha_k$ as list
    $Alias$ selected attributes alias $alias_1 \dots alias_u$ as list
    $W$: where conditions $conW_1 \dots conW_n$ as list
    $N$: join conditions $conN_1 \dots conN_p$ as list
    $B$: group by attributes $\beta_1 \dots \beta_q$ as list
    $H$: having conditions $conH_1 \dots conH_v$ as list
    $F$: function call $f_1 \dots f_u$ as list
output: s
/* select and join                                                    */
1 s ← ”for $\alpha_i \in A$ where N and W return $\alpha$”
/* group by                                                           */
2 s ← ”for $\beta \in B$ where H group by B return $\alpha$”
/* function                                                           */
3 s ← ”for $\beta \in B$ return F”
/* rebuild path                                                       */
4 s ← ”lca:=getLCA(A)”
5 s ← ”removeCLAnodes(lca)”
6 s ← ”for $alias_i \in Alias$ return update insert (Alias) into cla”
/* remove empty elements                                              */
7 s ← ”removeEmptyElements(lca)”
8 return s
```

5.3.7.2 Remove Rows

Figure 5.13a shows the remove rows interface. This interface also inspired by SQL `delete` statement. It provides the functionalities to remove rows based on predefined conditions. The first field is for selecting the delete attributes. The second field is for data resources selection. The last field is for delete condition. Since these fields match the SQL `delete`, `from`, `where` exactly, it is easy to translate it into SQL queries.

To translate this to XQuery, we collect information from an interface of the form $\langle resName, A, R, D\rangle$ to generate the XQuery script. Algorithm 6 shows the definition in input section. The basic idea of this algorithm has three steps. The first step is to find the LCA nodes (line 1), then filter out the nodes by conditions (line 2) and lastly remove the matched nodes. Figure 5.13b shows an example of removing elements where a gene is equal to 217. Similar to the preferred `join` example, highlight code blocks match the algorithm steps one by one. Another part of this script is the supported functions and code for downloading to the file system.

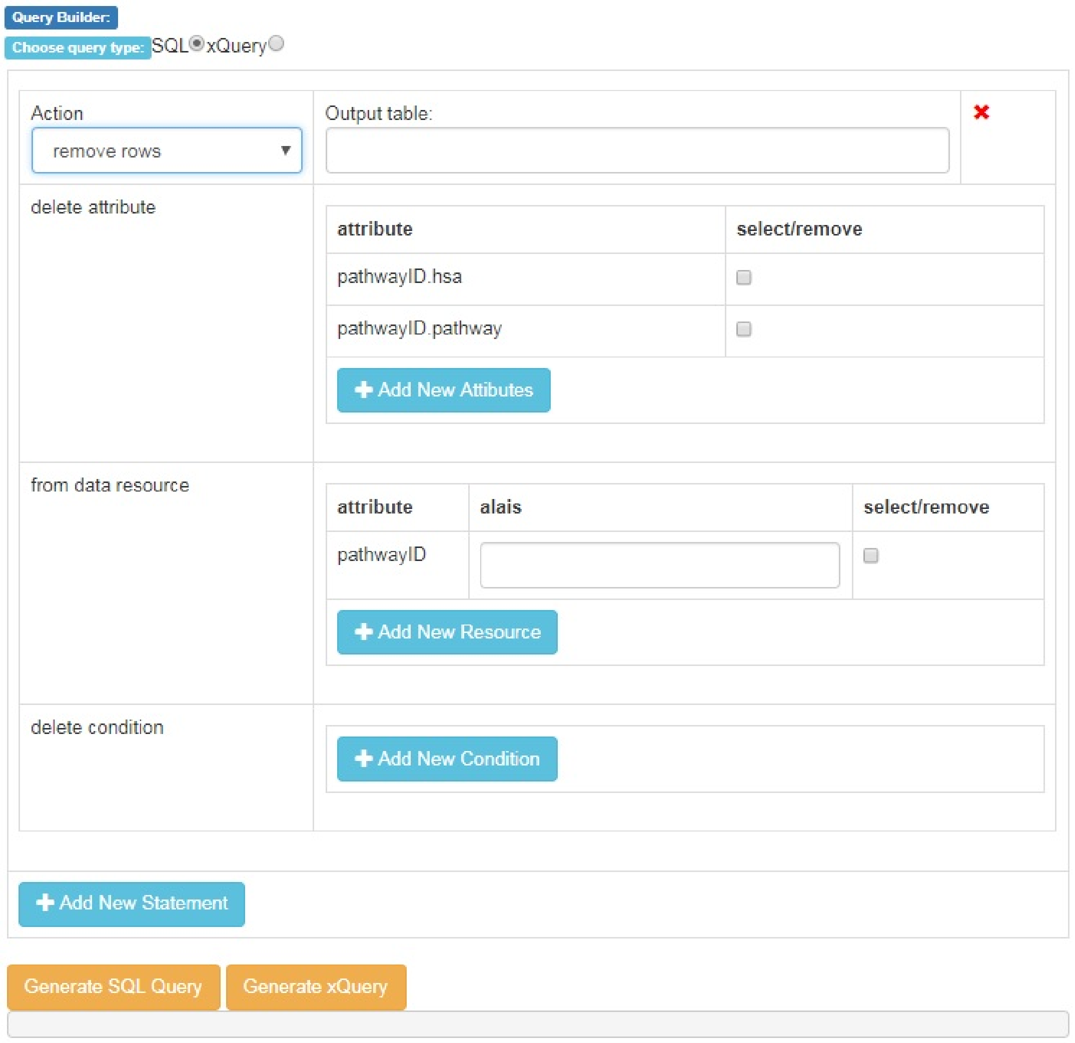
Algorithm 6: Remove Rows to XQuery

```
   input :
   $resName$ : $resourcename$
   $A$: selected attributes $\alpha_1 \dots \alpha_k$ as list
   $D$: delete conditions $conD_1 \dots conD_n$ as list
   output: s
   /* delete                                                     */
1  s ← ”lca:=getLCA(A)”
2  s ← ”deleteNodes:=for row in lca where D return row”
3  s ← ”removeElements(deleteNodes)”
4  return s
```

5.3.7.3 Modify Columns

The modify columns interface shown in Figure 5.14a is based on SQL `update` statements, the first field is for update attributes, the second is for the rules, and the last is for the update condition. These three columns are matched SQL `update`, `set` and `where` statements.

Algorithm 7 shows how to update elements in XQuery. The whole idea is based on a recursion function named updateDeep (line 1) which takes the nodes and attributes as parameters. It steps through each node in the data resources. If one of the nodes match the conditions, it will update the value; the rest of this function is for recursively calling. The algorithm generates the attribute variable and passes as it through the updateDeep function at Line 2. Figure 5.14b shows an example



Action	Output table:
remove rows	

delete attribute

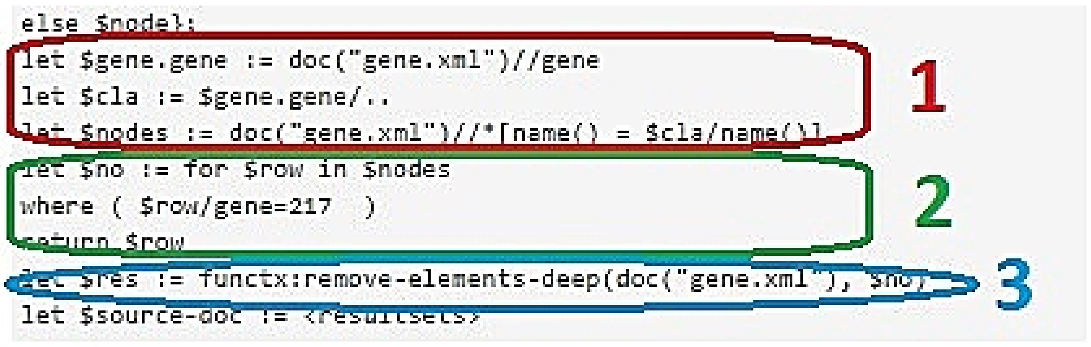
attribute	select/remove
pathwayID.hsa	☐
pathwayID.pathway	☐

from data resource

attribute	alais	select/remove
pathwayID		☐

delete condition

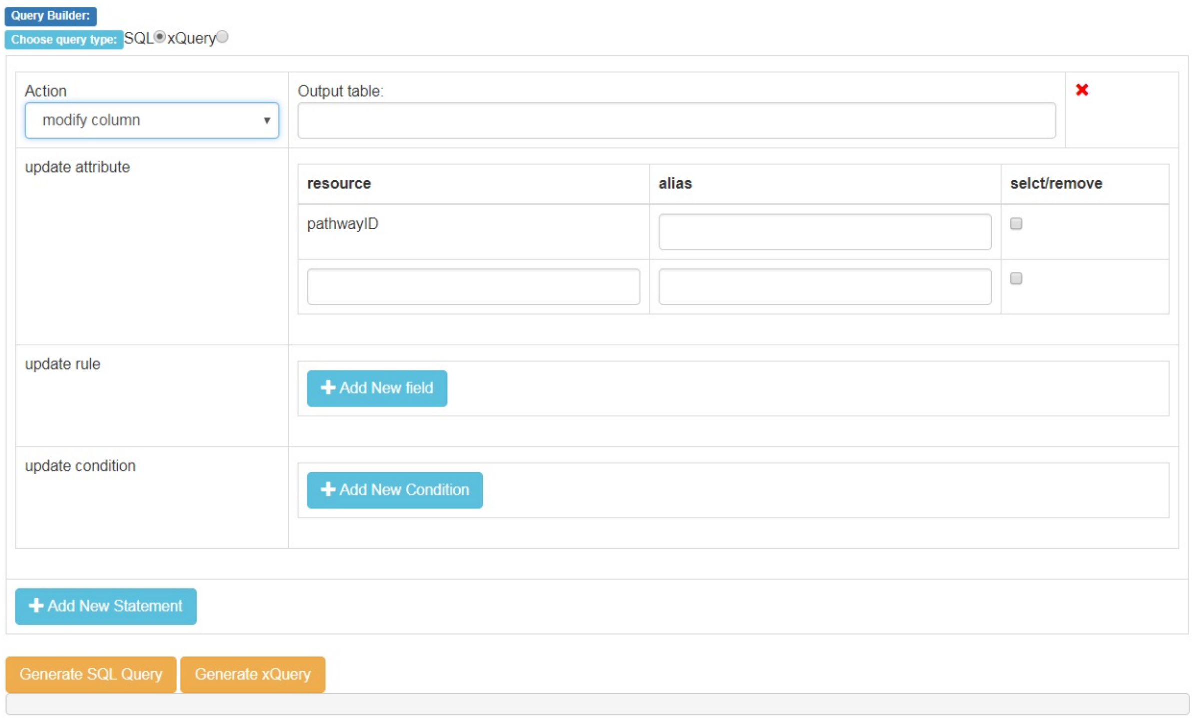
(a) Visual Query Builder: Remove Rows



```
else $node};
let $gene.gene := doc("gene.xml")//gene
let $cla := $gene.gene/..
let $nodes := doc("gene.xml")//*[name() = $cla/name()]
let $no := for $row in $nodes
where ( $row/gene=217  )
return $row
let $res := functx:remove-elements-deep(doc("gene.xml"), $no)
let $source-doc := <resultsets>
```

(b) Visual Query Builder: Remove Rows Script

Figure 5.13: Examples of Remove Rows



Choose query type: [✓ SQL] xQuery

Action	Output table:
modify column	

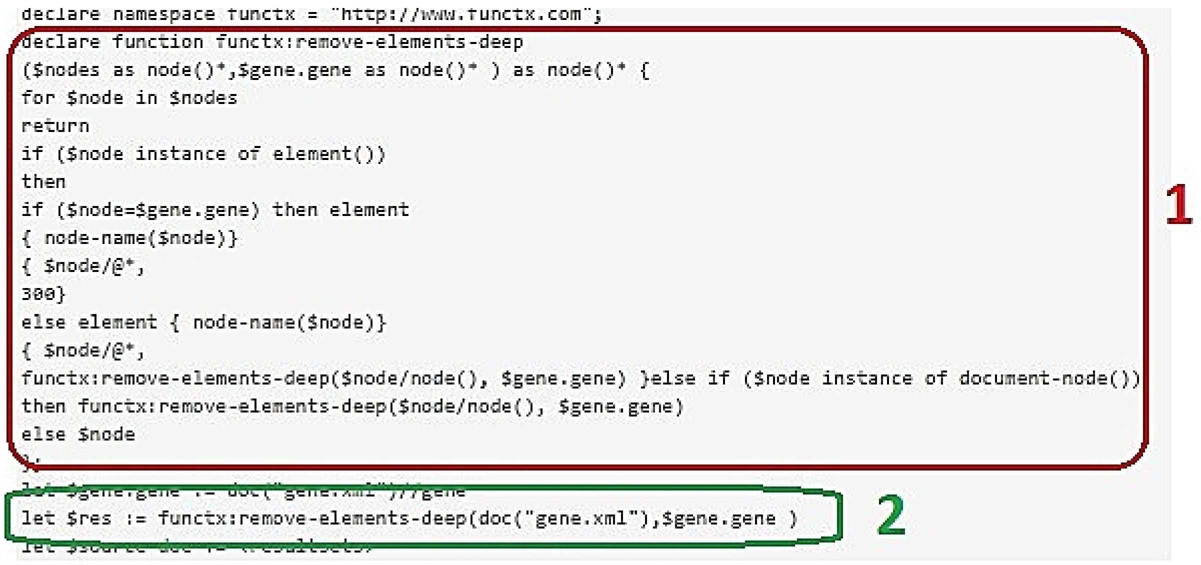
update attribute

resource	alias	selct/remove
pathwayID		☐
		☐

update rule

update condition

(a) Visual Query Builder:Modify Columns



```
declare namespace functx = "http://www.functx.com";
declare function functx:remove-elements-deep
($nodes as node()*,$gene.gene as node()* ) as node()* {
for $node in $nodes
return
if ($node instance of element())
then
if ($node=$gene.gene) then element
{ node-name($node)}
{ $node/@*,
300}
else element { node-name($node)}
{ $node/@*,
functx:remove-elements-deep($node/node(), $gene.gene) }else if ($node instance of document-node())
then functx:remove-elements-deep($node/node(), $gene.gene)
else $node
};
let $gene.gene := doc("gene.xml")//gene
let $res := functx:remove-elements-deep(doc("gene.xml"),$gene.gene )
let $source_doc := <resultset>
```

(b) Visual Query Builder:Modify Columns Script

Figure 5.14: Examples of Modify Columns

of replacing gene value 217 to 300 in gene.xml. The two highlight blocks are our updateDeep function, and the statement calls the function at the end.

Algorithm 7: Modify Columns to XQuery

```
   input :
   $resName : resourcename$
   $A$: update attributes $\alpha_1 \ldots \alpha_k$ as list
   $R$: update rules $r_1 \ldots r_m$ as list
   $U$: update conditions $conU_1 \ldots conU_n$ as list
   output: s
   /* define function                                  */
1  s $\leftarrow$ "declare function updateDeep(nodes, A) {
2    for $node \in nodes$ return
3    if (U) then $R$
4    else updateDeep(nodes, A)"
   /* call function                                    */
5  s $\leftarrow$ "updateDeep(nodes, A)"
6  return s
```

Chapter 6: VisFlow Evaluations

6.1 VisFlow Features for Usability

VisFlow is a highly abstract visual language based workflow management system that improves usability by its unique features. It helps users quickly and efficiently complete their tasks better than other workflow systems provides. Usability is the most important concept in designing a successful workflow management system that has been widely used. International Standards Organization (ISO) / International Electrotechnical Commission (IEC) defines usability as follows:

Definition 3. *The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. [55]*

Simply speaking, usability typically focuses on how easily and efficiently users can complete their tasks. Domain scientists are a group of people who have a thorough knowledge of their domains but may not familiar with computer science cutting-edge technologies. Domain scientists mainly use scientific workflow systems as a tool to help them analyze distributed datasets across the world. With the massive explosion of data generation and the trend of putting data on the Internet, whenever domain scientists need to do their research and experiment, like explore, analyze and visualize the data, they need computer scientists' help to retrieve, apply analytics algorithms and produce the meaningful results, which may contain a large number of different tasks. When using a computer program or application, domain scientists are more concerned with the abstract steps of their experiments, not lower level application execution issues. The less time the domain scientists spend on learning those tools, the more productive they will be in their researches. If the tool helps them construct and configure these components and avoid writing programming code, it will be more valuable to them.

VisFlow provides auto data integration, ad hoc debugging, and auto parallel execution to help domain scientists compositing workflows. Data integration is a common but complex task the domain scientists encounter in their daily life when they construct a workflow. Providing a data integration interface to help them to solve the heterogeneous data problem reduces their time composing workflow systems. In VisFlow, *fusion* and *combine* are two auto data integration icons aim to merge two different data resources. When submitting queries to web services, VisFlow has auto-suggestions to pick up the attributes and generates the queries. Users are also free of file type transformations among CSV, XML, and relational table by VisFlow auto transform algorithm. Users only need to think of the meaning of the data itself instead of worry about the compatibility of different file types. With all these features together, VisFlow handles data integration smoothly.

Debugging is an integral part of workflow development. Debugging resolves the existing and potential errors in the workflow that can cause the workflow to produce wrong results, stop executing, or crash. When more and more components are added to the workflow, debugging becomes harder and harder, as small changes may cause more unexpected behaviors to occur. VisFlow supports ad hoc debugging, which allows users to generate a partial workflow and execute it to validate the results. This feature allows users to design and develop the workflow iteratively and on the fly. When the whole workflow completes, they don't need to worry about the validation of the workflow.

Parallel execution allows users to execute a workflow in parallel to reduce the total running time. VisFlow supports auto parallel execution without user configuration which doesn't require users to know the concepts of the parallel execution and be experts in the related area to optimize the performance.

With these features, VisFlow is designed as a system that improves the usability by reducing workflow composing time and execution time. In the next two sections, we first provide a discussion about the construction of a workflow in VisFlow compared to the Taverna workflow system regarding workflow compositions. After that, another example is provided to show the time evaluation comparison between a VisFLow workflow and a published workflow pipeline.

6.2 VisFlow vs. Taverna: Composition Comparison

6.2.1 An Example: Entrez to KEGG Image

Taverna is a robust workflow system for bioinformatics design, and it contains a vast repository of public workflows stored on a website, `www.myexperiment.org` . Figure 6.1 shows a Taverna workflow example named entreztoKeggImage[1]. This workflow accepts a list of Entrez Gene IDs, finds the defined KEGG pathway that involves each gene and displays the pathway diagram image and description. This workflow generates three different files as results. Boxes in this workflow with brown colors are user-defined Java code segments. For example, users wrote a Java program in node 1 to add a prefix string "ncbi-geneid:" to each gene id.

The blue boxes (nodes 2, 4, 6 and 7) are web service calls to the KEGG database. Node 2 converts NCBI IDs to HSA IDs, and node 4 submits a list of HSA IDs to retrieve the pathways back, while node 6 gets the pathway entry descriptions, and lastly node 7 downloads the pathway images. For each web service node, users need to provide the URL and select the header information each time they plan to use it.

[1]`https://www.myexperiment.org/workflows/4453.html`

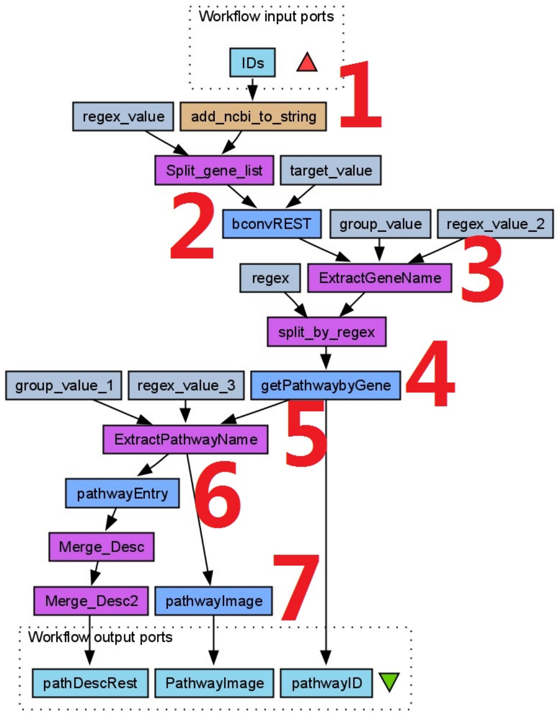


Figure 6.1: Entrez to KEGG Image

The purple boxes (nodes 3 and 5) are predefined Taverna functions for standard function calls. Node 3 splits the results text streams based on tab character and extracts the HSA IDs returned by the web service, while node 5 selects the first matched entry from the returned pathways. These predefined functions are written in Java, which contains standard functionalities like split strings, a select array of elements based on the index, etc.

Configuring blue boxes is relatively simple enough by providing required inputs. However, working with brown and purple boxes is dependent on the tasks the experiments expected. Sometimes the configurations are complex and need a large section of custom coding using Java, which requires programming experiences as shown in the right window of node 1.

The whole workflow accesses four web services in the KEGG database, and it contains one input file. The workflow in Figure 4.1 shows the Taverna entreztoKeggImage workflow example implemented using VisFlow. This figure illustrates the entire organization of this workflow. The brown window contains the workflow graph with one *procedure* icon, one *data* icon, four *adapter* icons, one *combine* icon and one *terminal* icon. The figure above the brown box is the nested workflow imported by a *procedure* icon labeled node 1, and the figure under the brown box is the results after the workflow finished execution.

Node 1 is a *procedure* icon that imports pre-stored modules saved via the *export* button. This type of icon is offered to reduce the size of the workflow and reuse repeated workflow steps when the experiments are complex. Node 1 contains a sub-workflow module shown above named KEGG Entrez to Pathway which includes two *data* icons and three *adapter* icons, nodes 8 through 10. Users use module names to filter out the modules they stored before. The show button is provided to view the modules. A back button will be shown on the top right of the interface to go back to the current workflow. KEGG Entrez to Pathway module generates a single SQL table named pathwayID.

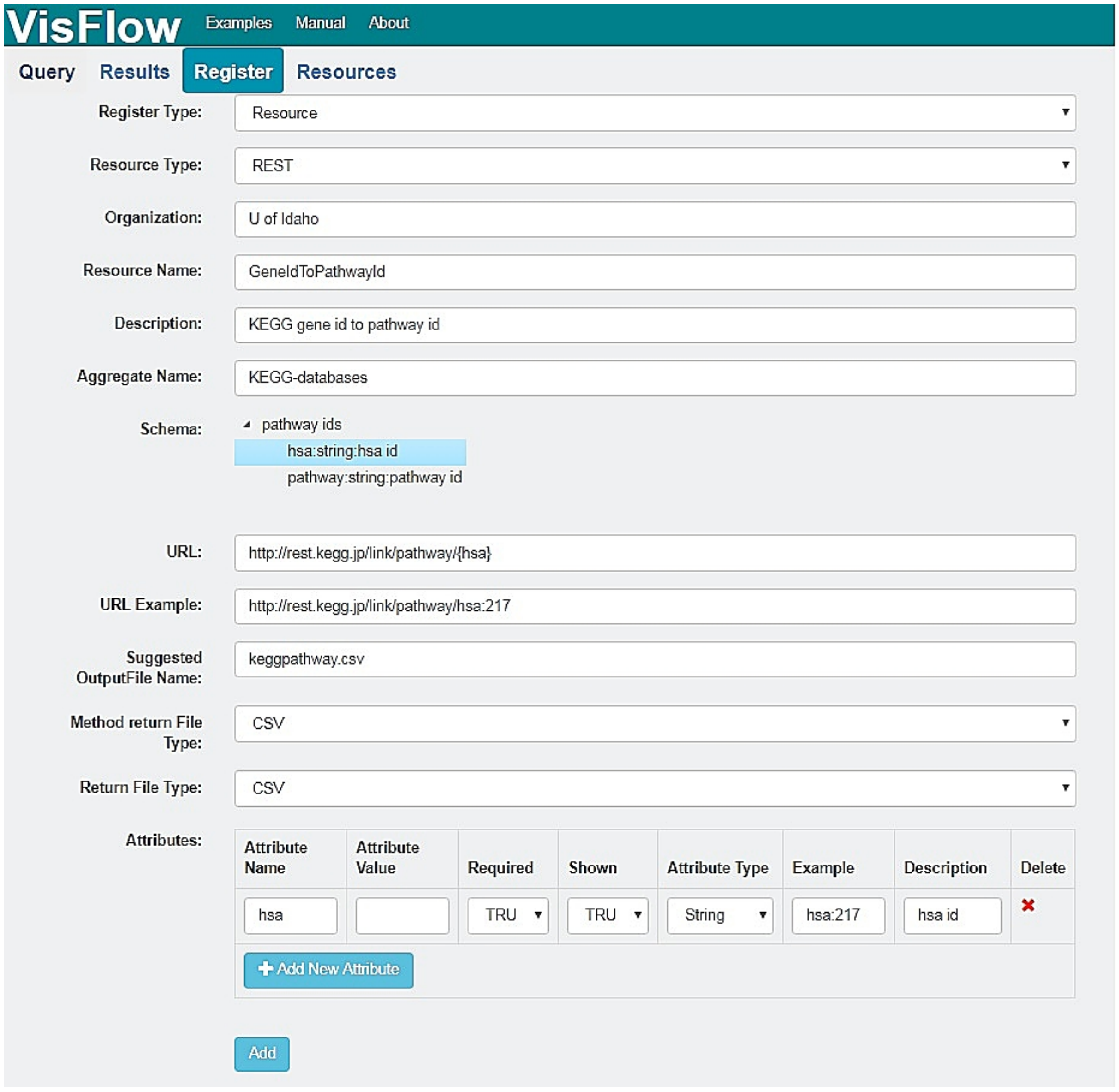
In this nested workflow, unlike Taverna, VisFlow requires that all the resources be registered into the VisFlow system via the register page before they can be used in this or other future workflows. Figure 6.2 shows the way to register the web service endpoint `http://rest.kegg.jp/link/pathway/`. Most of the information is straightforward; the only computer science concepts required of users are the HTTP query types and the supported VisFlow data types. These concepts are easy to understand.

The schema field contains the schema information to describe the return metadata for later usage in the workflow. The schema is a tree structure of the format $\{attributename\} : \{attributetype\} : \{attributedescription\}$. The URL field contains an attribute surrounded by curly brackets – it is the parameter which is required by the endpoint, as defined by the attribute table at the bottom. The Method return File type is the response page that the endpoint returns, where the return file type and suggested output file name defines the return file name.

VisFlow has predefined wrappers to extract the content from the response page and convert it to corresponding file type. The matcher and wrapper fields will be automatically selected based on the resources information; users are allowed to modify this as needed. All users need to do is select the web service required inputs from and fill out the attribute name if they are from previous files. Figure 6.3 shows the information to register the input file for the whole workflow, which is a CSV file named entrez.csv and the information required is basically the same.

After registering all the resources, users need to use a *data* or *library* icon to import them into the current workflow. Users can use the update page from a *data* icon to import a resource to the current workflow, as shown in Figure 6.4, where the submitted schema KEGG with HSA attributes submitted to VisFlow and the matched resources highlighted with the attributes are returned to allow users to select.

Node 8 in Figure 6.5 shows the SQL script generated by the visual query builder shown in Figure 6.6. Configuring the visual query builder to generate the SQL query is easy – users only need to select the attribute by replacing the substring function template and selecting the data resource. Then the visual query builder automatically generates the query for them. In this step, no programming knowledge background is required.

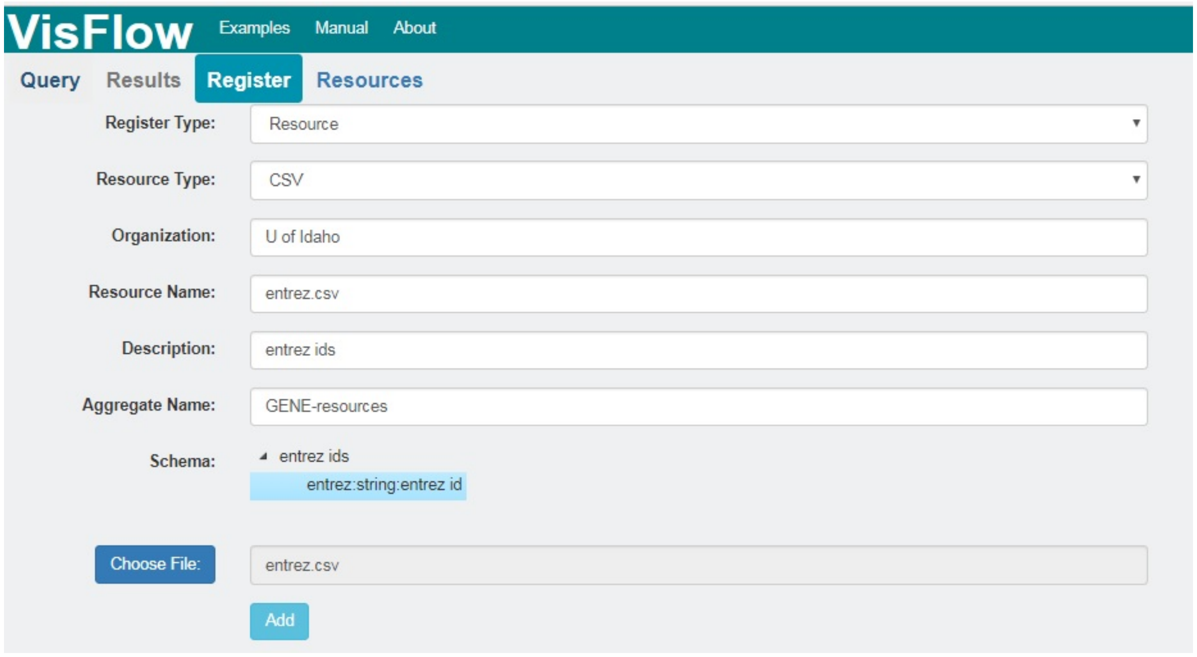


Register Type:	Resource
Resource Type:	REST
Organization:	U of Idaho
Resource Name:	GeneIdToPathwayId
Description:	KEGG gene id to pathway id
Aggregate Name:	KEGG-databases
Schema:	pathway ids hsa:string:hsa id pathway:string:pathway id
URL:	http://rest.kegg.jp/link/pathway/{hsa}
URL Example:	http://rest.kegg.jp/link/pathway/hsa:217
Suggested OutputFile Name:	keggpathway.csv
Method return File Type:	CSV
Return File Type:	CSV

Attributes:

Attribute Name	Attribute Value	Required	Shown	Attribute Type	Example	Description	Delete
hsa		TRU	TRU	String	hsa:217	hsa id	✖

Figure 6.2: Resource Register for GeneIdToPathwayid



Register Type: Resource

Resource Type: CSV

Organization: U of Idaho

Resource Name: entrez.csv

Description: entrez ids

Aggregate Name: GENE-resources

Schema: entrez ids
entrez:string:entrez id

Choose File: entrez.csv

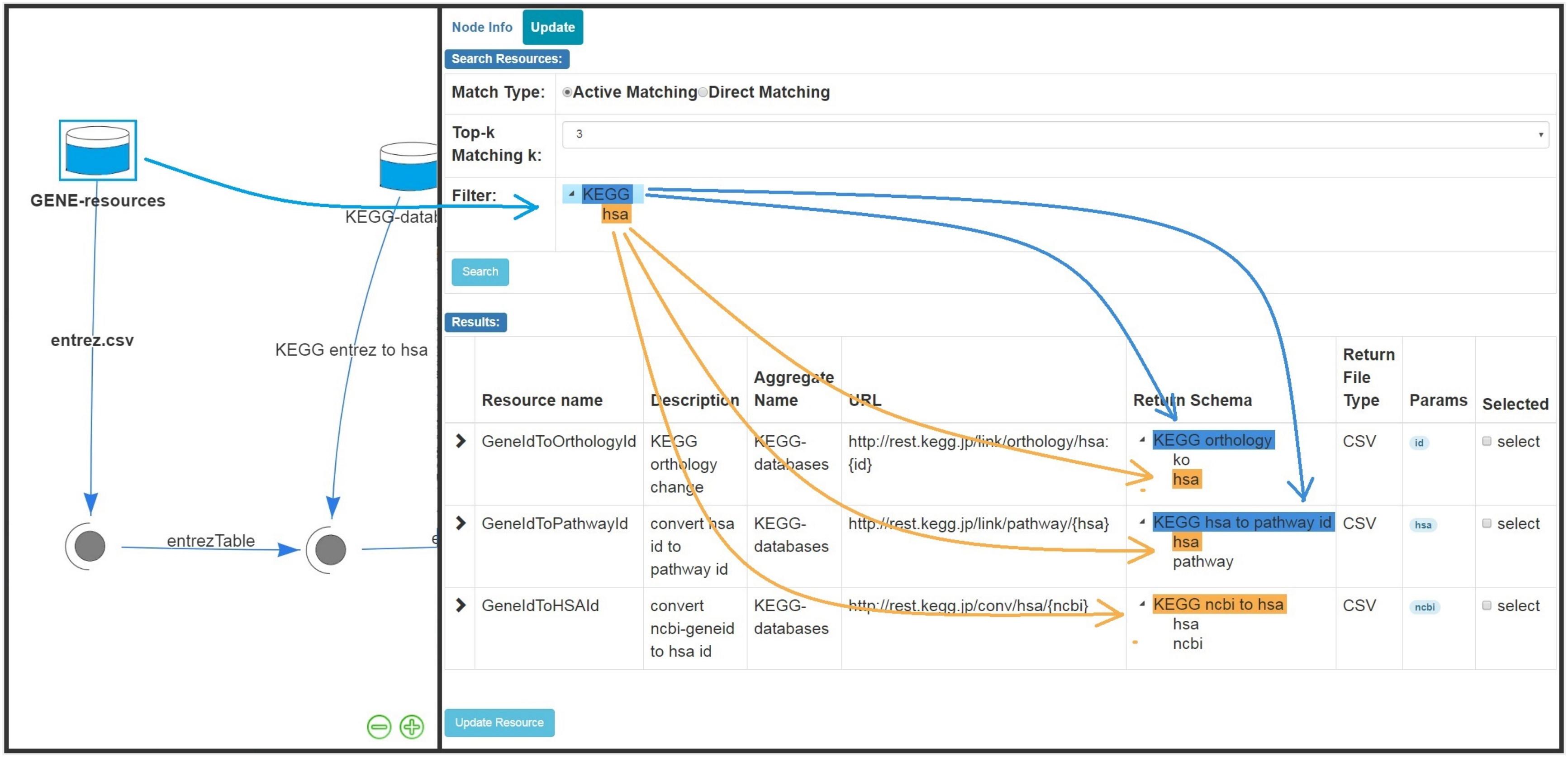
Figure 6.3: Resource Register for Entrez.csv

Nodes 9 and 10 submit the Entrez IDs to the KEGG database to get a list of pathways back. These two nodes have the same functionality as nodes 2 and 4 in the Taverna workflow, respectively. Figures 6.7 and 6.8 shows the configuration for nodes 9 and 10, respectively. Node 9 retrieves a list of HSA IDs, while node 10 retrieves a list of pathways. For example, the only configuration in node 10 users need to do is select the input from the data resource passed from the previous node, and VisFlow is smart enough to select the proper wrapper to collect the returned data from the returned page. Other information is predefined when users registered this web service.

After the pathwayID is generated and imported via node 1, this file is passed to nodes 2 and 3 to remove the prefix ‘path:’ and extract HSA corresponding pathway names, respectively. These two nodes use the visual query builder to generate the SQL query. Figures 6.9 and 6.10 show the configuration for node 2 and Figure 6.11 and Figure 6.12 show the configuration for node 3. The pathway names are submitted to KEGG Pathway Entry to get the pathway description back in node 4 (shown in Figure 6.13), and the configuration is similar to node 10.

Figures 6.14 and 6.15 shows Node 5 which merged the returned data with the submitted data into one table. This was done using an SQL script that was automatically generated by the visual query builder.

Node 6 is a *combine* icon that automatically merges two tables based on their key pairs to generate



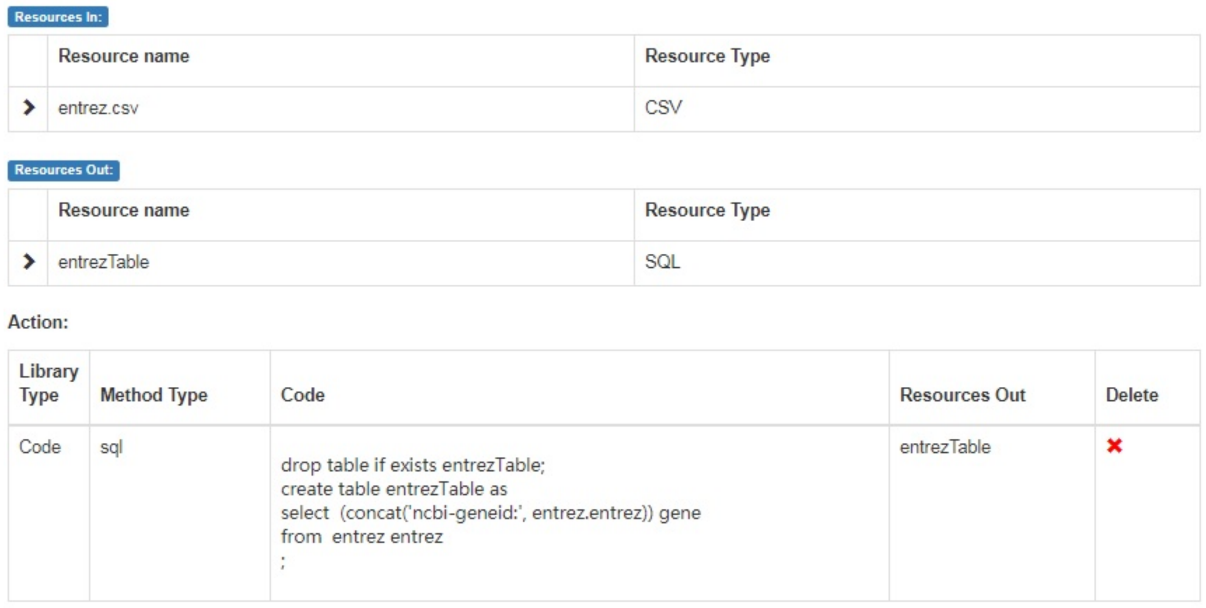
Match Type: ◉Active Matching ○Direct Matching

Top-k Matching k: 3

Filter: KEGG
hsa

	Resource name	Description	Aggregate Name	URL	Return Schema	Return File Type	Params	Selected
›	GeneIdToOrthologyId	KEGG orthology change	KEGG-databases	http://rest.kegg.jp/link/orthology/hsa:{id}	KEGG orthology ko hsa	CSV	id	☐ select
›	GeneIdToPathwayId	convert hsa id to pathway id	KEGG-databases	http://rest.kegg.jp/link/pathway/{hsa}	KEGG hsa to pathway id hsa pathway	CSV	hsa	☐ select
›	GeneIdToHSAId	convert ncbi-geneid to hsa id	KEGG-databases	http://rest.kegg.jp/conv/hsa/{ncbi}	KEGG ncbi to hsa hsa ncbi	CSV	ncbi	☐ select

Figure 6.4: Resource Search


Resources In:

	Resource name	Resource Type
>	entrez.csv	CSV

Resources Out:

	Resource name	Resource Type
>	entrezTable	SQL

Action:

Library Type	Method Type	Code	Resources Out	Delete
Code	sql	drop table if exists entrezTable; create table entrezTable as select (concat('ncbi-geneid:', entrez.entrez)) gene from entrez entrez ;	entrezTable	✖

Figure 6.5: Node 8: Add Prefix 'ncbi-id:'

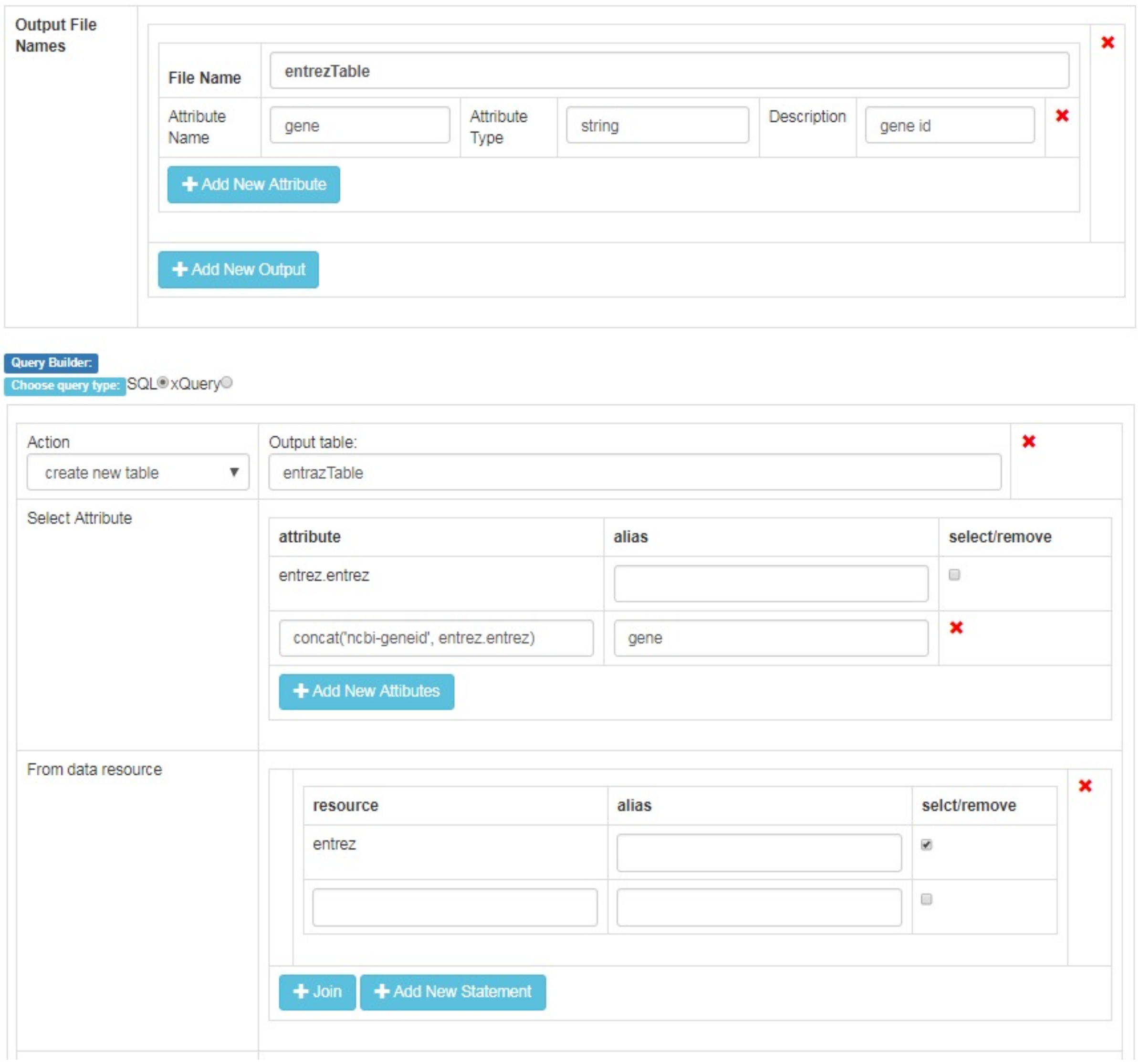
the final result. Figure 6.16 is the configuration interface for a *combine* icon; users are allowed to select the schema matcher and key identifier algorithms. The *combine* icon has a "generate key pairs" button to suggests the possible key pairs in two tables, and users are allowed to overwrite the suggested key pairs in the returned table.

Node 7, shown in Figure 6.17, is the *terminal* icon that contains predefined printing rules to generate the results. VisFlow has integrated some popular data visualization tools like Google Maps and Gplate Maps to help users explore the data resources. In this example, pathways are better viewed as an image instead of just strings, as VisFlow has built-in functions that automatically identify the pathway field, go to the KEGG database to retrieve the images, and then show it to users.

In contrast to the Taverna workflow, VisFlow has three different files. We merged all three results into one table, as shown in the bottom of this workflow. All the pathways with their images are generated as a link which allows users to view it in the right window.

6.2.2 Discussion

For biologists, the most crucial aspect of their experiment's design is how to design their experiments in an efficient way using a software application which provides high usability. Biologists are a group of experts who have extensive knowledge in bioinformatics but may lack knowledge of computer skills. They may not be familiar with programming, and they may not know how to submit queries through



Output File Names

File Name	entrezTable				
Attribute Name	gene	Attribute Type	string	Description	gene id

Choose query type: [✓ SQL] xQuery

Action	create new table
Output table:	entrazTable

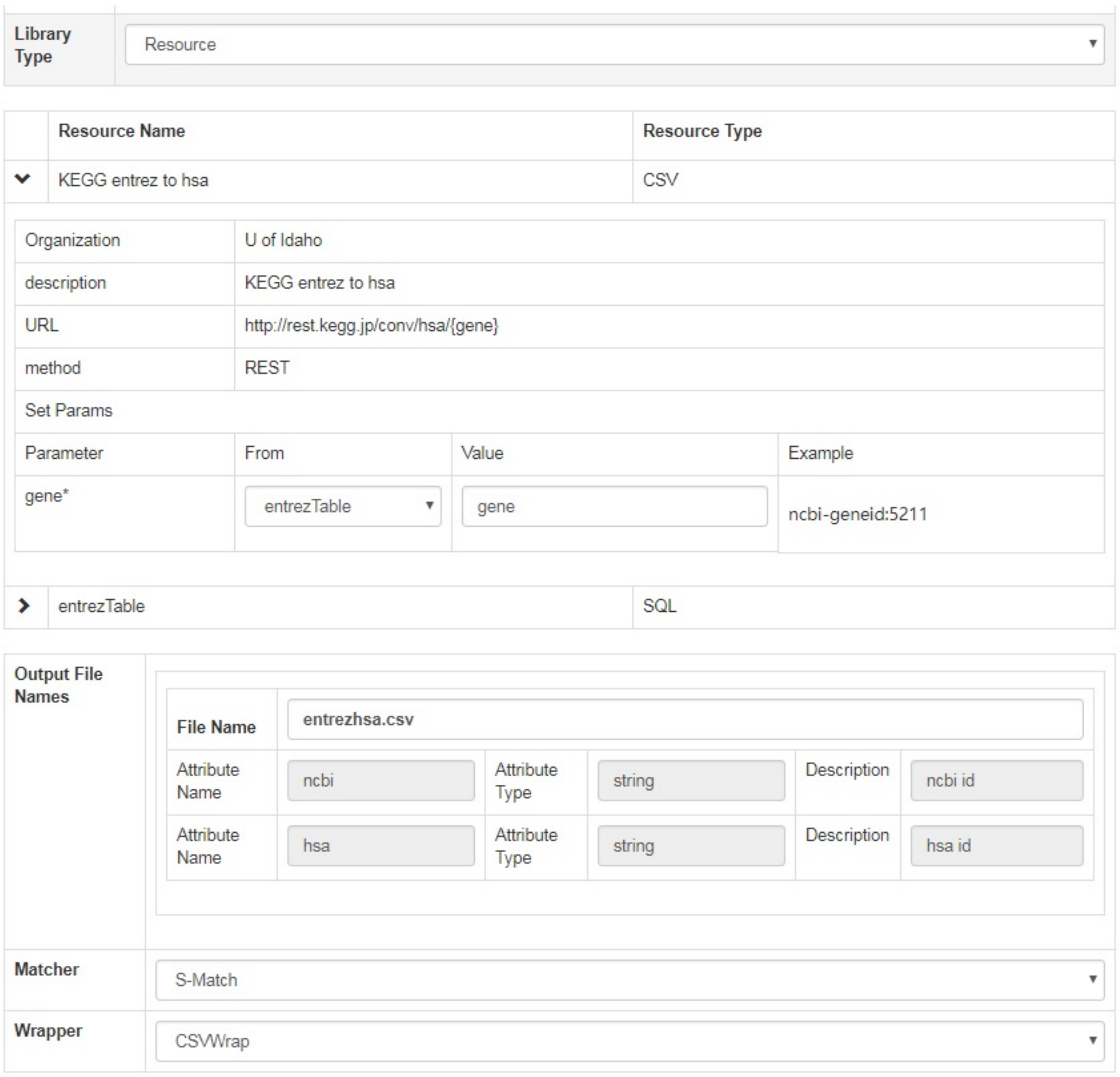
Select Attribute

attribute	alias	select/remove
entrez.entrez		☐
concat('ncbi-geneid', entrez.entrez)	gene	

From data resource

resource	alias	selct/remove
entrez		✓
		☐

Figure 6.6: Node 8: Visual Query Builder for Add Prefix 'ncbi-id:'



Library Type	Resource

	Resource Name	**Resource Type**
⌄	KEGG entrez to hsa	CSV

Organization	U of Idaho		
description	KEGG entrez to hsa		
URL	http://rest.kegg.jp/conv/hsa/{gene}		
method	REST		
Set Params			
Parameter	From	Value	Example
gene*	entrezTable	gene	ncbi-geneid:5211

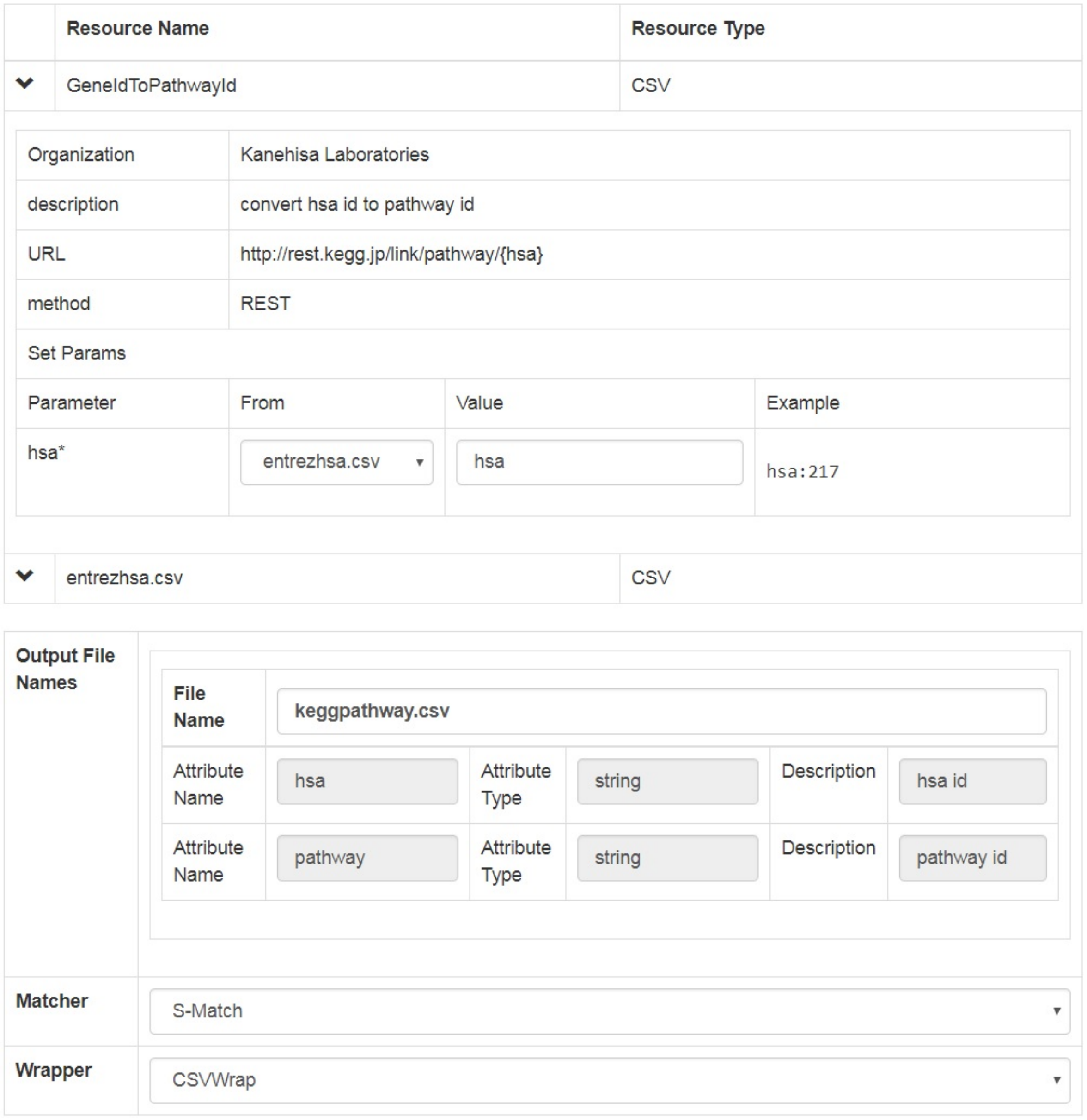
›	entrezTable	SQL

Output File Names

File Name	**entrezhsa.csv**				
Attribute Name	ncbi	Attribute Type	string	Description	ncbi id
Attribute Name	hsa	Attribute Type	string	Description	hsa id

Matcher	S-Match
Wrapper	CSVWrap

Figure 6.7: Node 9: Web Service Call for KEGG Entrez to HSA



	Resource Name	Resource Type
❯	GeneIdToPathwayId	CSV

Organization	Kanehisa Laboratories		
description	convert hsa id to pathway id		
URL	http://rest.kegg.jp/link/pathway/{hsa}		
method	REST		
Set Params			
Parameter	From	Value	Example
hsa*	entrezhsa.csv	hsa	`hsa:217`

❯	entrezhsa.csv	CSV

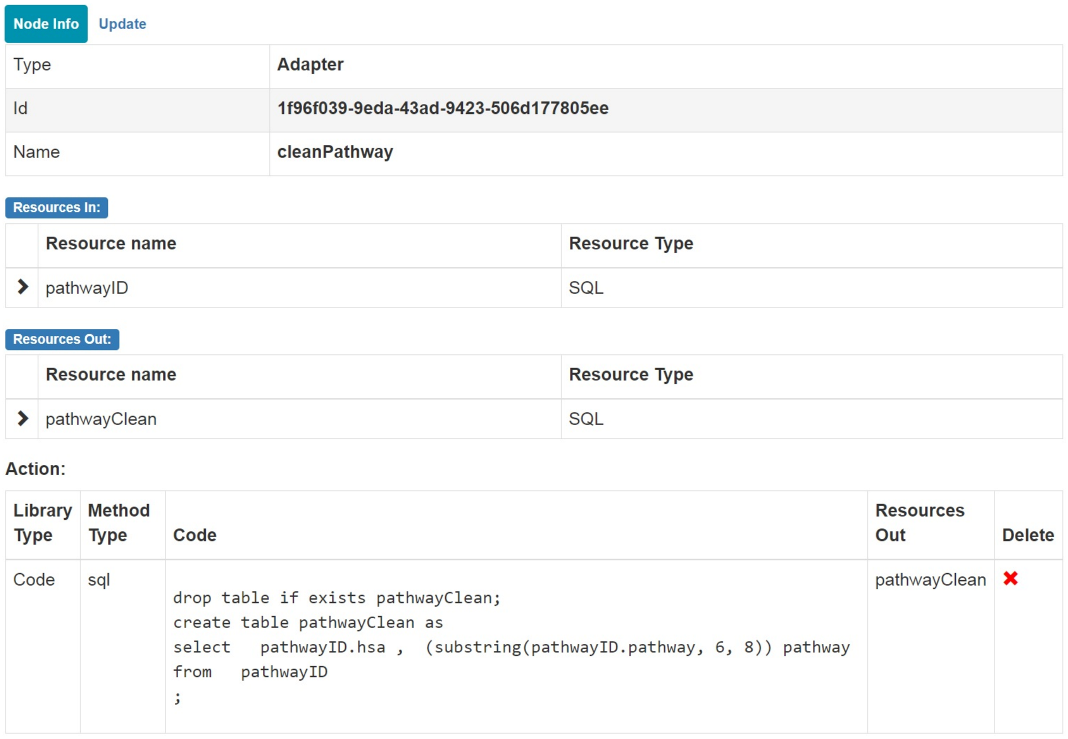
Output File Names

File Name	keggpathway.csv				
Attribute Name	hsa	Attribute Type	string	Description	hsa id
Attribute Name	pathway	Attribute Type	string	Description	pathway id

Matcher: S-Match

Wrapper: CSVWrap

Figure 6.8: Node 10: Web Service Call for GeneIdToPathwayId


Node Info Update

Type	**Adapter**
Id	**1f96f039-9eda-43ad-9423-506d177805ee**
Name	**cleanPathway**

Resources In:

	Resource name	Resource Type
›	pathwayID	SQL

Resources Out:

	Resource name	Resource Type
›	pathwayClean	SQL

Action:

Library Type	Method Type	Code	Resources Out	Delete
Code	sql	`drop table if exists pathwayClean; create table pathwayClean as select pathwayID.hsa , (substring(pathwayID.pathway, 6, 8)) pathway from pathwayID ;`	pathwayClean	✖

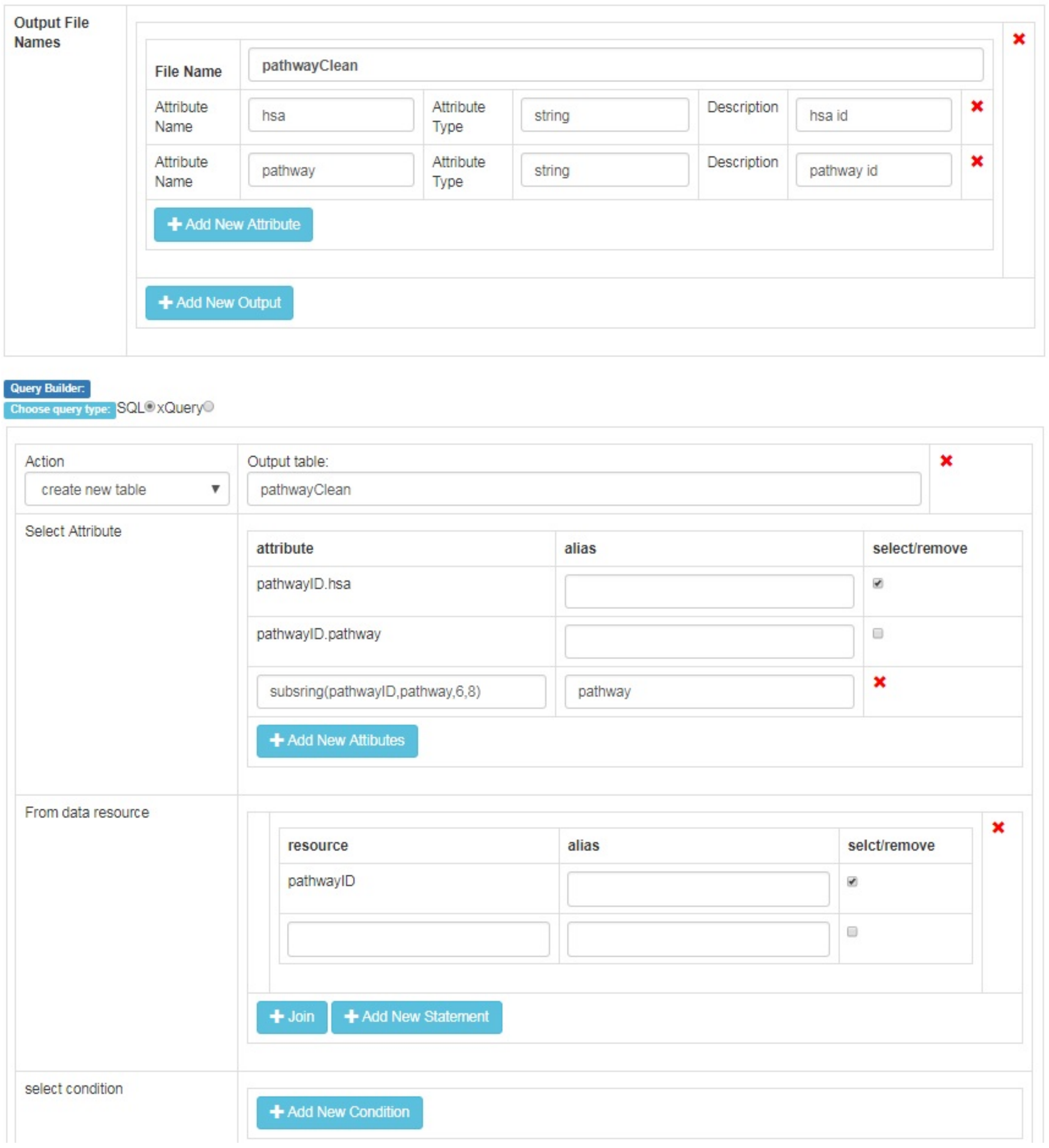
```
drop table if exists pathwayClean;
create table pathwayClean as
select   pathwayID.hsa ,  (substring(pathwayID.pathway, 6, 8)) pathway
from   pathwayID
;
```

Figure 6.9: Node 2: Adapter String Cleaning

web services. Providing an application that hides the computer science concepts and programming will reduce the amount of time biologists expend in order to design and compose their workflow, thereby improving their efficiency.

By analyzing the example presented here, it can be seen that VisFlow has a resources management system that allows all the resources (especially web services) to be reused anywhere in the workflow, while the Taverna system requires users to type the same information again to make it reusable in the workflow. In addition, there is minimal programming background required to use VisFlow. To utilize competent functionalities in the Taverna workflow, users have to learn the Java programming language in order to manipulate data appropriately for their users, while VisFlow provides a Visual Query Builder interface to take advantage of SQL script and XQuery script. VisFlow contains schema information for every resource, and it offers auto suggestions and error check features. VisFlow has predefined data visualization rules to show meaningful images without user interactions.

VisFlow is a web-based system, while Taverna is desktop based. One of the most significant advantages of using a web-based application is that it is platform independent and has no dedicated hardware requirements, while the desktop version have hardware restrictions and are supported on limited types of operating systems. The web-based version also offers computers with limited hardware capabilities



Output File Names	File Name	pathwayClean				
	Attribute Name	hsa	Attribute Type	string	Description	hsa id
	Attribute Name	pathway	Attribute Type	string	Description	pathway id

Query Builder: Choose query type: SQL [✓ ●] xQuery ○

Action	create new table
Output table:	pathwayClean

Select Attribute

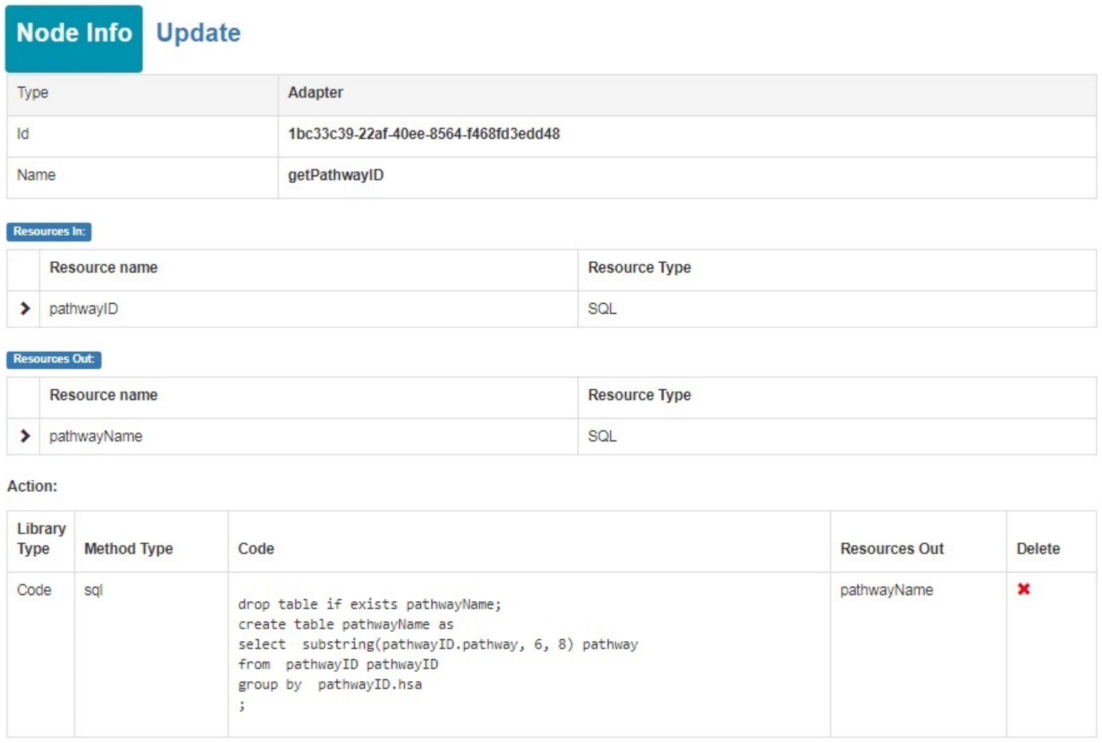
attribute	alias	select/remove
pathwayID.hsa		✓
pathwayID.pathway		☐
subsring(pathwayID,pathway,6,8)	pathway	

From data resource

resource	alias	selct/remove
pathwayID		✓
		☐

select condition

Figure 6.10: Node 2: Visual Query Builder for Cleaning String



Node Info Update

Type	Adapter
Id	1bc33c39-22af-40ee-8564-f468fd3edd48
Name	getPathwayID

Resources In:

	Resource name	Resource Type
>	pathwayID	SQL

Resources Out:

	Resource name	Resource Type
>	pathwayName	SQL

Action:

Library Type	Method Type	Code	Resources Out	Delete
Code	sql	(see below)	pathwayName	✖

```
drop table if exists pathwayName;
create table pathwayName as
select  substring(pathwayID.pathway, 6, 8) pathway
from  pathwayID pathwayID
group by  pathwayID.hsa
;
```

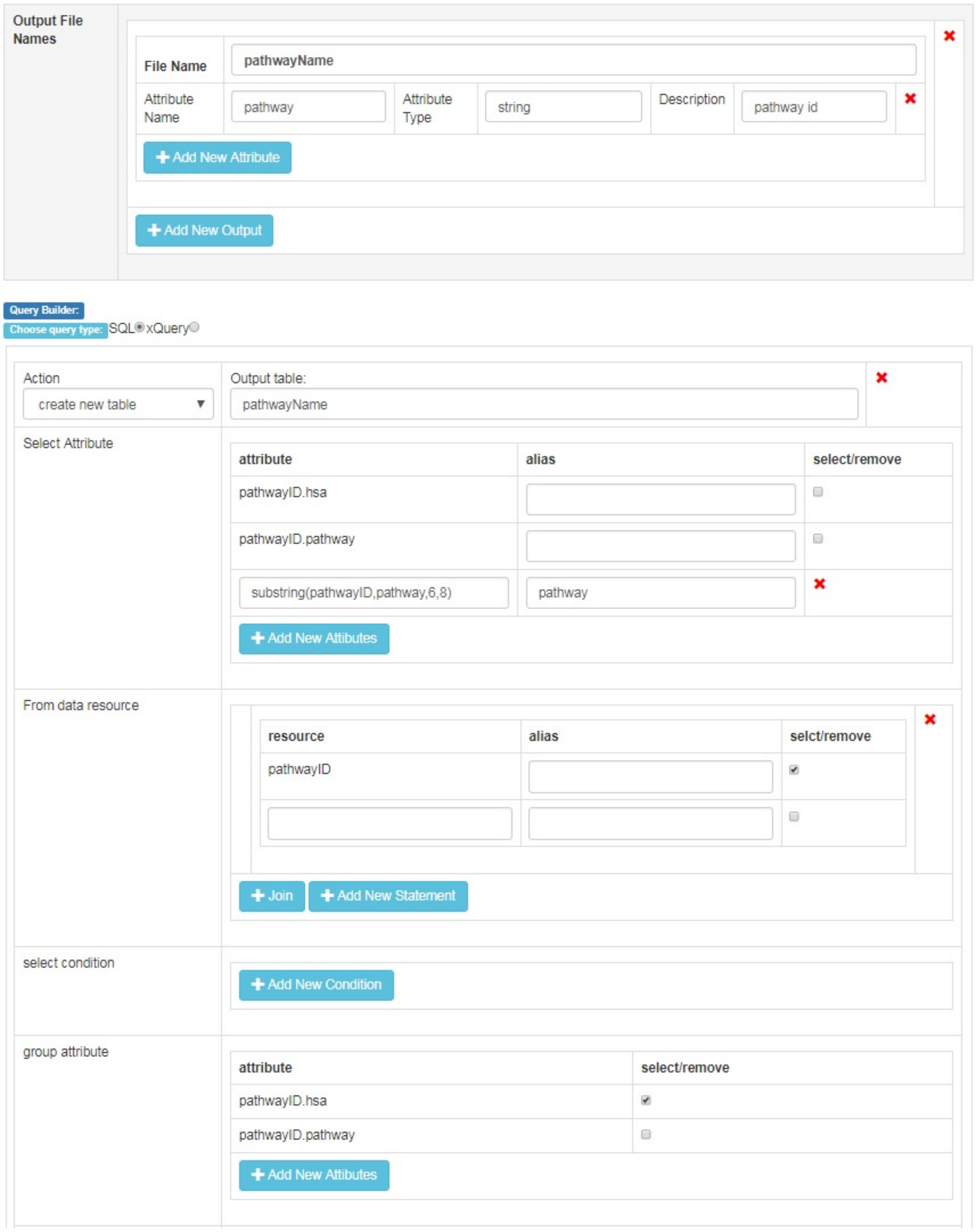
Figure 6.11: Node 3: Adapter Configuration for Pathway Name

to run complex tasks. Another advantage of a web-based application is users can skip the installation steps, which may require a variety of libraries and complicated steps to make it successfully installed on the local computer. Web applications make it convenient for users as they can access it anywhere in the world where there is an Internet connection. Meanwhile, the desktop version applications confine the location where users can access them. Unlike the desktop application, users are in charge of everything.

6.3 Workflow Compositing Time Evaluation

Figure 6.18 shows an example of Mineral 3D visualization in Ma's paper [97]. VisFlow has implemented the same workflow as Ma did. In order to generate a data visualization as they purposed, there are many complex steps needed in order to show a list of elements as a 3D cube. The first step of this experiment is that they go to Mineral Evolution Database[2] and collect a list of tables of elements that list how many of those minerals contain a specific pair of the rock-forming elements from the periodic table. After they cleaned the data and appended them as a big CSV file, they designed an application to visualize this table as a 3D cube using the D3.js library which allows users to select elements and

[2] `https://www.geo.arizona.edu/~jgolden/klee.html`


Output File Names

File Name	pathwayName				
Attribute Name	pathway	Attribute Type	string	Description	pathway id

Query Builder:

Choose query type: SQL [✓ ●] xQuery ○

Action	create new table
Output table:	pathwayName

Select Attribute

attribute	alias	select/remove
pathwayID.hsa		☐
pathwayID.pathway		☐
substring(pathwayID,pathway,6,8)	pathway	

From data resource

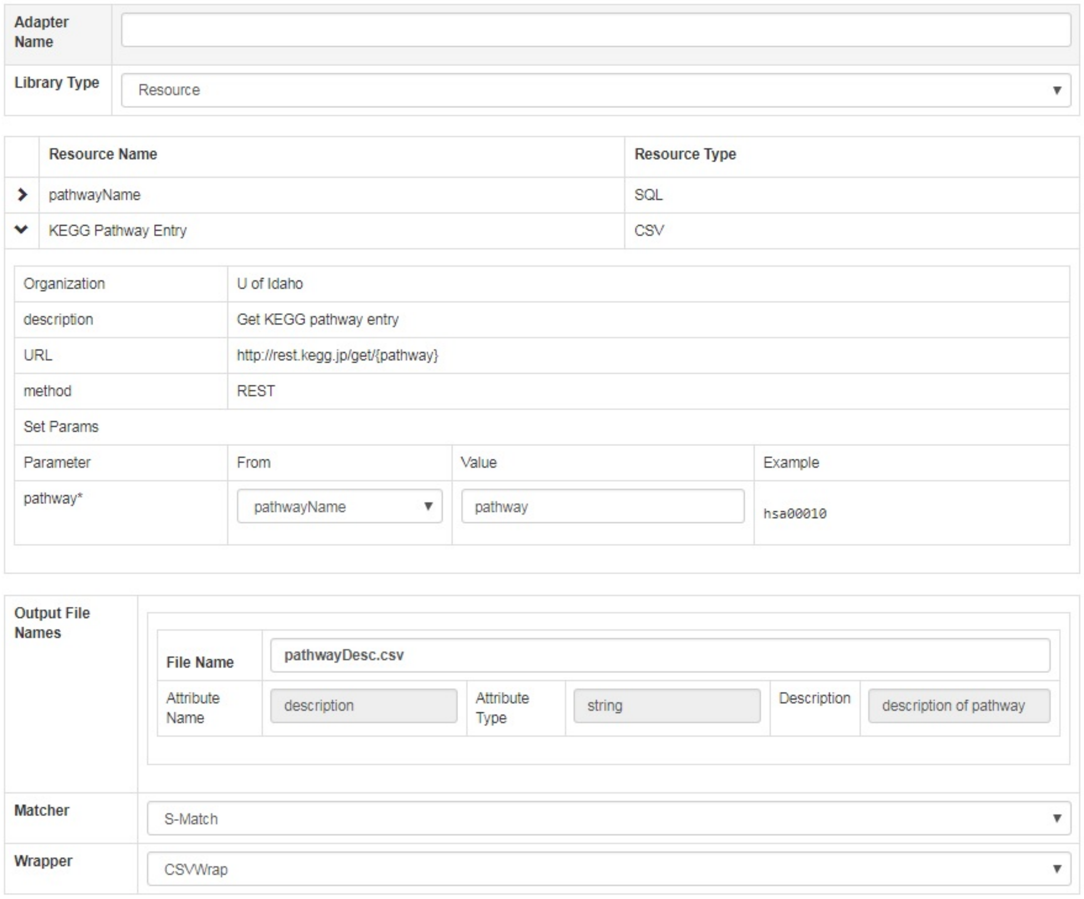
resource	alias	selct/remove
pathwayID		✓
		☐

select condition

group attribute

attribute	select/remove
pathwayID.hsa	✓
pathwayID.pathway	☐

Figure 6.12: Node 3: Visual Query Builder for Pathway Name



Adapter Name	
Library Type	Resource

	Resource Name	**Resource Type**
›	pathwayName	SQL
⌄	KEGG Pathway Entry	CSV

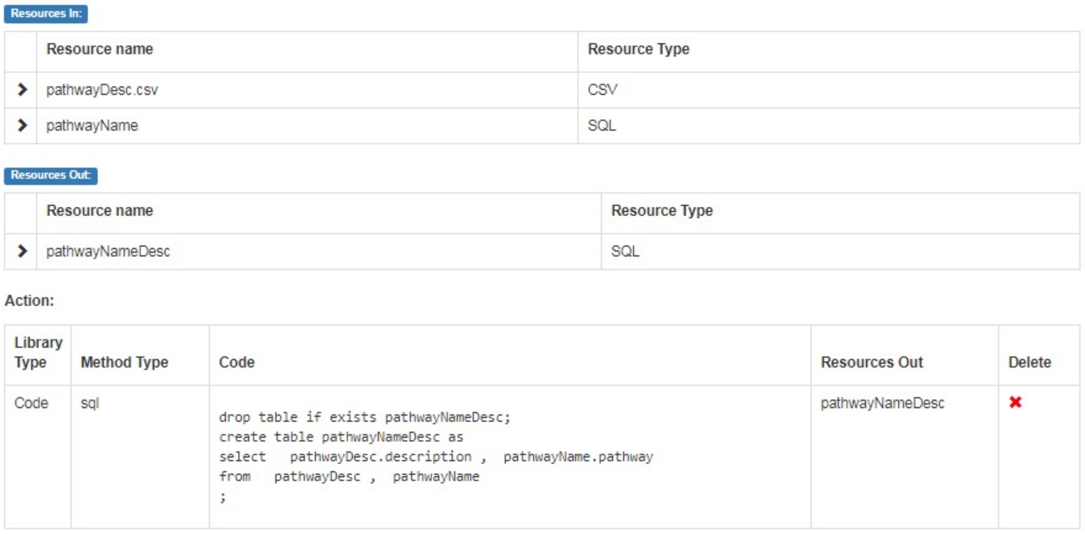
Organization	U of Idaho		
description	Get KEGG pathway entry		
URL	http://rest.kegg.jp/get/{pathway}		
method	REST		
Set Params			
Parameter	From	Value	Example
pathway*	pathwayName	pathway	hsa00010

Output File Names

File Name	pathwayDesc.csv				
Attribute Name	description	Attribute Type	string	Description	description of pathway

Matcher	S-Match
Wrapper	CSVWrap

Figure 6.13: Node 4: Get Pathway Entry



Resources In:

	Resource name	**Resource Type**
›	pathwayDesc.csv	CSV
›	pathwayName	SQL

Resources Out:

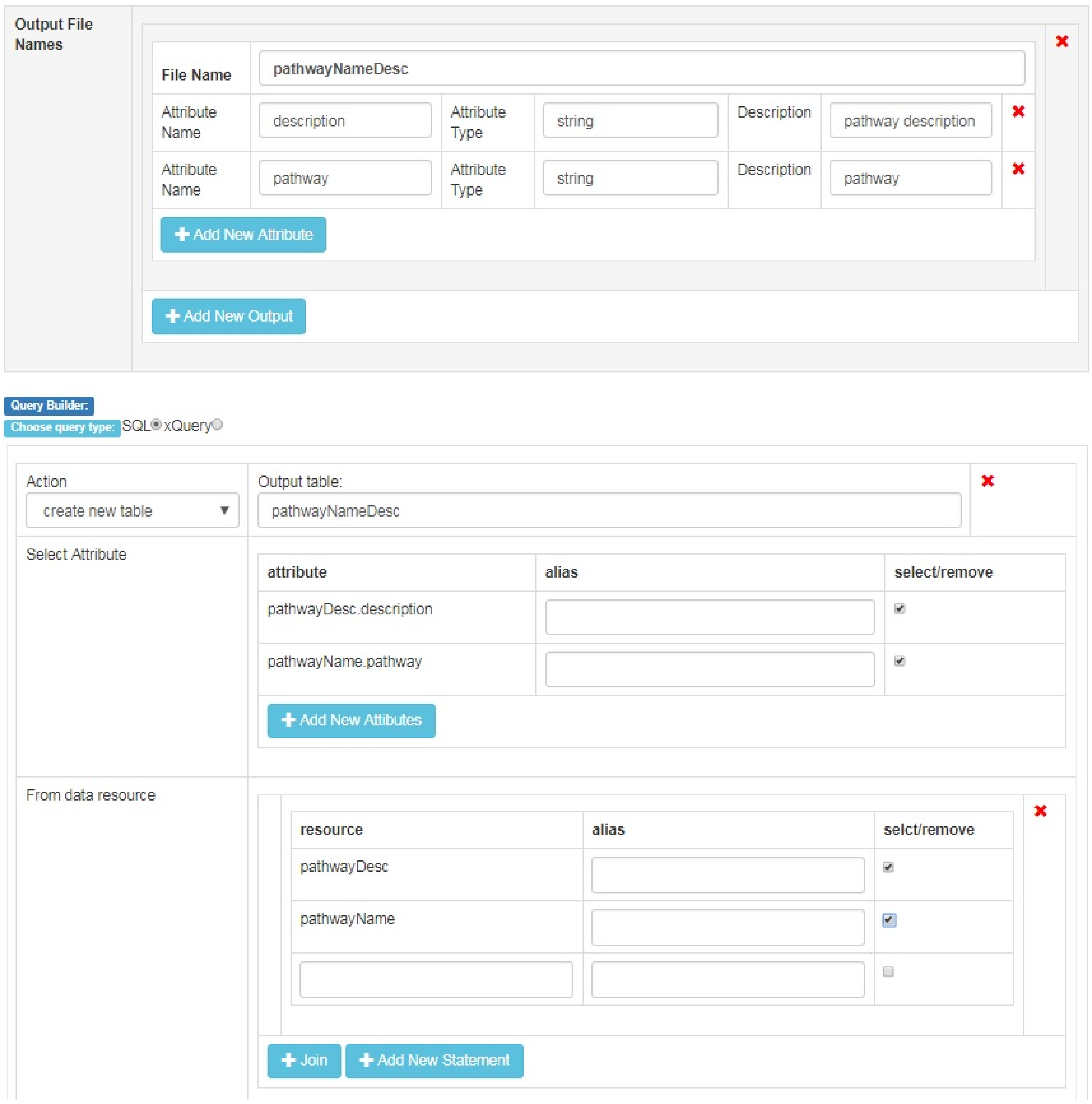
	Resource name	**Resource Type**
›	pathwayNameDesc	SQL

Action:

Library Type	**Method Type**	**Code**	**Resources Out**	**Delete**
Code	sql	see below	pathwayNameDesc	✖

```
drop table if exists pathwayNameDesc;
create table pathwayNameDesc as
select   pathwayDesc.description ,  pathwayName.pathway
from   pathwayDesc ,  pathwayName
;
```

Figure 6.14: Node 5: Merge Two Tables



Output File Names

File Name	pathwayNameDesc				
Attribute Name	description	Attribute Type	string	Description	pathway description
Attribute Name	pathway	Attribute Type	string	Description	pathway

Choose query type: [✓ SQL] xQuery

Action: create new table

Output table: pathwayNameDesc

Select Attribute

attribute	alias	select/remove
pathwayDesc.description		✓
pathwayName.pathway		✓

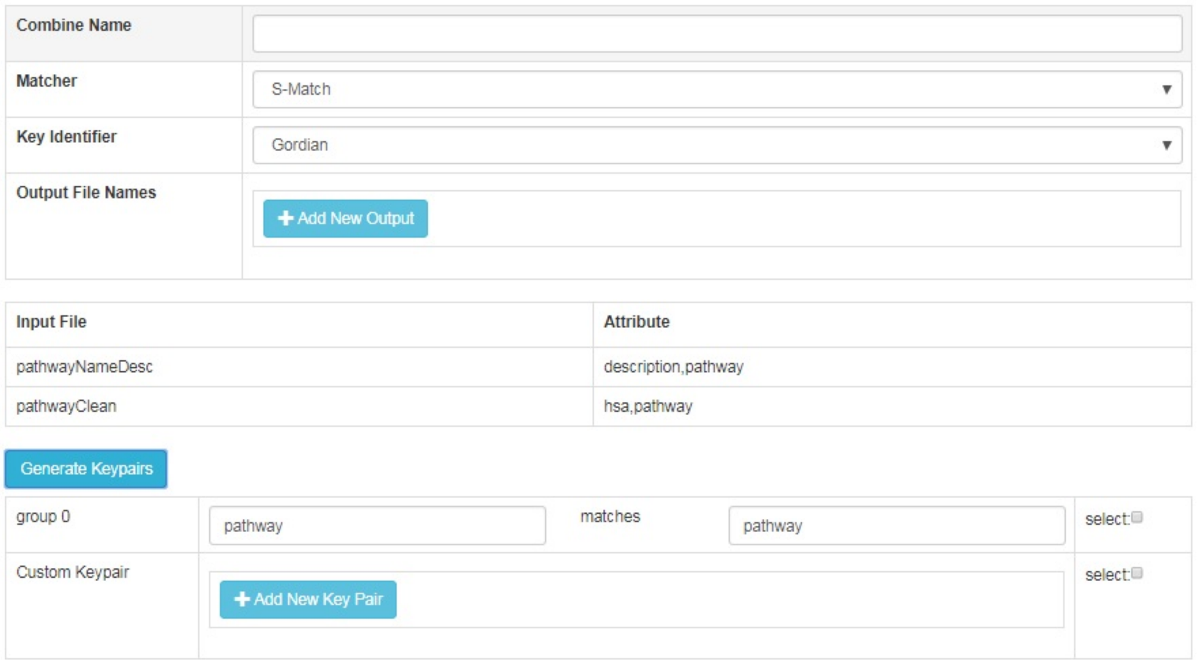
From data resource

resource	alias	selct/remove
pathwayDesc		✓
pathwayName		✓

Figure 6.15: Node 5: Visual Query Builder Configuration

see the matrix of the relationship among other elements.

The whole process has three parts: query build time, query prepare time and query execution time. The query build time in Ma's pipeline is simple, and it is time they spent on building the software. The time VisFlow spent is the construction time of the whole workflow. The first step is registering the two resources: one is the mineral evolution database, and the other is a list of elements that users



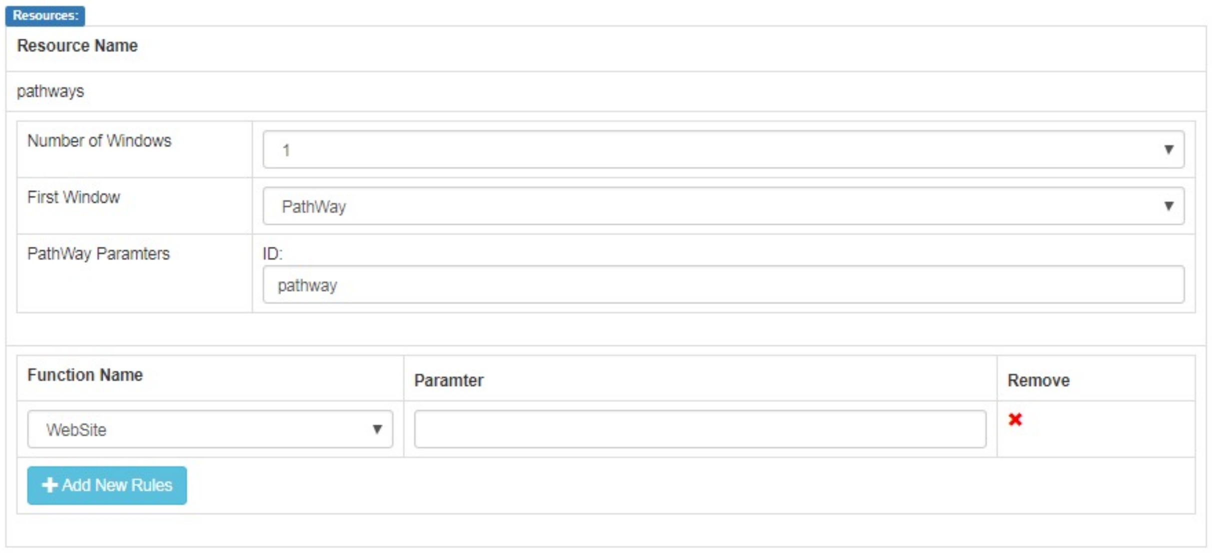
Combine Name	
Matcher	S-Match ▾
Key Identifier	Gordian ▾
Output File Names	+ Add New Output

Input File	Attribute
pathwayNameDesc	description,pathway
pathwayClean	hsa,pathway

Generate Keypairs

group 0	pathway	matches	pathway	select:☐
Custom Keypair	+ Add New Key Pair			select:☐

Figure 6.16: Node 6: Combine Icon



Resources:

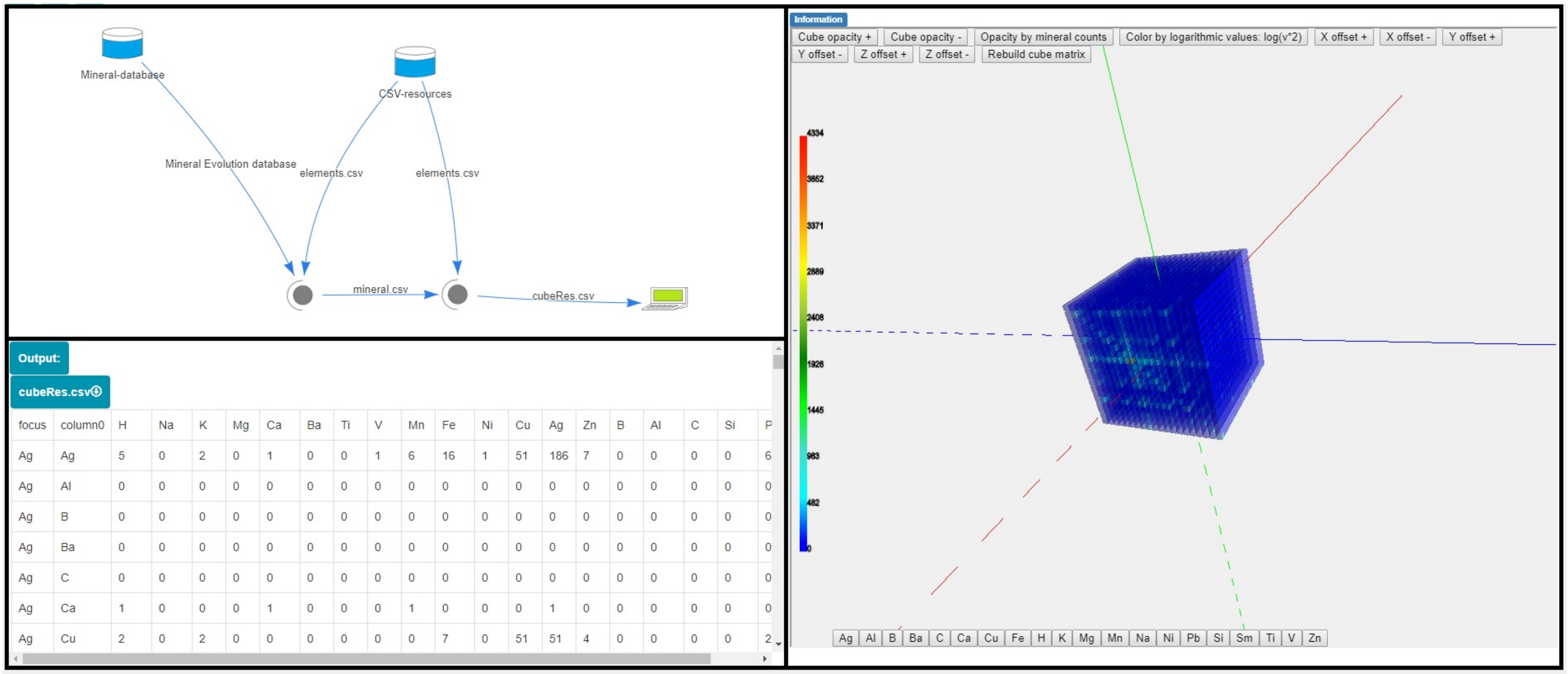
Resource Name
pathways

Number of Windows	1 ▾
First Window	PathWay ▾
PathWay Paramters	ID: pathway

Function Name	Paramter	Remove
WebSite ▾		✖

+ Add New Rules

Figure 6.17: Node 7: Terminal Icon



Output:

cubeRes.csv

focus	column0	H	Na	K	Mg	Ca	Ba	Ti	V	Mn	Fe	Ni	Cu	Ag	Zn	B	Al	C	Si	P
Ag	Ag	5	0	2	0	1	0	0	1	6	16	1	51	186	7	0	0	0	0	6
Ag	Al	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ag	B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ag	Ba	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ag	C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ag	Ca	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Ag	Cu	2	0	2	0	0	0	0	0	0	7	0	51	51	4	0	0	0	0	2

Figure 6.18: Mineral 3D Example

Catalog	Ma's pipeline[1]		VisFlow	
query build time	software development time	120h	register two resources	0.5h
			visual query builder	1h
query prepare time	submitting elements	2h	0h	
	generating CSV files	2h		
query execution time	9sec		314sec	

Table 6.1: Mineral 3D vs. VisFlow

plan to view. The next step of VisFlow is to submit these elements through the mineral evolution database to retrieve the number of counts matrix back. After that, unlike Ma's pipeline they manually clean and aggregate these tables, an *adapter* icon using visual query builder is configured to clean and select matched elements to generate the final results for visualization. The second part is named query prepare time, which only Ma's pipeline has. It contains all the pre-processing of the dataset, which includes submitting elements to the database, and generates the data visualization file. The third part of this evaluation are query execution. In VisFlow, query execution is the time after users click the run button until the results are appropriately rendered on the screen. In Ma's pipeline, it only contains the rendering the results as the 3D cube and plus the execution of the application after selecting the elements.

Table 6.1 shows the time matrix of generating the whole process. An interesting result of this table is that the running time of the software in Ma's pipeline is less than the VisFlow workflow execution time. This is because the whole datasets are generated and stored in his pipeline when selecting the elements from their interface; they are limited in these pre-defined datasets. In VisFlow, the datasets are created on the fly, which is more flexible when users plan to view the elements that Ma's pipeline didn't collect in their dataset. In this case, they need to submit the missing elements through the database and add, modify the dataset before they call their user interface. There are strong possibilities that the schema of the end point has changes over time, which means it returns the same data but in different format. Ma's pipeline needs to go back to the query prepare steps to resubmit elements and generate the CSV file again. In VisFlow, users just need to update the schema in the resource register page which saves a lot of time. Even though the running time VisFlow is slower, the total time spent for the whole process in VisFlow is far less than the time spent on building the pipeline and it handles the possible changes faster than the original pipeline.

[1]All the data entries are collected and validated by Xiaogang Ma

Chapter 7: Case Studies

This chapter provides five comprehensive examples from different domains to show that VisFlow is domain independent and easy to compose and execute workflows without a deep understanding of programming skills.

7.1 Bioinformatics Examples

7.1.1 KEGG Pathway Example

This Example is from the bioinformatics domain. A biological pathway is a series of interactions among molecules in a cell that leads to a certain product or a change in a cell. Such a pathway can trigger the assembly of new molecules, such as a fat or protein. Pathways can also turn genes on and off, or spur a cell to move [1]. Researchers have discovered many biological pathways in the past decades. However, there are still many unknown pathways waiting to be discovered, and many types of research are founded to explore and understand the complex connections among cells, bacteria in mice, and other creatures. Researches on pathways not only help us to understand a disease but also provide information about health. Kyoto Encyclopedia of Genes and Genomes (KEGG[1]) is a database of a collection of pathways that is popular and highly used by biologists. It provides a RESTful based API that allows a user to submit queries to this database and retrieve database information. It utilized the pathways and labeled each pathway with a unique id named KEGG ID. Entrez Gene is the National Center for Biotechnology Information (NCBI)'s database for gene-specific information [3]. It provides unique integers (Gene ID) as identifiers for genes and other loci for a subset of model organisms. The KEGG database offers two ID mapping function to make the Gene IDs compatible with cross-referencing with Pathway IDs. One mapping function is mapping the Gene ID to HSA ID, and the other mapping function is mapping HSA ID to Pathway ID. These two functions aim to help biologists to retrieve corresponding pathways in its database.

To do that, the biologist needs to submit the GeneID to KEGG web service to get the corresponding HSA ID back, and then submit these HSA IDs to KEGG Pathway API to retrieve the mapped pathways. We show its implementation in VisFlow in Figure 7.1, and one of the mapped pathways in Figure 7.2. These steps are included: First extract a list of gene IDs from the genes.csv file, which contains the Gene IDs. Then it adds a prefix string "ncbi-id:" to each Gene ID. After a list of NCBI-id's are generated, this list is submitted to the KEGG web portal to retrieve the Pathway ID and HSA ID, the Pathway

[1] `http://www.kegg.jp/`

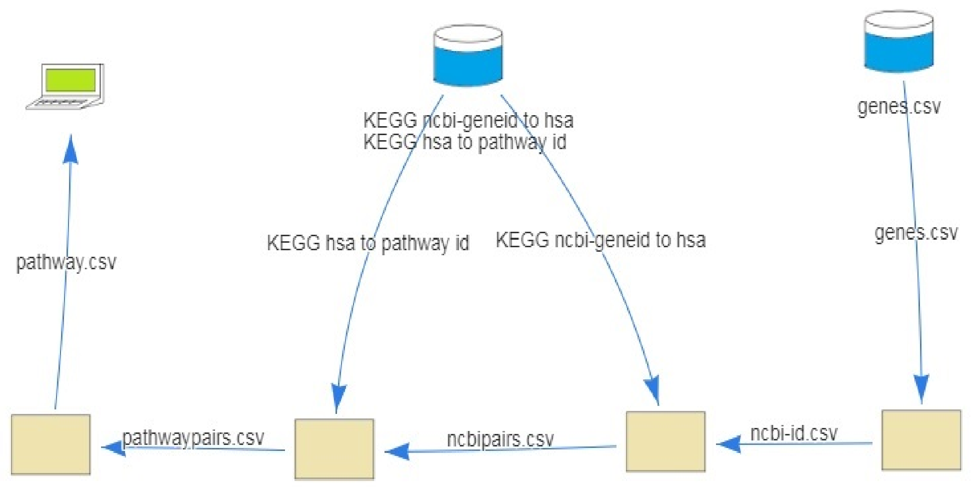


Figure 7.1: KEGG Pathway Workflow

IDs field has been extracted using VisFlow built-in function SplitFun to send to the terminal and shown to users.

7.1.2 Pathway Difference in Orthology Genes Example

This example also uses the KEGG databases, while it contains more complex computation logic in the workflow. This workflow is an investigation into cancer-related human genes that focused on determining the differences between the pathways in which they and their orthologs in mouse and rat participate. The workflow accepts a list of human genes as input, and finds the corresponding mouse and rat orthologs from KEGG database, and retrieves the pathways for all three sets of genes, and finds the pathways that are not shared by mouse or rat orthologs corresponding to a human gene. It produces four CSV files as outputs containing the orthologous genes, pathways in which they participate and the unique human pathways.

To do that, a list of human genes are submitted to the KEGG database to retrieve their HSA IDs and their Orthology IDs. These Orthology IDs are then used to get the mouse and rat gene IDs. After that, these IDs are then mapped to the corresponding pathway IDs for human, mouse, and rat. After we have all three sets of pathway IDs, we differentiate these results to get their common and difference IDs and show them as a list of CSV files. We show its implementation in VisFlow in Figure 7.3, and

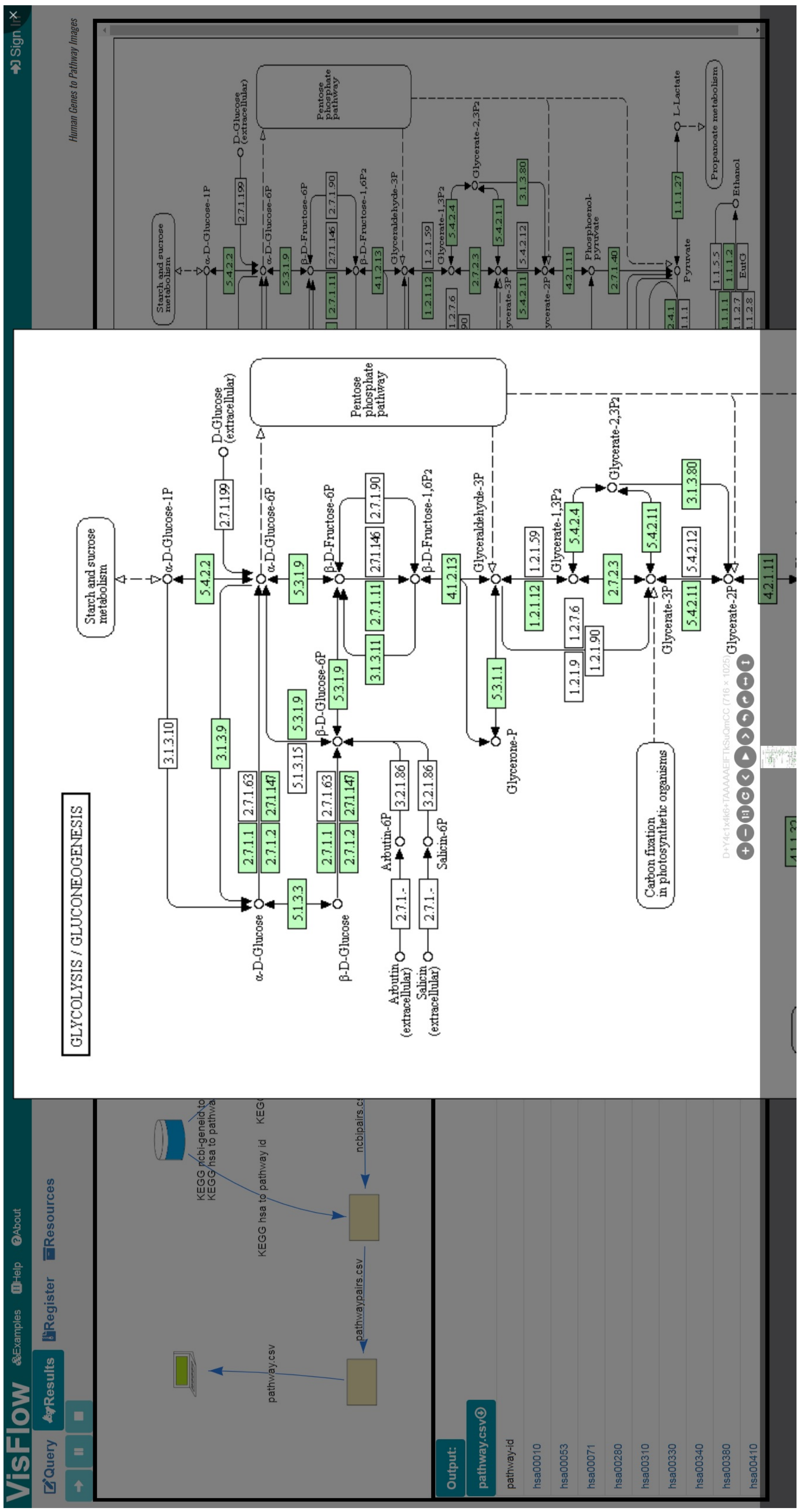


Figure 7.2: KEGG Pathway Workflow Result

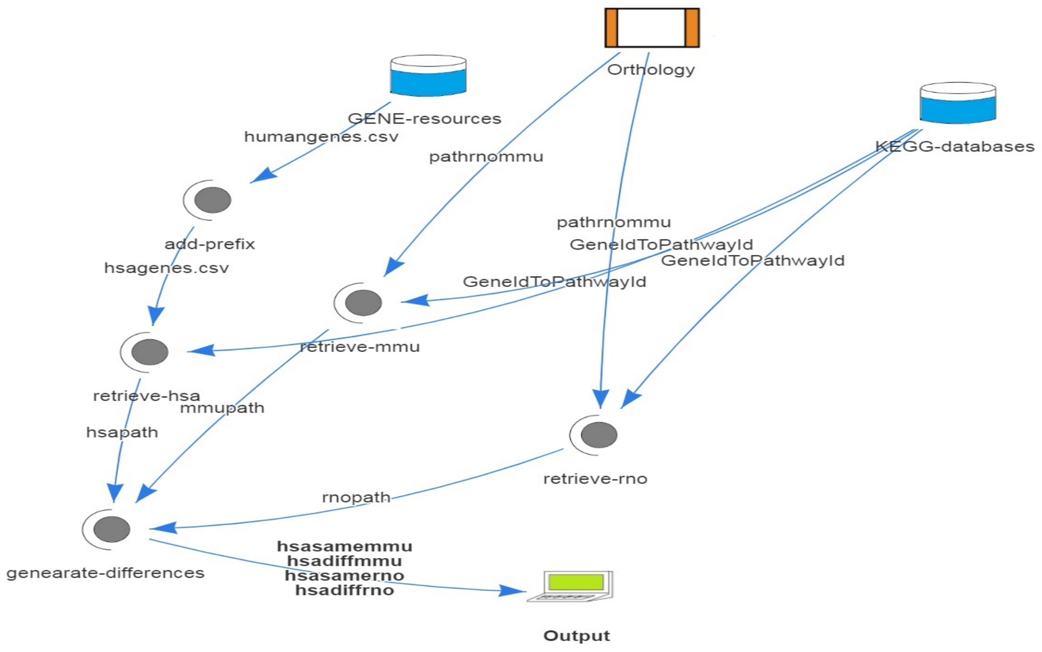
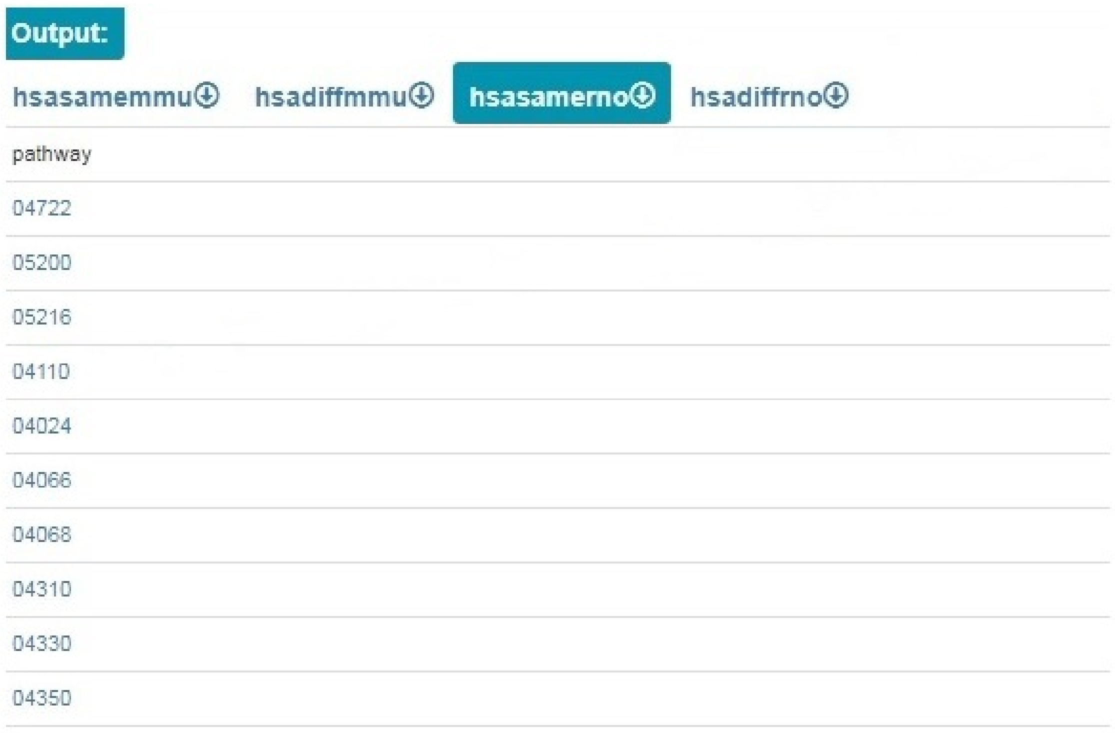


Figure 7.3: Pathway Difference in Orthology Genes

the result in Figure 7.4.

7.1.3 Genome Analytics Example

This example workflow aims to emphasize the capabilities of VisFlow involving different kinds of analytics tools for RNA analysis. Mapping a large sequence of DNA or RNA is critical in bioinformatics, as it helps to identify new gene expression, human re-sequencing, and other purposes. In order to analyze a big trunk of DNA sequence, three executable functions are imported in this workflow. The first one is named Burrows-Wheeler Aligner (BWA) [2] which is a piece of software that maps low-divergent sequences against a large reference genome, such as the human genome. The second one is GenomeAnalysisToolkit (GATK)[4], a collection of command-line tools for analyzing high-throughput sequencing data with a primary focus on variant discovery. Picard is a set of command line tools for manipulating high-throughput sequencing (HTS) data and formats such as SAM/BAM/CRAM and VCF. [5] This workflow is computationally complex, as it uses large data sets and consumes substantial CPU cycles and aligns a large set of sequences against a reference genome by sorting to study manifested phenotypes of individuals for therapeutic diagnosis. The first step of this workflow is to use the BWA tool to align the query sequences with the BWA-MEM algorithm. The next step is applying Picard



pathway
04722
05200
05216
04110
04024
04066
04068
04310
04330
04350

Figure 7.4: Pathway Difference in Orthology Genes

tools to sort the queries and then mark duplicates, and lastly build the BAM file. Then this BAM file is passed to the GATK tool to analyze and generate the matched query's sequences as a result returned to the workflow.

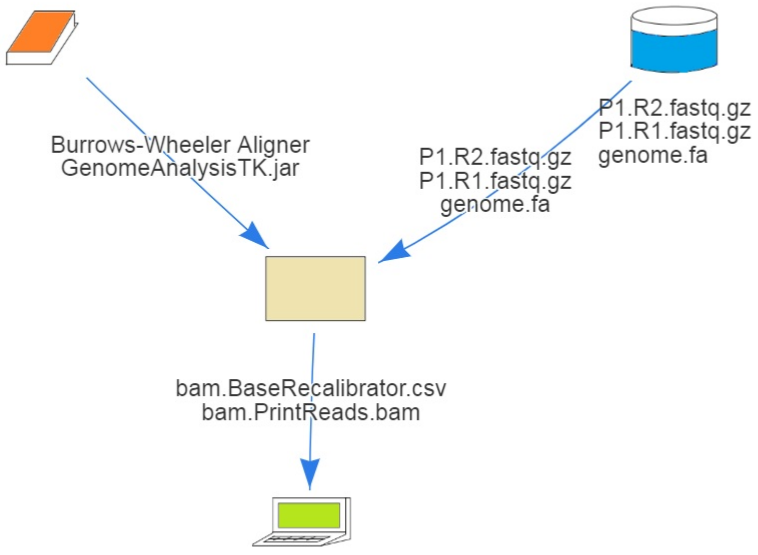
Its implementation in VisFlow is shown in Figure 7.5a. The three required tools are imported using a *library* icon while the input datasets are introduced in the workflow using a *data* icon. All the detailed aligning, sorting and matching steps are abstracted in a single analytics box to execute it. The results are two files passed into a *terminal* icon shown in Figure 7.5b.

7.2 Geoscience Examples

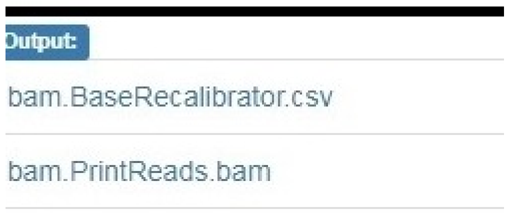
7.2.1 US Fossil Collections Example

In this example, we will illustrate the functionality of VisFlow using a powerful representative example in Geology that aims to study records of fossils collections in a specific geological time period that were discovered in North America in a specific geological timescale.

Fossils are the portions of animals, plants, and other organisms that are older than 10,000 years. The earth's crust consists of layers of rocks called strata that geologists uniquely label with geological



(a) Genome Analytics Workflow


Output:

bam.BaseRecalibrator.csv

bam.PrintReads.bam

(b) Genome Analytics Workflow Result

Figure 7.5: Examples of Genome Analytics

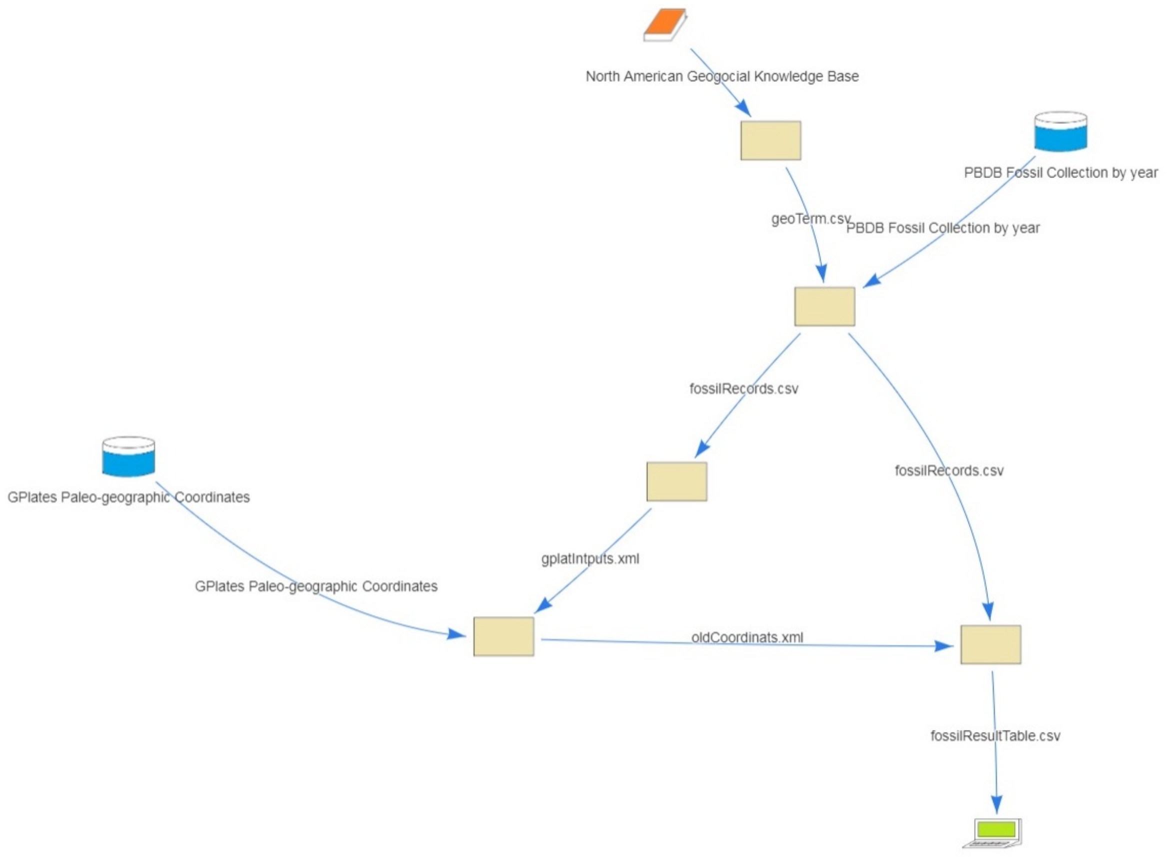
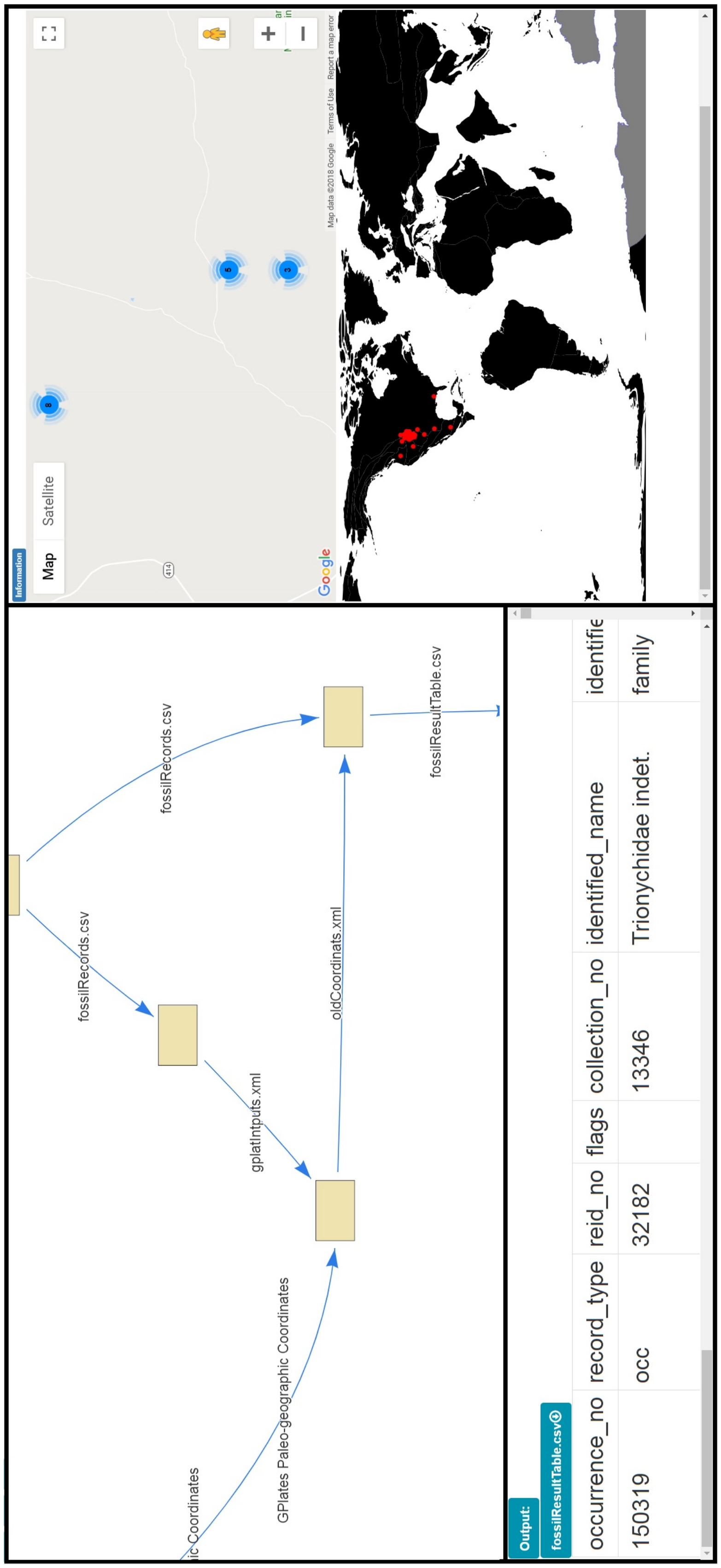

North American Geogocial Knowledge Base
PBDB Fossil Collection by year
geoTerm.csv
PBDB Fossil Collection by year
fossilRecords.csv
fossilRecords.csv
GPlates Paleo-geographic Coordinates
gplatIntputs.xml
GPlates Paleo-geographic Coordinates
oldCoordinats.xml
fossilResultTable.csv

Figure 7.6: US Fossil Collections Workflow

time terms such as Jurassic, Triassic, Permian and so on based on their age for geologic time intervals. The study of fossils across geological time helps geologists to determine the various strata and observe the biological evolution of creatures. Geologists and ecologists try to reconstruct the ecosystem and infer the landscape and terrain of the outcrop by learning the fossils discovered in that area. Fossils not only help geologists and ecologists to learn the food chain in the ecosystem and explain how the geological events affect the environment, the animals, and plants, but also help them to understand the origins, nature, and beauty of the biological diversity of the Earth.

Suppose a researcher wants to explore the fossil records that are 50 million years old and were discovered in North America. Moreover, he wants to link these fossil records back to the paleo-geographical map to identify the ancient biological creatures, discover the evidence of the origin and evolution of life, reconstruct the paleoenvironment, and paleoclimate, and thus provide a reference for the protection of the earth by examining and comparing the diversity, density, granularity and composition of these fossil records. Unfortunately, the information spread over multiple databases has heterogeneous representation formats and uses significantly different access technologies, i.e., North American Geo-



occurrence_no	record_type	reid_no	flags	collection_no	identified_name	identifie
150319	occ	32182		13346	Trionychidae indet.	family

Figure 7.7: US Fossil Collections Workflow Result

logical Knowledgebase, the Paleobiology Database (PaleoBioDB) and the GPlates web service. What's more, geology communities have various standards for describing the geological time scale in different granularities and different web services supporting different standards.

In VisFlow, as shown in Figure 7.6, the geologist can first specify a point of time (e.g., 50 million years ago) and submit to the North American knowledge base of geologic time to convert the point of time to a literal term. This term is then used as input to PaleoBioDB to retrieve fossil records that match the time term. Each fossil record contains the description, identity number, year information, plus a pair of coordinates to show where it was discovered. Submitting these coordinate pairs to the GPlates web server system, we can retrieve the paleo-coordinates. The last step is to call our predefined printer rules to show coordinates in a google map window, and a paleo-geographical map returned from the GPlates web service. The results are shown in Figure 7.7. On the left top window, the figure shows the workflow itself, while on the left bottom window it shows the result of the integrated table of fossil tuples. On the right top side, a built-in Google Map visualization tool is applied to show current coordinates for each fossil. On the right bottom side, another built-in Gplates Map visualization tool shows the paleo coordinates in the ancient map from 50 million years ago.

7.2.2 Jurassic Time Period Fossil information integration Example

This is another example in geoscience. The earth's crust consists of layers of rocks called strata that geologists uniquely label as Jurassic, Triassic, and so on based on their age. Major events in earth's history are often linked or described in reference to these time periods. Continental shifts are one of the events that are often described using these time periods and referenced using an internationally agreed upon reference point, called the Global Boundary Stratotype Section and Point (GSSP), which captures the lower boundary of each such period, together called the paleo coordinates. Geologists and ecologists try to reconstruct the earth at these paleo coordinates by linking and aggregating information on various other artifacts such as fossils, as well as landscape and terrain of an outcrop, from various sources and piecing them together in GSSP specific age determination manner. They look for possible fossils below the rock surface, learn how the geological structures were formed, and examine how the geological events shaped the earth in a specific area.

Suppose a researcher wants to link British Middle Jurassic rocks to Base Middle Jurassic rocks of Fuentelsaz, Spain using the GSSP paleo coordinates to understand the British Middle Jurassic geological stratification by examining and comparing the rock types, color, granularity and the composition of the two. By contrasting these two sets of information, she hopes to get a better understanding of the commonalities and differences of global and regional stratigraphic distribution during this period.

Unfortunately, the data she needs to link is spread over multiple databases that have significantly different access technologies and heterogeneous representation formats, i.e., Purdue University Geologic Time Foundation website, Wikipedia, the Open Geospatial Consortium (OGC) Web Map Service, the GPlates web service, and the Geotime RPI server.

In VisFlow, the workflow she composed is shown in Figure 7.8. She first obtains meta-data from the OGC database about the British geological map, which she then submits to OGC to retrieve the rock age list using HTTP service. She then sends this list to Geotime RPI database to obtain the corresponding GSSP paleo coordinates and age using SPARQL queries for onward transmission to the GPlates web service to retrieve their matching paleo coordinates. Finally, she gets the matching GSSP infographics from Purdue University and adds the Wikipedia page links for their description. In the entire query she composes, she never interrogated the VisFlow system for any of these details. The visual description panel correctly linked all the information on the right by adequately matching the fields using a schema matcher, extracting the columns for the next node in the workflow and combining data from multiple sites using a key discovery algorithm and linking protocol, all transparently. The visualization of information also followed a standard protocol that decided how to display various types of vastly different data types fully automatically as shown in Figure 7.9. For each column, if it is a color code, VisFlow shows the color, if it contains an URL, it wraps a hyperlink for users.

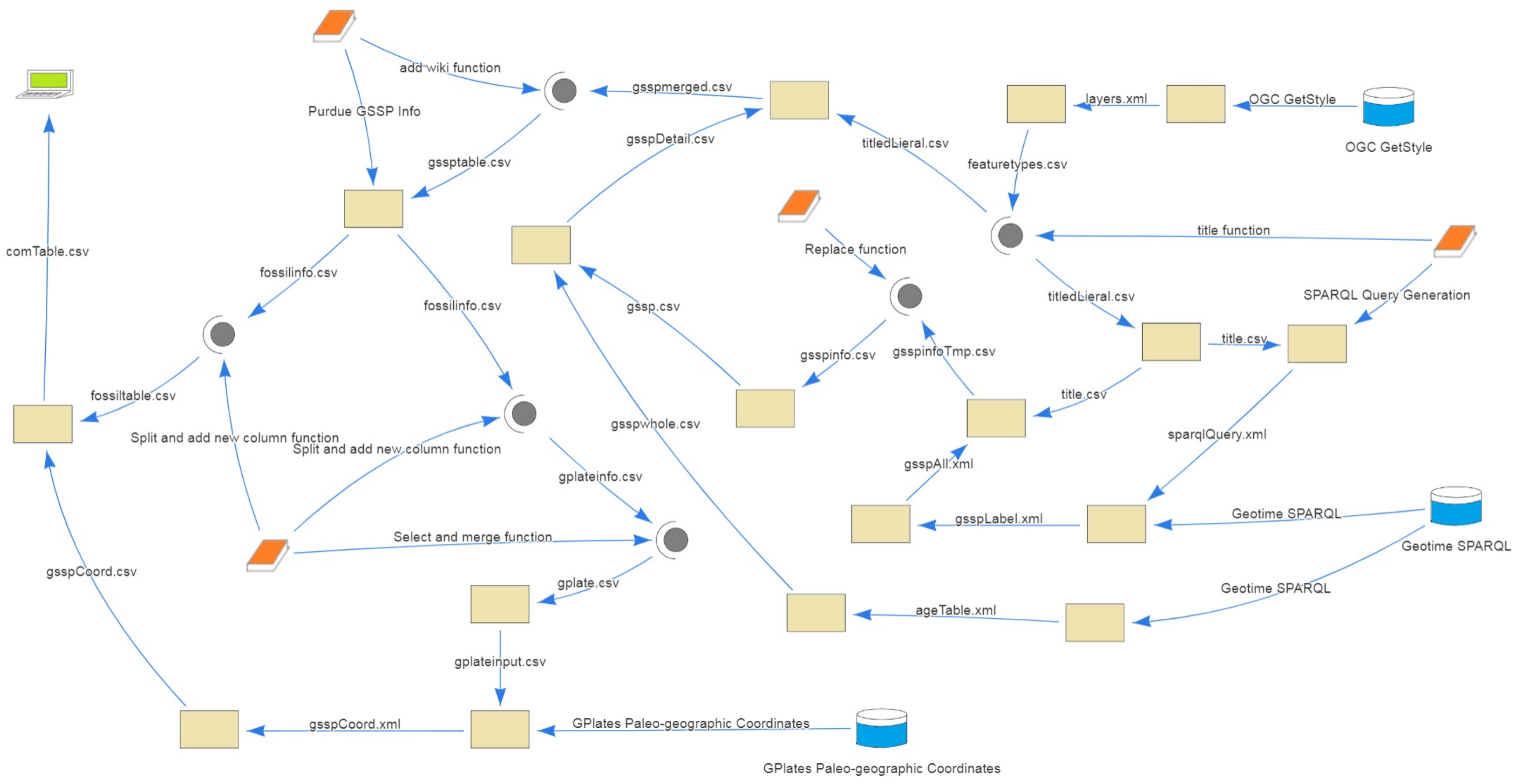
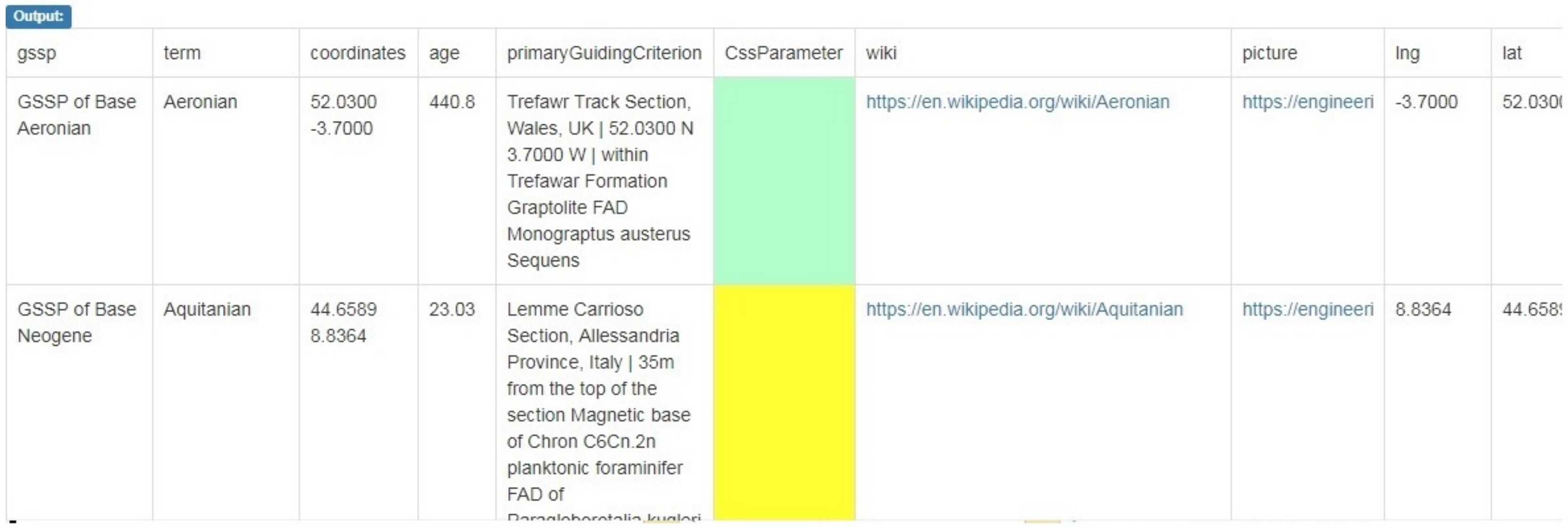


Figure 7.8: Jurassic Time Period Fossil Information Integration Workflow



Output:

gssp	term	coordinates	age	primaryGuidingCriterion	CssParameter	wiki	picture	lng	lat
GSSP of Base Aeronian	Aeronian	52.0300 -3.7000	440.8	Trefawr Track Section, Wales, UK | 52.0300 N 3.7000 W | within Trefawar Formation Graptolite FAD Monograptus austerus Sequens	🟩	https://en.wikipedia.org/wiki/Aeronian	https://engineeri	-3.7000	52.030
GSSP of Base Neogene	Aquitanian	44.6589 8.8364	23.03	Lemme Carrioso Section, Allessandria Province, Italy | 35m from the top of the section Magnetic base of Chron C6Cn.2n planktonic foraminifer FAD of Paragloborotalia kugleri	🟨	https://en.wikipedia.org/wiki/Aquitanian	https://engineeri	8.8364	44.658

Figure 7.9: Jurassic Time Period Fossil Information Integration Workflow Result

Chapter 8: Conclusion and Future work

8.1 Conclusion

In order to help scientists concentrate more on their science questions but not on how to compute it, several successful workflow orchestration languages and systems have been proposed. Despite their popularity, significant limitations reduce their usability and limit applicability in novel applications. In this dissertation, we have introduced a flexible yet powerful web-based data integration and workflow development system that leverages a declarative visual language for ad hoc application design using arbitrary resources. The degree of abstractions supported in a language helps lessen the depth of such familiarity needed, and aids in improving access to and usability of these resources. We advance the idea that once resources are minimally described and abstracted, arbitrary workflows can be designed solely using query primitives supported in VisFlow. Its capabilities can be augmented by including computational artifacts in the form of library functions written in R, Python, and Java, making it a genuinely extensible system.

Concerning supporting the abstract and declarative design, we have defined this visual language with fourteen predefined icons, and the algorithms transform this language to BioFlow to take advantage of the data management of hierarchical data resources. We have also shown an automatic execution which minimizes the execution time by performing parallel execution whenever needed without user configuration to speed up the whole workflow design process.

We have discussed how auto data integration, ad hoc debugging and visual query builder help users in different ways when composing the workflow. A detailed workflow composing comparison example between VisFlow and Taverna has been provided to show how VisFlow reduces the time of configuration. An in-depth pipeline time comparison example has also been discussed to show how effective VisFlow is.

We have present VisFlow's salient features and illustrate its capabilities using a substantial set of examples to prove VisFlow is domain independent workflow system works for distributed data resources that supports HTTP, FTP protocols.

With these features together, VisFlow is a highly abstract declarative workflow system that eliminates programming coding and executes workflows in parallel whenever needed. The prototype VisFlow system can be accessed online at `http://dblab2.nkn.uidaho.edu/VisFlow/`. The source code is also available under the MIT open source license: `https://github.com/dblabuofi/VisFlow` for the user interface and `https://github.com/dblabuofi/VisFlowWS` for the web service.

8.2 Future Work

Although VisFlow is a powerful workflow management system, there are still some improvements it can incorporate to make it better. The future release of VisFlow is expected to include more functionalities to improve usability.

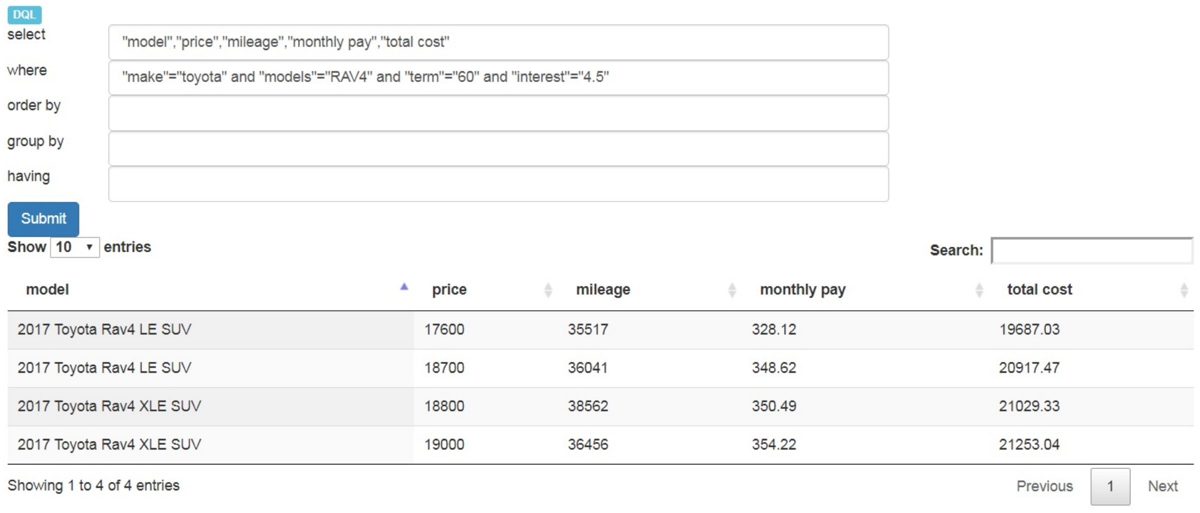
8.2.1 Printing Rules

The current release of VisFlow does not have a sophisticated information visualization capability, in addition to not supporting customization. In our future release, we plan to include libraries for application specific visualization of complex objects, and include device specific display policies so that depending on the data types needing rendering, the system is able to select default display actions, and allow user selectable functions to facilitate customization.

8.2.2 Fully Automated Composing Workflow

A fully automated workflow system has still not been created yet. Besides VisFlow, some systems aim to help users to construct the workflow. For example, Wings [61] uses the ontology to help users select possible datasets and possible parameters. They also provide a collection of workflow templates with an enhanced AI algorithm to generate the concert workflow using these templates. In VisFlow, we also store the metadata information to automate processes. In order to build a fully automated workflow management system, we are planing to offer a text window which allows users to write a description of the workflow they are planning to build, then we apply Natural Language Processing (NLP) algorithms [146, 155] with domain knowledge bases like Wikidata[1] or DBpedia[2] to extract the concepts and their relationships. If we can provide an intermediate query language which allows searching different domains by providing attributes with their values based on the concepts and relationships we get in previous step, we can transfer it into VisFlow for execution. The basic idea has been proposed in the Deep Query Language (DQL) [78]. A simple example that is illustrated in Figure 8.1 is in the second-hand car market domain. Users submitted a query to show the model, price, mileage, monthly pay and total cost fields of Toyota RAV4 in 60 months with interest rate at 4.5% within 100 miles of their location. The whole process behind this query is, first it submits the make and model fields to the Hertz website as shown in Figure 8.2 to get the price, mileage, and price back; then it submits the price to auto loan calculator (shown in Figure 8.3) to extract the total cost in 60 months, and lastly, it divided this value to get the monthly payment. The result is shown as the lower part of the figure.

[1]http://www.wikidata.org
[2]http://wiki.dbpedia.org

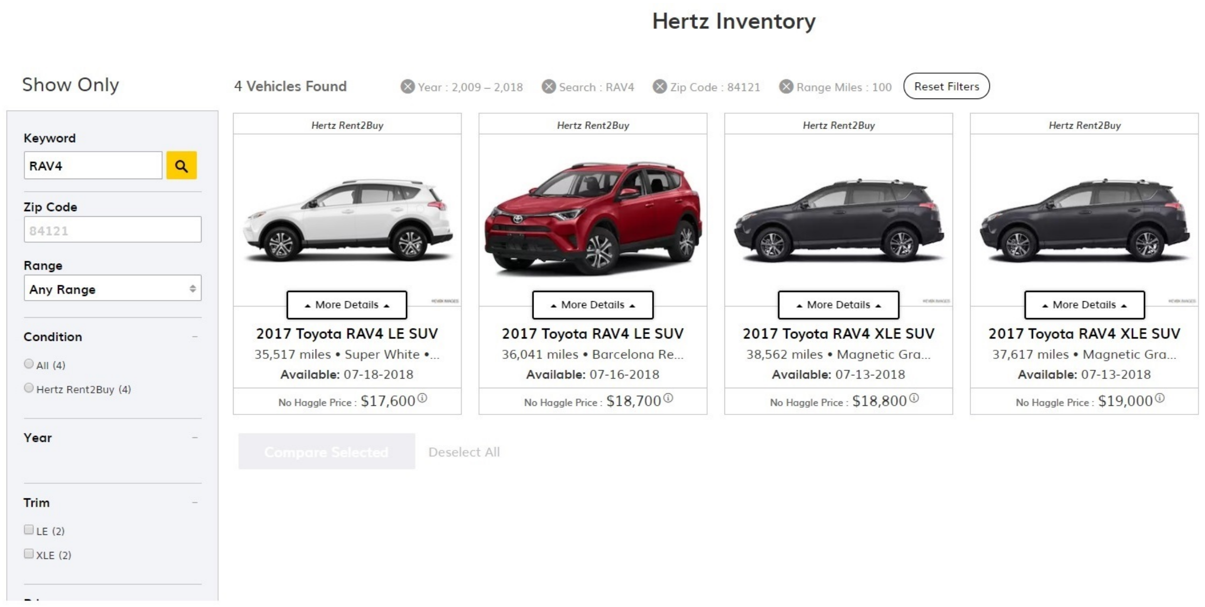

DQL

select	"model","price","mileage","monthly pay","total cost"
where	"make"="toyota" and "models"="RAV4" and "term"="60" and "interest"="4.5"
order by	
group by	
having	

model	price	mileage	monthly pay	total cost
2017 Toyota Rav4 LE SUV	17600	35517	328.12	19687.03
2017 Toyota Rav4 LE SUV	18700	36041	348.62	20917.47
2017 Toyota Rav4 XLE SUV	18800	38562	350.49	21029.33
2017 Toyota Rav4 XLE SUV	19000	36456	354.22	21253.04

Showing 1 to 4 of 4 entries

Figure 8.1: DQL Query Example



2017 Toyota RAV4 LE SUV
35,517 miles • Super White •...
Available: 07-18-2018
No Haggle Price : $17,600

2017 Toyota RAV4 LE SUV
36,041 miles • Barcelona Re...
Available: 07-16-2018
No Haggle Price : $18,700

2017 Toyota RAV4 XLE SUV
38,562 miles • Magnetic Gra...
Available: 07-13-2018
No Haggle Price : $18,800

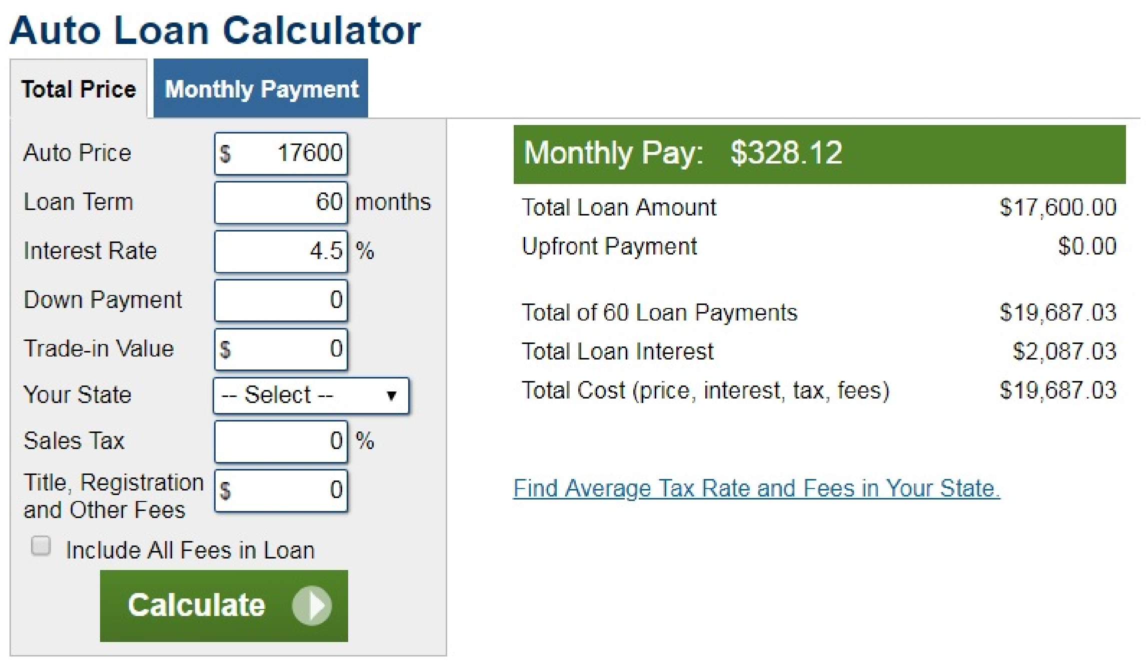
2017 Toyota RAV4 XLE SUV
37,617 miles • Magnetic Gra...
Available: 07-13-2018
No Haggle Price : $19,000

Figure 8.2: Hertz Used Car Selling[1]

To generate and execute a DQL query like this, several standard parts are needed. First, we need a repository of web services with recommendation systems to help us clarify the domain of the query and select the possible websites to submit queries. When submitting queries, auto form filling systems are needed to match and submit corresponding data to the websites. After that, comprehensive wrappers are provided to extract the repeated information back and match the information users need. The last step is combining and collecting this calculation together to show as results. To describe these different

[1]`https://www.hertzcarsales.com/`

steps is easy. However, when considering building a fully automatic system like this, it is still very challenging.



Auto Price	$ 17600	
Loan Term	60	months
Interest Rate	4.5	%
Down Payment	0	
Trade-in Value	$ 0	
Your State	-- Select --	
Sales Tax	0	%
Title, Registration and Other Fees	$ 0	

☐ Include All Fees in Loan

Monthly Pay: $328.12

Total Loan Amount	$17,600.00
Upfront Payment	$0.00
Total of 60 Loan Payments	$19,687.03
Total Loan Interest	$2,087.03
Total Cost (price, interest, tax, fees)	$19,687.03

Figure 8.3: Auto Loan Calculator[2]

First, it requires a repository to store the different websites' metadata information similar to VisFlow's resource and function management. This information is highly likely to change very often and auto updates of this information will be a strong requirement. This requires the computer to understand the HTML code fully. Based on this repository, a recommendation system is needed to pick up the specific websites to submit queries. The recommendation system needs to understand the knowledge of different domains with their context meanings in the queries to make the right decision. After that, auto form filling systems compose the queries and submit to the website. The current limitation of auto form filling systems is that they can only deal with the static website. However, most of the websites with form submission we access are dynamic. This research topic has not yet been studied yet. This also holds for web wrappers. Although many wrappers have been proposed in the last few decades, they still require human interaction to clean and extract the repeated data. To glue different parts of the data together requires current mapping queries into different small steps and executing them based on their logic orders and then assembling the data as needed. VisFlow is a good candidate as the lower level of DQL when submitting queries and processing data. However, it lacks the logic control system to map

[2]https://www.calculator.net/auto-loan-calculator.html

DQL queries to VisFlow queries. To auto-generate VisFlow queries, one of the practical ways is that the application has a vast repository of workflow templates, and we store a paragraph of descriptions about what this workflow does, arguments it takes and metadata information about the output of the workflow. Then we can apply subgraph matching algorithms when we design our workflow to pick up the possible candidates and assign input datasets to it. Based on the knowledge we have, we connect different templates to generate the whole workflow and execute it.

References

[1] Biological pathways fact sheet. https://www.genome.gov/27530687/. Accessed: 2018-02-05.

[2] Burrows-wheeler aligner. http://bio-bwa.sourceforge.net/. Accessed: 2018-02-05.

[3] Entrez gene: gene-centered information at ncbi. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3013746/. Accessed: 2018-02-05.

[4] Gatk quick start guide. https://software.broadinstitute.org/gatk/documentation/quickstart.php. Accessed: 2018-02-05.

[5] Picard. https://broadinstitute.github.io/picard/. Accessed: 2018-02-05.

[6] *XAML - Extensible Application Markup Language.* http://msdn.microsoft.com/en-us/library/ms752059.aspx.

[7] Mohamed Abouelhoda, Shadi Issa, and Moustafa Ghanem. Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support. *BMC Bioinformatics*, 13(1):77, 2012.

[8] Majed Al-Mashari, Zahir Irani, and Mohamed Zairi. Business process reengineering: a survey of international experience. *Business Process Management Journal*, 7(5):437–455, 2001.

[9] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on Scientific and Statistical Database Management*, pages 423–424. IEEE, 2004.

[10] Oszkár Ambrus, Knud Möller, Siegfried Handschuh, et al. Konduit vqb: a visual query builder for sparql on the social semantic desktop. In *Workshop on Visual Interfaces to the Social and Semantic Web*. CEUR-WS, 2010.

[11] Mohammad Shafkat Amin and Hasan Jamil. An efficient web-based wrapper and annotator for tabular data. *International Journal of Software Engineering and Knowledge Engineering*, 20(02):215–231, 2010.

[12] Samira Babalou, Alsayed Algergawy, and Birgitta König-Ries. An ontology-based scientific data integration workflow. In *29th GI-Workshop Grundlagen von Datenbanken*, pages 30–35. ACM, 2017.

[13] Tyler WH Backman and Thomas Girke. systempiper: Ngs workflow and report generation environment. *BMC bioinformatics*, 17(1):388, 2016.

[14] Bartosz Balis. Increasing scientific workflow programming productivity with hyperflow. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, pages 59–69. IEEE Press, 2014.

[15] Detlev Bannasch, Alexander Mehrle, Karl-Heinz Glatting, Rainer Pepperkok, Annemarie Poustka, and Stefan Wiemann. Lifedb: a database for functional genomics experiments integrating information from external sources, and serving as a sample tracking system. *Nucleic Acids Research*, 32(suppl_1):D505–D508, 2004.

[16] Louis Bavoil, Steven P Callahan, Patricia J Crossno, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Visualization, 2005.*, pages 135–142. IEEE, 2005.

[17] Stephan Beisken, Mark Earll, David Portwood, Mark Seymour, and Christoph Steinbeck. Masscascade: Visual programming for lc-ms data processing in metabolomics. *Molecular informatics*, 33(4):307–310, 2014.

[18] Andreas Bender, Angela Poschlad, Stefan Bozic, and Ivan Kondov. A service-oriented framework for integration of domain-specific data models in scientific workflows. *Procedia Computer Science*, 18:1087–1096, 2013.

[19] Michael R Berthold, Nicolas Cebron, Fabian Dill, Thomas R Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime-the konstanz information miner: version 2.0 and beyond. *ACM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.

[20] Sourav S Bhowmick, Byron Choi, and Curtis Dyreson. Data-driven visual graph query interface construction and maintenance: challenges and opportunities. *Proceedings of the VLDB Endowment*, 9(12):984–992, 2016.

[21] Sourav S Bhowmick, Huey Eng Chua, Benji Thian, and Byron Choi. Visual: An hci-inspired simulator for blending visual subgraph query construction and processing. In *2015 IEEE 31st International Conference on Data Engineering (ICDE)*, pages 1480–1483. IEEE, 2015.

[22] Clemente Rafael Borges and José Antonio Macías. Feasible database querying using a visual end-user approach. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 187–192. ACM, 2010.

[23] Marat Boshernitsan and Michael Sean Downes. Visual programming languages: A survey. Technical report, U.C. Berkeley, 2004.

[24] Daina Bouquin, Katie Frey, Maria McEachern, James Damon, Daniel Guarracino, Alex McGrath, Edwin Henneken, and Lindsay Smith Zrull. Project phaedra: Preserving harvard's early data and research in astronomy. In *EPJ Web of Conferences*, volume 186, page 07003. EDP Sciences, 2018.

[25] Richard Boyle, Bahram Parvin, Darko Koracin, Nikos Paragios, and Syeda-Mahmood Tanveer. *Advances in Visual Computing.* Springer, 2007.

[26] Jean Bresson. Spatial structures programming for music. In *Spatial Computing Workshop (SCW)*, page 1. Autonomous Agents and MultiAgent Systems (AAMAS), 2012.

[27] Jean Bresson, Carlos Agon, and Gérard Assayag. Openmusic: visual programming environment for music composition, analysis and research. In *Proceedings of the 19th ACM international conference on Multimedia*, pages 743–746. ACM, 2011.

[28] Jean Bresson, John MacCallum, and Adrian Freed. o. om: structured-functional communication between computer music systems using osc and odot. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, pages 41–47. ACM, 2016.

[29] Elizabeth Brunk, Kevin W George, Jorge Alonso-Gutierrez, Mitchell Thompson, Edward Baidoo, George Wang, Christopher J Petzold, Douglas McCloskey, Jonathan Monk, Laurence Yang, et al. Characterizing strain variation in engineered e. coli using a multi-omics-based workflow. *Cell systems*, 2(5):335–346, 2016.

[30] Jim Burton and John Howse. The semiotics of spider diagrams. *Logica Universalis*, 11(2):177–204, 2017.

[31] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Claudio T Silva, and Huy T Vo. Managing the evolution of dataflows with vistrails. In *22nd International Conference on Data Engineering Workshops*, pages 71–71. IEEE, 2006.

[32] Tiziana Catarci, Maria F Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[33] Joshua Cates, Lisa Nevell, Suresh I Prajapati, Laura D Nelon, Jerry Y Chang, Matthew E Randolph, Bernard Wood, Charles Keller, and Ross T Whitaker. Shape analysis of the basioccipital bone in pax7-deficient mice. *Scientific reports*, 7(1):17955, 2017.

[34] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[35] Ryan H Choi and Raymond K Wong. Vxq: A visual query language for xml data. *Information Systems Frontiers*, 17(4):961–981, 2015.

[36] Wayne Citrin, Richard Hall, and Benjamin Zorn. Programming with visual expressions. In *11th IEEE International Symposium on Visual Languages*, pages 294–301. IEEE, 1995.

[37] Leandro Corrêa, Ronnie Alves, Fabiana Goés, Cristian Chaparro, and Lucinéia Thom. A pipeline for functional and visual analytics of microbial genetic networks. *Dynamic Networks and Knowledge Discovery*, page 13, 2014.

[38] Valter Crescenzi, Paolo Merialdo, and Disheng Qiu. Crowdsourcing large scale wrapper inference. *Distributed and Parallel Databases*, 33(1):95–122, 2015.

[39] Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4(4):219–230, 2011.

[40] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.

[41] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[42] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, et al. Orange: data mining toolbox in python. *The Journal of Machine Learning Research*, 14(1):2349–2353, 2013.

[43] Luc Di Gallo, Cedric Reux, Frédéric Imbeaux, J-F Artaud, Michal Owsiak, Bernard Saoutic, Giacomo Aiello, P Bernardi, Guido Ciraolo, Jérôme Bucalossi, et al. Coupling between a multiphysics workflow engine and an optimization framework. *Computer Physics Communications*, 200:76–86, 2016.

[44] Ivo D. Dinov, Federica Torri, Fabio Macciardi, Petros Petrosyan, Zhizhong Liu, Alen Zamanyan, Paul Eggert, Jonathan Pierce, Alex Genco, James A. Knowles, Andrew P. Clark, John D. Van Horn, Joseph Ames, Carl Kesselman, and Arthur W. Toga. Applications of the pipeline environment for visual informatics and genomics computations. *BMC Bioinformatics*, 12(1):1–20, 2011.

[45] Long H Do, Francisco F Esteves, Harvey J Karten, and Ethan Bier. Booly: a new data integration platform. *BMC bioinformatics*, 11(1):513, 2010.

[46] Xin Luna Dong and Felix Naumann. Data fusion: resolving data conflicts for integration. *Proceedings of the VLDB Endowment*, 2(2):1654–1655, 2009.

[47] Dolev Dotan and Ron Y. Pinter. HyperFlow: An integrated visual query and dataflow language for end-user information analysis. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 27–34, Washington, DC, USA, 2005. IEEE.

[48] Gerard Tromp Eric B. Lipsky, Brian R. King. Node-oriented workflow (now): A command template workflow management tool for high throughput data analysis pipelines. *Journal of Data Mining in Genomics & Proteomics*, 5(2):–, 2014.

[49] Martin Erwig, Karl Smeltzer, and Xiangyu Wang. What is a visual language? *Journal of Visual Languages & Computing*, 38:9–17, 2017.

[50] Martin Erwig and Eric Walkingshaw. Visual explanations of probabilistic reasoning. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 23–27. IEEE, 2009.

[51] Thomas Fahringer, Sabri Pllana, and Alex Villazon. A-gwl: Abstract grid workflow language. In *International Conference on Computational Science*, pages 42–49. Springer, 2004.

[52] Thomas Fahringer, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, et al. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for e-Science*, pages 450–471. Springer, 2007.

[53] Peter M Fischer, Dana Florescu, Martin Kaufmann, and Donald Kossmann. Translating sparql and sql to xquery. *XML Prague*, pages 81–98, 2011.

[54] Marco Fondi and Pietro Liò. Multi-omics and metabolic modelling pipelines: challenges and tools for systems microbiology. *Microbiological research*, 171:52–64, 2015.

[55] International Organization for Standardization and International Electrotechnical Commission. *Software Engineering–Product Quality: Quality model*, volume 1. ISO/IEC, 2001.

[56] Alexander Forst, Eva Kühn, and Omran Bukhres. General purpose work flow languages. *Distributed and Parallel Databases*, 3(2):187–218, 1995.

[57] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[58] Avigdor Gal, Giovanni Modica, and Hasan M Jamil. OntoBuilder: Fully automatic extraction and consolidation of ontologies from web sources. In *International Conference on Data Engineering*, page 853. IEEE, 2004.

[59] Sandesh Ghimire, Jwala Dhamala, Jaume Coll-Font, Jess D Tate, Maria S Guillem, Dana H Brooks, Rob S MacLeod, and Linwei Wang. Overcoming barriers to quantification and comparison of electrocardiographic imaging methods: A community-based approach. *Computing*, 44:1, 2017.

[60] Yolanda Gil. Workflow composition: Semantic representations for flexible automation. In *Workflows for e-Science*, pages 244–257. Springer, 2007.

[61] Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1767. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[62] Syed Zeeshan Haider Gillani, Muhammad Intizar Ali, and Alessandra Mileo. Xsparql-viz: A mashup-based visual query editor for xsparql. In *Extended Semantic Web Conference*, pages 219–224. Springer, 2013.

[63] Fausto Giunchiglia, Aliaksandr Autayeu, and Juan Pane. S-match: an open source framework for matching lightweight ontologies. *Semantic Web*, 3(3):307–317, 2012.

[64] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. S-match: an algorithm and an implementation of semantic matching. In *European semantic web symposium*, pages 61–75. Springer, 2004.

[65] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[66] Florian Haag, Steffen Lohmann, Stephan Siek, and Thomas Ertl. Queryvowl: Visual composition of sparql queries. In *International Semantic Web Conference*, pages 62–66. Springer, 2015.

[67] Florian Haag, Steffen Lohmann, Stephan Siek, and Thomas Ertl. Visual querying of linked data with queryvowl. *SumPre-HSWI@ ESWC*, 4:2–2, 2015.

[68] Ákos Hajnal, Zoltán Farkas, and Péter Kacsuk. Data avenue: remote storage resource management in ws-pgrade/guse. In *6th International Workshop on Science Gateways*, pages 1–5. IEEE, 2014.

[69] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[70] Shazzad Hosain and Hasan Jamil. An algebraic language for semantic data integration on the hidden web. In *IEEE International Conference on Semantic Computing*, pages 237–244. IEEE, 2009.

[71] Shahriyar Hossain and Hasan Jamil. A visual interface for on-the-fly biological database integration and workflow design using vizbuilder. In *International Workshop on Data Integration in the Life Sciences*, pages 157–172. Springer, 2009.

[72] John Howse, Fernando Molina, John Taylor, and Stuart Kent. Reasoning with spider diagrams. In *1999 IEEE Symposium on Visual Languages*, pages 138–145. IEEE, 1999.

[73] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res*, 34, July 2006. Web Server issue.

[74] Hasan Jamil and Bilal El-Hajj-Diab. Bioflow: A web-based declarative workflow language for life sciences. In *2008 IEEE Congress on Services-Part I*, pages 453–460. IEEE, 2008.

[75] Hasan Jamil, Aminul Islam, and Shahriyar Hossain. A declarative language and toolkit for scientific workflow implementation and execution. *International Journal of Business Process Integration and Management*, 5(1):3–17, 2010.

[76] Hasan M Jamil. Improving integration effectiveness of id mapping based biological record linkage. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 12(2):473–486, 2015.

[77] Hasan M Jamil. A visual interface for querying heterogeneous phylogenetic databases. *IEEE/ACM transactions on computational biology and bioinformatics*, 14(1):131–144, 2017.

[78] Hasan M Jamil and Hosagrahar V Jagadish. A structured query model for the deep relational web. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1679–1682. ACM, 2015.

[79] Nandish Jayaram, Rohit Bhoopalam, Chengkai Li, and Vassilis Athitsos. Orion: Enabling suggestions in a visual query builder for ultra-heterogeneous graphs. *arXiv preprint arXiv:1605.06856*, 2016.

[80] Sunil Jigyasu, Sujeet Banerjee, Vinayak Borkar, Michael Carey, Kanad Dixit, Anil Malkani, and Sachin Thatte. Sql to xquery translation in the aqualogic data services platform. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 97–97. IEEE, 2006.

[81] Peter Kacsuk, Zoltan Farkas, Miklos Kozlovszky, Gabor Hermann, Akos Balasko, Krisztian Karoczkai, and Istvan Marton. Ws-pgrade/guse generic dci gateway framework for a large variety of user communities. *Journal of Grid Computing*, 10(4):601–630, 2012.

[82] Roozbeh Kalateh, Lynne Ogg, Matanat Charkazova, and Dimitrios I. Gerogiorgis. A database and workflow integration methodology for rapid evaluation and selection of improved oil recovery (IOR) technologies for heavy oil fields. *Advances in Engineering Software*, 100:176–197, 2016.

[83] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *ACM SIGPLAN Notices*, volume 32, pages 327–341. ACM, 1997.

[84] Jihie Kim, Marc Spraragen, and Yolanda Gil. An intelligent assistant for interactive workflow composition. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 125–131. ACM, 2004.

[85] Jacqueline C Kirby, Peter Speltz, Luke V Rasmussen, Melissa Basford, Omri Gottesman, Peggy L Peissig, Jennifer A Pacheco, Gerard Tromp, Jyotishman Pathak, David S Carrell, et al. Phekb: a catalog and workflow for creating electronic phenotype algorithms for transportability. *Journal of the American Medical Informatics Association*, 23(6):1046–1052, 2016.

[86] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90. IOS Press, 2016.

[87] Yung-Chih Lai, Randall B Widelitz, and Cheng-Ming Chuong. Systems biology analyses in chicken: Workflow for transcriptome and chip-seq analyses using the chicken skin paradigm. In *Avian and Reptilian Developmental Biology*, pages 87–100. Springer, 2017.

[88] Anna-Lena Lamprecht and Tiziana Margaria. Automatic synthesis of bioconductor pipelines: A domain modeling challenge. In *Proceedings of the 8th Semantic Web Applications and Tools for Life Sciences International Conference, Cambridge UK, December 7-10*, pages 216–217. CEUR-WS.org, 2015.

[89] W Lei, Y Ruan, E Bozdag, JA Smith, RT Modrak, L Krischer, Y Chen, MP Lefebvre, and J Tromp. Automation of global adjoint tomography based on asdf and workflow management tools. In *AGU Fall Meeting Abstracts*. American Geophysical Union, 2016.

[90] Chenliang Li, Jianshu Weng, Qi He, Yuxia Yao, Anwitaman Datta, Aixin Sun, and Bu-Sung Lee. Twiner: named entity recognition in targeted twitter stream. In *The 35th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '12, Portland, OR, USA, August 12-16, 2012*, pages 721–730. ACM, 2012.

[91] Chun Li and Mingyao Li. Gwasimulator: a rapid whole-genome simulation program. *Bioinformatics*, 24(1):140–142, 2007.

[92] Jorge Lloret-Gazo. A survey on visual query systems in the web era (extended version). *arXiv preprint arXiv:1708.00192*, 2017.

[93] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2006.

[94] Bertram Ludäscher, Mathias Weske, Timothy McPhillips, and Shawn Bowers. Scientific workflows: Business as usual? In *International Conference on Business Process Management*, pages 31–47. Springer, 2009.

[95] Carol M Lushbough, Etienne Z Gnimpieba, and Rion Dooley. Life science data analysis workflow development using the bioextract server leveraging the iplant collaborative cyberinfrastructure. *Concurrency and Computation: Practice and Experience*, 27(2):408–419, 2015.

[96] Xiaogang Ma, Linyun Fu, Peter Fox, and Gang Liu. An integrated golden spike information portal enabled by data visualization and semantic web technologies. In *Geostatistical and Geospatial Approaches for the Characterization of Natural Resources in the Environment*, pages 829–833. Springer, 2016.

[97] Xiaogang Ma, Daniel Hummer, Joshua J Golden, Peter A Fox, Robert M Hazen, Shaunna M Morrison, Robert T Downs, Bhuwan L Madhikarmi, Chengbin Wang, and Michael B Meyer. Using visual exploratory data analysis to facilitate collaboration and hypothesis generation in cross-disciplinary research. *ISPRS International Journal of Geo-Information*, 6(11):368, 2017.

[98] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with Cupid. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 49–58. Morgan Kaufmann, 2001.

[99] Philip Maechling, Hans Chalupsky, Maureen Dougherty, Ewa Deelman, Yolanda Gil, Sridhar Gullapalli, Vipin Gupta, Carl Kesselman, Jihic Kim, Gaurang Mehta, et al. Simplifying construction of complex workflows for non-expert users of the southern california earthquake center community modeling environment. *ACM SIGMOD Record*, 34(3):24–30, 2005.

[100] Luis N Marenco, Rixin Wang, Anita E Bandrowski, Jeffrey S Grethe, Gordon M Shepherd, and Perry L Miller. Extending the nif disco framework to automate complex workflow: coordinating the harvest and integration of data from diverse neuroscience information resources. *Frontiers in neuroinformatics*, 8:58, 2014.

[101] Jérôme Mariette, Frédéric Escudié, Philippe Bardou, Nabihoudine Ibouniyamine, Céline Noirot, Marie-Stéphane Trotard, Christine Gaspin, and Christophe Klopp. Jflow: a workflow management system for web applications. *Bioinformatics*, 32(3):456–458, 2016.

[102] Kim Marriott and Bernd Meyer. *Visual language theory.* Springer Science & Business Media, 2012.

[103] Shehadeh K Masalmeh, Xudong Jing, Sven Roth, Chenchen Wang, Hu Dong, Martin Blunt, et al. Towards predicting multi-phase flow in porous media using digital rock physics: workflow to test

the predictive capability of pore-scale modeling. In *Abu Dhabi International Petroleum Exhibition and Conference*. Society of Petroleum Engineers, 2015.

[104] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *World Wide Web Consortium recommendation*, 10(10):2004, 2004.

[105] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940. ACM, 2006.

[106] Saqib Mir, Steffen Staab, and Isabel Rojas. Web-prospector - an automatic, site-wide wrapper induction approach for scientific deep-web databases. In *Datenbanksysteme in Business, Technologie und Web*, pages 87–106. Gesellschaft für Informatik, Bonn, 2009.

[107] Aurélien Naldi. biolqm: a java library for the manipulation and conversion of logical qualitative models of biological networks. *bioRxiv*, page 287011, 2018.

[108] N Hari Narayanan and Roland Hübscher. Visual language theory: Towards a human-computer interaction perspective. pages 87–128, 1998.

[109] Hassana Nassiri, Mustapha Machkour, and Mohamed Hachimi. Integrating xml and relational data. *Procedia Computer Science*, 110:422–427, 2017.

[110] Thanh Hoang Nguyen, Hoa Nguyen, and Juliana Freire. PruSM: a prudent schema matching approach for web forms. In *Conference on Information and Knowledge Management*, pages 1385–1388. ACM, 2010.

[111] Jérôme Nika, Marc Chemillier, and Gérard Assayag. Improtek: introducing scenarios into human-computer music improvisation. *Computers in Entertainment (CIE)*, 14(2):4, 2016.

[112] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Mark Greenwood, Carole Goble, Anil Wipat, Peter Li, and Tim Carver. Delivering web service coordination capability to users. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 438–439. ACM, 2004.

[113] Tom Oinn, Mark Greenwood, Matthew Addis, M Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[114] Fatma Ozcan, Don Chamberlin, Krishna Kulkarni, and J-E Michels. Integration of sql and xquery in ibm db2. *IBM Systems Journal*, 45(2):245–270, 2006.

[115] María Constanza Pabón and César A Collazos. A visual query language for data graphs: an approach from the user-centered design. In *Proceedings of the XVI International Conference on Human Computer Interaction*, page 49. ACM, 2015.

[116] George Papadakis, Ekaterini Ioannou, Claudia Niederée, and Peter Fankhauser. Efficient entity resolution for large heterogeneous information spaces. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 535–544. ACM, 2011.

[117] Nikolaos Papadakis, Dimitrios Skoutas, Konstantinos Raftopoulos, and Theodora A. Varvarigou. Stavies: A system for information extraction from unknown web data sources through automatic web wrapper generation using clustering techniques. *IEEE Trans. Knowl. Data Eng.*, 17(12):1638–1652, 2005.

[118] Steven G Parker and Christopher R Johnson. Scirun: a scientific programming environment for computational steering. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 52. ACM, 1995.

[119] Phi Giang Pham and Mao Lin Huang. Mcquery: interactive visual query of relational data with coordinating context displays. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 47. ACM, 2016.

[120] Robert Pienta, Fred Hohman, Acar Tamersoy, Alex Endert, Shamkant Navathe, Hanghang Tong, and Duen Horng Chau. Visual graph query construction and refinement. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1587–1590. ACM, 2017.

[121] Robert Pienta, Acar Tamersoy, Hanghang Tong, Alex Endert, and Duen Horng Polo Chau. Interactive querying over large network data: Scalability, visualization, and interaction design. In *Proceedings of the 20th International Conference on Intelligent User Interfaces Companion*, pages 61–64. ACM, 2015.

[122] Lavanya Ramakrishnan, Sarah Poon, Valerie Hendrix, Daniel Gunter, Gilberto Z Pastorello, and Deborah Agarwal. Experiences with user-centered design for the tigres workflow api. In *2014 IEEE 10th International Conference on e-Science (e-Science)*, volume 1, pages 290–297. IEEE, 2014.

[123] Peter Rodgers, Gem Stapleton, John Howse, and Leishi Zhang. Euler graph transformations for euler diagram layout. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 111–118. IEEE, 2010.

[124] Mauno Rönkkö, Jani Heikkinen, Ville Kotovirta, and Venkatachalam Chandrasekar. Automated preprocessing of environmental data. *Future Generation Computer Systems*, 45:13–24, 2015.

[125] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual*. Pearson Higher Education, 2004.

[126] Carlos E Scheidegger, Huy T Vo, David Koop, Juliana Freire, and Claudio T Silva. Querying and re-using workflows with vstrails. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1251–1254. ACM, 2008.

[127] Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke. Pushing the boundaries of crowd-enabled databases with query-driven schema expansion. *Proceedings of the VLDB Endowment*, 5(6):538–549, 2012.

[128] Zohreh Shams, Mateja Jamnik, Gem Stapleton, and Yuri Sato. Reasoning with concept diagrams about antipatterns in ontologies. In *International Conference on Intelligent Computer Mathematics*, pages 255–271. Springer, 2017.

[129] Yannis Sismanis, Paul Brown, Peter J Haas, and Berthold Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the 32nd international conference on Very large data bases*, pages 691–702. VLDB Endowment, 2006.

[130] Richard N. Smith, Jelena Aleksic, Daniela Butano, Adrian Carr, Sergio Contrino, Fengyuan Hu, Mike Lyne, Rachel Lyne, Alex Kalderimis, Kim Rutherford, Radek Stepan, Julie Sullivan, Matthew Wakeling, Xavier Watkins, and Gos Micklem. InterMine: a flexible data warehouse system for the integration and analysis of heterogeneous biological data. *Bioinformatics*, 28(23):3163–3165, 2012.

[131] Ahmet Soylu, Martin Giese, Rudolf Schlatte, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Özgür Özçep, Christian Neuenstadt, and Sebastian Brandt. Querying industrial stream-temporal data: An ontology-based visual approach 1. *Journal of Ambient Intelligence and Smart Environments*, 9(1):77–95, 2017.

[132] Ahmet Soylu, Evgeny Kharlamov, Dmitriy Zheleznyakov, Ernesto Jimenez-Ruiz, Martin Giese, and Ian Horrocks. Optiquevqs: Ontology-based visual querying. In *VOILA@ ISWC*, page 91. Aachen: CEUR, 2015.

[133] Alessandro Spinuso, Rosa Filgueira, Amrey Krause, Jonas Matser, Emanuele Casarotti, Federica Magnoni, Andre Gemund, Laurent Frobert, Lion Krischer, and Malcolm Atkinson. The verce platform: Enabling computational seismology via streaming workflows and science gateways. In *EGU General Assembly Conference Abstracts*, volume 17. European Geosciences Uniony, 2015.

[134] Aravind Subramanian, Pablo Tamayo, Vamsi K. Mootha, Sayan Mukherjee, Benjamin L. Ebert, Michael A. Gillette, Amanda Paulovich, Scott L. Pomeroy, Todd R. Golub, Eric S. Lander, and Jill P. Mesirov. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences of the United States of America*, 102(43):15545–15550, October 2005.

[135] Wei Tan, Paolo Missier, Ravi K. Madduri, and Ian T. Foster. Building scientific workflow with Taverna and BPEL: A comparative study in caGrid. In *ICSOC Workshops*, pages 118–129. Springer, 2008.

[136] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. In Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 320–339. Springer, New York, 2007.

[137] Condor Team. Dagman: A directed acyclic graph manager. *See website at http://www. cs. wisc. edu/condor/dagman*, 2005.

[138] James F Terwilliger, Lois ML Delcambre, and Judith Logan. Querying through a user interface. *Data & Knowledge Engineering*, 63(3):774–794, 2007.

[139] Brian Thomas and Edward Shaya. A user interface for semantically oriented data mining of astronomy repositories. *arXiv preprint arXiv:1502.06492*, 2015.

[140] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994.

[141] Guilherme A Toda, Eli Cortez, Altigran S da Silva, and Edleno de Moura. A probabilistic approach for automatically filling form-based web interfaces. *Proceedings of the VLDB Endowment*, 4(3):151–160, 2010.

[142] Wil MP Van der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013, 2013.

[143] Ben P Vandervalk, E Luke McCarthy, and Mark D Wilkinson. Share: A web service based framework for distributed querying and reasoning on the semantic web. *arXiv preprint arXiv:1305.4455*, 2013.

[144] Guillermo Vega-Gorgojo, Laura Slaughter, Martin Giese, Simen Heggestøyl, Ahmet Soylu, and Arild Waaler. Visual query interfaces for semantic datasets: An evaluation study. *Web Semantics: Science, Services and Agents on the World Wide Web*, 39:81–96, 2016.

[145] Gregor Von Laszewski, Mihael Hategan, and Deepti Kodeboyina. Java cog kit workflow. In *Workflows for e-Science*, pages 340–356. Springer, 2007.

[146] Han Wang, Jin Guang Zheng, Xiaogang Ma, Peter Fox, and Heng Ji. Language and domain independent entity linking with quantified collective validation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 695–704. ACM, 2015.

[147] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[148] Mark D. Wilkinson, Benjamin P. Vandervalk, and E. Luke McCarthy. The semantic automated discovery and integration (SADI) web service design-pattern, API and reference implementation. *J. Biomedical Semantics*, 2:8, 2011.

[149] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

[150] Lei Wu, Yang Hu, Mingjing Li, Nenghai Yu, and Xian-Sheng Hua. Scale-invariant visual language modeling for object categorization. *IEEE Transactions on Multimedia*, 11(2):286–294, 2009.

[151] Zongda Wu, Guandong Xu, Yanchun Zhang, Zhongsheng Cao, Guiling Li, and Zhiwen Hu. Gmql: A graphical multimedia query language. *Knowledge-Based Systems*, 26:135–143, 2012.

[152] Mohamed Yakout, Ahmed K Elmagarmid, Hazem Elmeleegy, Mourad Ouzzani, and Alan Qi. Behavior based record linkage. *Proceedings of the VLDB Endowment*, 3(1-2):439–448, 2010.

[153] Peipei Yi, Byron Choi, Sourav S Bhowmick, and Jianliang Xu. Autog: a visual query autocompletion framework for graph databases. *The International Journal on Very Large Data Bases*, 26(3):347–372, 2017.

[154] Ustun Yildiz, Adnene Guabtni, and Anne HH Ngu. Business versus scientific workflows: A comparative study. In *2009 World Conference on Services-I*, pages 340–343. IEEE, 2009.

[155] Boliang Zhang, Xiaoman Pan, Tianlu Wang, Ashish Vaswani, Heng Ji, Kevin Knight, and Daniel Marcu. Name tagging for low-resource incident languages based on expectation-driven learning. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 249–259. Association for Computational Linguistics, 2016.

[156] Shuai Zhao and Quan Qian. Ontology based heterogeneous materials database integration and semantic query. *AIP Advances*, 7(10):105325, 2017.

[157] Yongjun Zhu and Erjia Yan. Searching bibliographic data using graphs: A visual graph query interface. *Journal of Informetrics*, 10(4):1092–1107, 2016.

Appendix A: BioFlow Querying Language

A.1 BioFlow Language Definitions

One of the most powerful features of VisFlow is its high-level abstraction of the underlying resources. As mentioned earlier, all data resources are viewed as nested relational tables, all web services, and analytics as functions, and a distributed workflow as a graph. Mapping such workflows to underlying resources is viewed as an approximate graph matching problem over the resources indexed in the VisFlow library. An indefinite stepwise mapping creates the abstract view of each resource to the lowest level of detail. To understand how the stepwise mapping is modeled, consider three data repositories d_1, d_2 and d_3. Assume that d_1 is a traditional deep web database interface exposed as a simple HTML web form which upon submission of the required inputs, returns an HTML page with a set of objects we call records which can be in the form of a table or a semi-structured nested table. d_2, on the other hand, is a web service, which returns an XML document when properly called. Finally, d_3 is a local text data set on a local machine.

Since all three views are relational at the VisFlow level, all references are in the form of a table name with a flat set of attributes. An index looks up then exposes the fact that d_1 is an in-depth web interface requiring a (possibly empty) set of inputs, resides at an URL u, and returns a flat set of records in HTML form. An additional lookup reveals that to access this deep web table, a wrapper is necessary, and to resolve schema heterogeneity a schema matcher is also needed. At this stage, a second mapping to identify a suitable wrapper and a matcher is performed based on the deep web properties. This information then is used to generate codes to access the table.

In VisFlow, we have transformed it into a declarative language when executing a workflow, which is inspired by a recently proposed declarative language for data integration, called BioFlow [74, **?**], as it already includes the mechanism schema matching and wrapping, in addition to accessing remote sites. We have extended and improved this idea to design the new syntax and make it suitable for different data resources, support complex function calls and enrich the functionaries of it for the VisFlow application.

A.1.1 Extract Statement

The statement to access deep web resources in BioFlow is the extract statement with the following syntax,

EXTRACT $\alpha_1, \ldots, \alpha_k$

USING MATCHER μ WRAPPER ω Filler ϕ

FROM φ

SUBMIT $r(\beta_1, \ldots, \beta_k)$

where $\alpha_1, \ldots, \alpha_k$ is the projection list, μ is the schema matcher, ω is the wrapper, ϕ is form filler, r is the resource and $\beta_1, \ldots, \beta_k$ is the attributes in the resource. This statement returns a table by submitting columns from each tuple in r to the deep web resources at φ, and can be constructed based on the description of the web service. When a matcher, wrapper or filler isn't necessary, the corresponding clauses are omitted. This feature of VisFlow allows users to write applications without worrying about the details of the web service protocol, location, return data format, and schema heterogeneity. etc. What's more, VisFlow keeps a global mapping table that allows users to conceptualize a complete workflow at the highest abstraction level knowing that the underlying data management and integration apparatus will be able to map the application onto the potentially heterogeneous resources correctly and efficiently without any loss in query semantics.

A.1.2 Call Statement

The call statement in BioFlow is the statement that will execute executable functions that are registered in the VisFlow workflow management system. The call statement has the following syntax:

CALL FUNCTION f WITH r

where f is the function and r is the dataset passed in. In VisFlow, we support Python, SQL, R, XQuery, and any third party libraries scripts that can be called from the windows command line. The same as web services, users only need to know the purpose of the functions and don't worry about the implementation. These libraries are easy to import in the workflow for reuse. Users are allowed to write their library functions and register them in the VisFlow system.

A.1.3 Combine and Link Statement

The combine and link statements are used to support horizontal data integration and vertical integration. The combine statement is defined as the follows:

COMBINE r, s

USING MATCHER μ IDENTIFIER ω

FROM φ

and the link statement is defined as the following:

LINK r, s

USING MATCHER μ IDENTIFIER ω

FROM φ

where r and s are two relational tables, μ are schema matchers, and ω are key identifiers. Using combine and link statement has two steps. The first step is applying the key identifier algorithms to identify the possible composite key pairs in two tables, and then schema matchers select the corresponding key pairs from the two tables. In the end, the SQL join and union are applied based on the keys for these tables to merge into one.

A.1.4 If and Repeat Statement

The following is the definition of if syntax:

IF(c) THEN p_1 [ELSE p_2]

The use of the supports branching if statement condition c is the valid logical comparator involving attributes in the data resource. The else branch is optional. p_1 and p_2 are both process type, which are collections of statements that will be explained later. Looping is also supported in BioFlow using repeat until statement, where the definition is:

REPEAT p UNTIL(c)

c is also the valid logical comparator for attributes in the data resource, and in repeat VisFlow has a internal counter that also supports executing the workflow at certain times. p is also a process.

A.1.5 Pause Statement

The pause statement is directly mapping the GeneralIO icon in order to wait for user interactions. The following is the definition of pause syntax:

PAUSE $r_1 \ldots r_k$

where $r_1 \ldots r_k$ are list of data resources to wait.

A.1.6 Process, Perform and Wait on Statement

To perform a parallel execution on workflow, the process and wait on statements are needed. Process is an execution block that contains a series of statements which cannot be nested using a pair of curly braces. The syntax of process is defined as follows:

PROCESS P:{

$statement_1$;

. . .

statement$_k$;}

The execution statement uses the perform keyword to initialize the execution and it is defined as:

PERFORM PARALLEL $p_1, \ldots, p_k$ [AFTER $q_1, \ldots, q_k$]

This is sophisticated and allows user control it as needed. After is not required. $q_1, \ldots, q_k$ in parallel executed first; after they are all finished, $p_1, \ldots, p_k$ start to execute. The wait on statement is pending until all the processes have finished executing in the workflow, which is defined as follows:

WAIT ON $p_1, \ldots, p_k$

A.2 BioFlow Scripts and Resources for All Examples

This section list all the BioFlow scripts and depended resources that used in this dissertation.

A.2.1 Pathway Difference in Orthology Genes Example

```
PROCESS retrieve-orthology{
EXTRACT ko,hsa
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/orthology/hsa:
SUBMIT humangenes(id); }
PROCESS retrieve-pathways{
EXTRACT ko,gene
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/genes/
SUBMIT orthology(id); }
PROCESS generate-rno-mmu{
CALL generate rno and mmu WITH (orthologygene,orthology); }
PROCESS retrieve-mmu{
EXTRACT hsa,pathway
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/pathway/
SUBMIT pathrnommu(hsa); }
PROCESS retrieve-rno{
EXTRACT hsa,pathway
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/pathway/
SUBMIT pathrnommu(hsa); }
PROCESS add-prefix{
CALL addprefix WITH (humangenes); }
PROCESS retrieve-hsa{
EXTRACT hsa,pathway
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/pathway/
SUBMIT hsagenes(hsa); }
PROCESS genearate-differences{
CALL comparetables WITH (hsapath,mmupath,rnopath); }

```

Listing A.1: BioFlow Processes for Pathway Difference in Orthology Genes in Figure 7.3

```
{ codeName: generate rno and mmu
codeType: sql
code: drop table if exists pathrnommu;
create table pathrnommu as
select m1.hsa, m1.gene mmu, m2.gene rno
from
(
select o1.hsa, o1.ko, g1.gene
from orthology o1, orthologygene g1
where o1.ko = g1.ko and g1.gene like 'mmu%' group by o1.hsa
) m1
join
(
select o1.hsa, o1.ko, g1.gene
from orthology o1, orthologygene g1
where o1.ko = g1.ko and g1.gene like 'rno%' group by o1.hsa
) m2
on m1.hsa = m2.hsa;}
{ codeName: addprefix
codeType: sql
code: SET SQL_SAFE_UPDATES = 0;
update  humangenes humangenes
set  humangenes.gene = concat('hsa:',humangenes.gene);
drop table if exists hsagenes;
create table hsagenes
select * from humangenes;
SET SQL_SAFE_UPDATES = 1;}
{ codeName: comparetables
codeType: sql
code: drop table if exists hsawhole;
create table hsawhole as
select   hsapath.pathway , (substring(hsapath.pathway, 9, 6)) geneid, (@nh := @nh + 1) id
from  hsapath hsapath
inner join
(select (@nh := 0)) m;
delete  h1
from  hsawhole h1, hsawhole h2
where  h1.geneid = h2.geneid and h1.id > h2.id ;
drop table if exists hsadict;
create table hsadict
select * from hsawhole;
drop table if exists mmuwhole;
create table mmuwhole as
select   mmupath.pathway , (substring(mmupath.pathway, 9, 6)) geneid, (@nm := @nm + 1) id
from  mmupath mmupath
inner join
(select (@nm := 0)) m;

```

Listing A.2: Resources for Pathway Difference in Orthology Genes (Part 1)

```
delete  m1
from  mmuwhole m1, mmuwhole m2
where  m1.geneid = m2.geneid and m1.id > m2.id ;
drop table if exists mmudict;
create table mmudict
select * from mmuwhole;
drop table if exists hsasamemmu;
create table hsasamemmu as
select  (substring(hsadict.pathway, 9, 5)) pathway
from  hsadict hsadict, mmudict mmudict
where  mmudict.geneid = hsadict.geneid and hsadict.pathway != '';
drop table if exists hsadiffmmu;
create table hsadiffmmu as
select substring(hsadict.pathway, 6, 9) diff_mmu
from  hsadict hsadict
where not exists (select geneid from mmudict where hsadict.geneid = geneid) and hsadict.
   pathway is not null union select mmudict.pathway from mmudict mmudict where not
   exists (select geneid from hsadict where hsadict.geneid = geneid);
drop table if exists rnowhole;
create table rnowhole as
select   rnopath.pathway , (substring(rnopath.pathway, 9, 6)) geneid, (@nr := @nr + 1) id
from  rnopath rnopath
inner join
(select (@nr := 0)) m;
delete  r1
from  rnowhole r1, rnowhole r2
where  r1.geneid = r2.geneid and r1.id > r2.id ;
drop table if exists rnodict;
create table rnodict
select * from rnowhole;
drop table if exists hsasamerno;
create table hsasamerno as
select  (substring(hsadict.pathway, 9, 5)) pathway
from  hsadict hsadict, rnodict rnodict
where  rnodict.geneid = hsadict.geneid and hsadict.pathway != '' ;
drop table if exists hsadiffrno;
create table hsadiffrno as
select substring(hsadict.pathway, 6, 9) diff_rno
from  hsadict hsadict
where not exists (select geneid from rnodict where hsadict.geneid = geneid) and hsadict.
   pathway is not null union select rnodict.pathway from rnodict rnodict where not
   exists (select geneid from hsadict where hsadict.geneid = geneid);}

```

Listing A.3: Resources for Pathway Difference in Orthology Genes (Part 2)

```
{ resource type:CSV
resource name:humangenes}
{ resource type:REST
resource name:GeneIdToPathwayId
method:REST
url:http://rest.kegg.jp/link/pathway/{hsa}}
{ resource type:CSV
resource name:orthology}
{ resource type:CSV
resource name:hsagenes}
{ resource type:CSV
resource name:orthologygene}
{ resource type:REST
resource name:GetListOfGenes
method:REST
url:http://rest.kegg.jp/link/genes/{id}}

```

Listing A.4: Resources for Pathway Difference in Orthology Genes (Part 3)

A.2.2 Minerals Evolution 3D Exploration

```
PROCESS retrieveMineral {
EXTRACT H, Na, K, Mg, Ca, Ti, V, Mn, Fe, Ni, Cu, Ag, Zn, B, Al, C, Si, Pb, Sm
USING MATCHER S-Match WRAPPER FastWrap
FROM http://rruff.info/mineral_list/element_match.php
SUBMIT elements (focus); }
PROCESS generateMatrix {
CALL OgLwXDS  WITH (elements,mineral); }

```

Listing A.5: BioFlow Processes for Minerals Evolution 3D Exploration in Figure 6.18

```
{ codeName: OgLwXDS
codeType: sql
code: drop table if exists cubeRes;
create table cubeRes as
select f2.focus, column0, H,Na,K,Mg,Ca,Ba,Ti,V,Mn,Fe,Ni,Cu,Ag,Zn,B,Al,C,Si,Pb,Sm
from
(
select @ln := @ln + 1 ln, floor(@ln / 72) gp, m.*
from mineral m
join (SELECT @ln:=-1) t1
) f1
join
(
select focus, @rn := @rn + 1 rn
from elements
join (SELECT @rn:=-1) t2
) f2
on f1.gp = f2.rn and column0 in (select * from elements)
order by focus, column0}
{ resource type:CSV
resource name:mineral}
{ resource type:CSV
resource name:cubeRes}
{ resource type:CSV
resource name:elements}

```

Listing A.6: Resources for Minerals Evolution 3D Exploration

A.2.3 KEGG Pathway Example

```
PROCESS addBefore{
CALL addbefore.py WITH (genes); }
PROCESS retrieveNCBIIds{
EXTRACT ncbigeneid,hsa
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/conv/hsa/
SUBMIT ncbi-id(ncbi-geneid); }
PROCESS retrievePathwayIDs{
EXTRACT pathwayid,hsaid
USING MATCHER S-Match WAPPER FastWrap
FROM http://rest.kegg.jp/link/pathway/
SUBMIT ncbipairs(hsa-id); }
PROCESS extractPathway{
CALL extract.py WITH (pathwaypairs); }

```

Listing A.7: BioFlow Processes for KEGG Pathway Example in Figure 7.2

```
{ codeName: addbefore.py
codeType: python
code: def addbefore():
handle = open('genes.csv', 'r');
target = open('ncbi-id.csv', 'w');
target.write('"ncbigeneid"\n');
i = 0;
for line in handle:
if i > 0:
words = line.split(',');
target.write('"ncbi-geneid:' + words[1].replace('"', '') + '"\n');
i += 1;
addbefore();}
{ codeName: extract.py
codeType: python
code: def extractPathway():
handle = open('pathwaypairs.csv', 'r');
target = open('pathway.csv', 'w');
target.write('"pathway-id"\n');
i = 0;
for line in handle:
if i > 0:
line=line.replace('"','');
word = line.split(":");
target.write('"' + word[2].strip() + '"\n');
i += 1;
extractPathway();}
{ resource type:CSV
resource name:pathwaypairs}
{ resource type:CSV
resource name:genes}
{ resource type:REST
resource name:KEGG ncbi-geneid to hsa
method:REST
url:http://rest.kegg.jp/conv/hsa/{ncbi-geneid}}
{ resource type:REST
resource name:KEGG hsa to pathway id
method:REST
url:http://rest.kegg.jp/link/pathway/{hsa-id}}
{ resource type:CSV
resource name:pathway}
{ resource type:CSV
resource name:ncbipairs}
{ resource type:CSV
resource name:ncbi-id}

```

Listing A.8: Resources for KEGG Pathway Example

A.2.4 Genome Analytics Example

```
PROCESS genomeAnalytics {
CALL analysis.bat WITH (P1,P1,genome); }

```

Listing A.9: BioFlow Processes for Genome Analytics Example in Figure 7.5

```
{ codeName: analysis.bat
codeType: bash
code: ./bwa mem genome.fa P1.R1.fastq.gz P1.R2.fastq.gz > bam.mem
java -jar picard.jar SortSam I=bam.mem O=Sorted_bam SO=coordinate
java -jar picard.jar MarkDuplicates INPUT=Sorted_bam OUTPUT=bam.MarkDuplicates
    METRICS_FILE=bam.MarkDuplicates.metrics
java -jar picard.jar AddOrReplaceReadGroups I=bam.MarkDuplicates O=Validated_bam.bam RGID=
    bam.AddOrReplaceReadGroups RGSM=P1 RGCN=@HWI-ST932 RGPL=ILLUMINA RGPU=C5JGVACXX-.2
    RGLB=Nextera
java -jar picard.jar BuildBamIndex INPUT=Validated_bam.bam OUTPUT=Validated_bam.bai
java -jar picard.jar ValidateSamFile INPUT=Validated_bam.bam OUTPUT=Validated_bam.val
    VALIDATE_INDEX=true MODE=SUMMARY
java -jar picard.jar BuildBamIndex INPUT=Validated_bam.bam OUTPUT=Validated_bam.bai
java -jar GenomeAnalysisTK.jar -T RealignerTargetCreator -R genome.fa -I Validated_bam.bam
     -o bam.AddOrReplaceReadGroups.intervals -known dbsnp.vcf
java -jar GenomeAnalysisTK.jar -T IndelRealigner -R genome.fa -I Validated_bam.bam -
    targetIntervals bam.AddOrReplaceReadGroups.intervals -o bam.IndelRealigner.bam -known
     dbsnp.vcf -LOD 0.4 -model USE_READS -compress 0 --disable_bam_indexing
java -jar picard.jar BuildBamIndex INPUT=bam.IndelRealigner.bam OUTPUT=bam.IndelRealigner.
    bai
java -jar GenomeAnalysisTK.jar -T BaseRecalibrator -R genome.fa -I bam.IndelRealigner.bam
    -o bam.BaseRecalibrator.csv -knownSites dbsnp.vcf -l INFO
java -jar GenomeAnalysisTK.jar -T PrintReads -R genome.fa -I bam.IndelRealigner.bam -BQSR
    bam.BaseRecalibrator.csv -o bam.PrintReads.bam}
{ resource type:CSV
resource name:bam}

```

Listing A.10: BioFlow Processes for Genome Analytics Example Resources

A.2.5 US Fossil Collections Example

```
PROCESS getGeologicalName {
CALL getGeologicalName WITH (age); }
PROCESS retrieveFossilCollection {
EXTRACT occurrence_no,record_type,reid_no,flags,collection_no,identified_name,
    identified_rank,identified_no,difference,accepted_name,accepted_rank,accepted_no,
    early_interval,late_interval,max_ma,min_ma,reference_no,lng,lat,paleomodel,paleolng,
    paleolat,geoplate,cc,state,county,latlng_basis,latlng_precision,geogscale,
    geogcomments
USING MATCHER S-Match WRAPPER FastWrap
FROM https://paleobiodb.org/data1.2/occs/list.txt
SUBMIT geoTerm (min_ma, max_ma); }
PROCESS getLocations {
CALL getLocations WITH (fossilRecords); }
PROCESS retrieveOldCoordinates {
EXTRACT coordinates
USING MATCHER S-Match WRAPPER FastWrap
FROM https://gws.gplates.org/reconstruct/reconstruct_points/
SUBMIT gplateInputs (points, time); }
PROCESS mergeResults {
CALL mergeResults WITH (fossilRecords, oldCoordinates); }

```

Listing A.11: BioFlow processes for US Fossil Collections Example in Figure 7.6

```
{ codeName: generateGPlateInputs.py
codeType: python
code: import csv;
from collections import defaultdict;
def f7(seq):
seen = set()
seen_add = seen.add
return [x for x in seq if not (x in seen or seen_add(x))]
columns = defaultdict(list)
with open('fossilRecords.csv') as f:
reader = csv.DictReader(f) # read rows into a dictionary format
for row in reader: # read a row as {column1: value1, column2: value2,...}
for (k,v) in row.items(): # go over each column name and value
columns[k].append(v)
f.close();
locationTuples = list();
lngs = columns['lng'];
lats = columns['lat'];
bases = columns['max_ma'];
tops = columns['min_ma'];
index = 0;
for lng in lngs:
if float(lngs[index]) > -180 and float(lngs[index]) < 180 and float(lats[index]) > -180
   and float(lats[index]) < 180:
locationTuples.append((bases[index], tops[index], lngs[index], lats[index]));
index = index + 1;
locationTuples = f7(locationTuples);
print locationTuples;
target = open('gplatIntputs.xml', 'w');
target.write("\n");
age = (float(locationTuples[0][0]) + float(locationTuples[0][1])) / 2.0;
for tuple in locationTuples:

```

Listing A.12: Resources for US Fossil Collections Example (Part 1)

```
target.write("");
target.write("");
target.write(tuple[2] + "," + tuple[3]);
target.write("");
target.write("");
target.write(str(age));
target.write("");
target.write("\n");
target.write("\n");
target.close();}
{ codeName: combineTogether.py
codeType: python
code: import csv;
import xml.etree.ElementTree
import codecs
from collections import defaultdict;
def f7(seq):
seen = set()
seen_add = seen.add
return [x for x in seq if not (x in seen or seen_add(x))]
fossilRecrods = defaultdict(list)
with open('fossilRecords.csv') as f:
reader = csv.DictReader(f) # read rows into a dictionary format
for row in reader: # read a row as {column1: value1, column2: value2,...}
for (k,v) in row.items(): # go over each column name and value
fossilRecrods[k].append(v)
f.close();
locationTuples = list();
lngs = fossilRecrods['lng'];
lats = fossilRecrods['lat'];
reference = fossilRecrods['reference_no'];
index = 0;
for lng in lngs:
if float(lngs[index]) > -180 and float(lngs[index]) < 180 and float(lats[index]) > -180
    and float(lats[index]) < 180:
locationTuples.append((lngs[index], lats[index]));
index = index + 1;
locationTuples = f7(locationTuples);
e = xml.etree.ElementTree.parse('oldCoordinats.xml').getroot();
longList = list();
latList = list();
for line in e:

```

Listing A.13: Resources for US Fossil Collections Example (Part 2)

```
if line.tag == 'coordinates':
long = line.find('.//array[1]').text;
lat = line.find('.//array[2]').text;
longList.append(long);
latList.append(lat);
elif line.tag == 'ConverJSONError':
longList.append(long);
latList.append(lat);
locationCoord = dict();
index = 0;
for item in locationTuples:
print item;
locationCoord[(item[0], item[1])] = (longList[index], latList[index]);
index = index + 1;
target = open('fossilResultTable.csv', 'w');
fossils = open('fossilRecords.csv', 'r');
attrs = next(fossils);
target.write(attrs.strip() + ",\"oldlat\",\"oldlng\"\n");
length = len(attrs.split("\""));
print length;
index = 0;
for line in fossils:
if float(lngs[index]) > -180 and float(lngs[index]) < 180 and float(lats[index]) > -180
   and float(lats[index]) < 180:
appendLine = locationCoord[(lngs[index], lats[index])];
leng = len(line.split("\""));
print leng;
if length == leng:
target.write(line.strip() + ",\"" + appendLine[0] + "\",\"" + appendLine[1] + "\"\n");
index = index + 1;
fossils.close();
target.close();}
{ resource type:CSV
resource name:fossilRecords}
{ resource type:CSV
resource name:fossilResultTable}
{ resource type:CSV
resource name:geoTerm}

```

Listing A.14: Resources for US Fossil Collections Example (Part 3)

A.2.6 Jurassic Time Period Fossil Information Integration Example

```
PROCESS retrieveGeologicalLayer{
EXTRACT Name,ogcPropertyName,ogcLiteral,CssParameter
USING MATCHER S-Match WAPPER FastWrap
FROM http://ogc.bgs.ac.uk/cgi-bin/BGS_Bedrock_and_Superficial_Geology/wms
SUBMIT input(layers,version,REQUEST,SERVICE,format); }
PROCESS retrieveAgeTable{
EXTRACT datatype,type,value
USING MATCHER S-Match WAPPER FastWrap
FROM http://geotime.tw.rpi.edu/fuseki/ds/query
SUBMIT input(query,output); }
PROCESS extractFeatureTypes{
CALL extractFeatureTypes WITH (layers); }
PROCESS extractAgeLabel{
CALL extractAgeLabel WITH (ageTable); }
PROCESS transformTitle{
CALL transformTitle WITH (featuretypes); }
PROCESS extractTitle{
CALL extractTitle WITH (titledLieral); }
PROCESS generateSPARQLQuery{
CALL generateSPARQLQuery WITH (title); }
PROCESS retrieveGeotime{
EXTRACT datatype,type,value
USING MATCHER S-Match WAPPER FastWrap
FROM http://geotime.tw.rpi.edu/fuseki/ds/query
SUBMIT sparqlQuery(query,output); }
PROCESS extractLabel{
CALL extractLabel WITH (gsspLabel); }
PROCESS mergeGsspInfo{
CALL mergeGsspInfo WITH (title,gsspAll); }
PROCESS generateGsspInfo{
CALL generateGsspInfo WITH (gsspinfoTmp); }
PROCESS extractGssp{
CALL extractGssp WITH (gsspinfo); }
PROCESS mergeGssp{
CALL mergeGssp WITH (gsspwhole,gssp); }

```

Listing A.15: BioFlow Processes for Jurassic Time Period Fossil Information Integration in Figure 7.8 (Part 1)

```
PROCESS mergeGsspDetail{
CALL mergeGsspDetail WITH (gsspDetail,titledLieral); }
PROCESS addWiki{
CALL addWiki WITH (gsspmerged); }
PROCESS generatePurdueGsspLink{
CALL generatePurdueGsspLink WITH (gssptable); }
PROCESS generateGplateInfo{
CALL generateGplateInfo WITH (fossilinfo); }
PROCESS generateFossileTable{
CALL generateFossileTable WITH (fossilinfo); }
PROCESS generateGplate{
CALL generateGplate WITH (gplateinfo); }
PROCESS generateGplateInput{
CALL generateGplateInput WITH (gplate); }
PROCESS retrieveLayer{
EXTRACT coordinates
USING MATCHER S-Match WAPPER FastWrap
FROM https://gws.gplates.org/reconstruct/reconstruct_points/
SUBMIT gplateinput(points,time); }
PROCESS extractCooridinates{
CALL extractCooridinates WITH (gsspCoord); }
PROCESS mergecomTable{
CALL mergecomTable WITH (gsspCoord,fossiltable); }

```

Listing A.16: BioFlow Processes for Jurassic Time Period Fossil Information Integration in Figure 7.8 (Part 2)

```
{ resource type:CSV
resource name:gsspDetail}
{ resource type:CSV
resource name:gssptable}
{ resource type:CSV
resource name:fossilinfo}
{ resource type:CSV
resource name:title}
{ resource type:CSV
resource name:gssp}
{ resource type:CSV
resource name:comTable}
{ resource type:CSV
resource name:titledLieral}
{ resource type:CSV
resource name:gplateinfo}
{ resource type:CSV
resource name:gsspinfo}
{ resource type:CSV
resource name:gsspCoord}
{ resource type:CSV
resource name:featuretypes}
{ resource type:CSV
resource name:gsspwhole}
{ resource type:CSV
resource name:gplate}
{ resource type:CSV
resource name:gplateinput}
{ resource type:CSV
resource name:fossiltable}
{ resource type:CSV
resource name:gsspinfoTmp}
{ resource type:CSV
resource name:gsspmerged}

```

Listing A.17: Resources for Jurassic Time Period Fossil Information Integration