daGui: A DataFlow Graphical User Interface

Adam Uhlir

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Kista, Sweden 10. 6. 2017

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

Assoc. Prof. Jim Dowling

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Adam Uhlir

Title: daGui: A DataFlow Graphical User Interface

Date: 10. 6. 2017 Language: English Number of pages: 8+50

Department of Computer Science

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: Assoc. Prof. Jim Dowling

Big Data is a growing trend. It focuses on storing and processing a vast amount of data in a distributed environment. There are many frameworks and tools which can be used to work with this data. Many of them utilise Directed Acyclic Graphs (DAGs) in some way. A DAG is often used for expressing the dataflow of computation as it offers the possibility to optimise the execution, because it contains the overview of the whole computation. This thesis aims to create an Integrated Development Environment (IDE) like software, which is user-friendly, interactive and easily extendable. The software enables to draw a DAG which represents the dataflow of a program. The DAG is then transformed into launchable source code. Moreover, the software offers a simple way to execute the generated source code. It compiles the code (if necessary), and launches it based on the user's configuration, either on localhost or cluster. The software primarily aims to help beginners learn these technologies, but experts can also use it as visualisation for their workflow or as a prototyping tool. The software has been implemented using Electron and Web technologies, which ensure its platform independence. Its main features are code generation (i.e. translation of a DAG into source code) and code execution. It is created with extensibility in mind, to be able to plug-in support for more frameworks and tools in the future.

Keywords: Big Data, Apache Spark, DAG, dataflow, GUI

Acknowledgements

I am very grateful to my supervisor, Amir H. Payberah, for his guidance and help during the thesis writing process. He welcomed me with open arms and was always ready to spend time to explain me the right way or his point of view on a problem.

My thanks also belong to my next supervisor, Keijo Heljanko, who gave me valuable feedback during the writing process and offered me many bits of advice.

I would also like to thank Jim Dowling for making it possible to create this thesis and his valuable feedback.

Lastly, big thanks belong to my friend Peter Sykora, who created the visual design of daGui and helped me with styling problems.

Kista, Sweden, 10.6.2017

Adam Uhlir

Contents

1 Introduction

Data — the primary drive of our time. The birth of the Internet enabled easy communication and exchange of data across any distances. In its beginning, the usage was very limited, but after several decades the Internet became an important part of our lives. People use the World-Wide-Web to access information, E-Mails to communicate and more recently the uprise of social networks made it possible to share small bits of everybody's daily lives with their surroundings. But this visible type of data is just the tip of the "data iceberg". Just the data exchange itself generates information (traffic logs, server logs and so on). Many companies understood that they need to monitor their infrastructure (for example electric grid, highway traffic and so on) and lastly the Internet of Things promises to interconnect a vast amount of devices. All these aspects generate secondary data, which has its primary meaning (for example logs primarily serve as a tool for the system administrators to resolve issues), but when the amount of the data is big (units, hundreds, thousands and more of terabytes), additional processing can bring valuable insights.

The storing and processing of such a large amount of data brings new challenges and problems. To tackle these issues, there was an important shift toward a distributed environment, since no single monolith server can store or process that much data in a reasonable manner (processing time, price of hardware and so on). To support such a new paradigm, the community created new projects, tools and frameworks, which require a slightly different mindset while working with them as the distributed environment possesses specific restrictions and characteristics. To tackle these challenges, authors of several of the frameworks used Directed-Acyclic-Graph (DAG) dataflow, for example, Apache Spark [29], TensorFlow [14] and more. The authors usually employ the DAG for defining the dataflow of computation, which is then used for planning the execution as it offers ways to optimise it. The developers do not necessarily need to come in touch with the DAG representation, but for the in-depth understanding of the technology and advanced usage such as tweaking the performance of the programs, the understanding is critical.

This thesis aims to create a simple integrated development environment (IDE) like software, which will ease the learning curve of earlier described technologies based on DAG dataflow execution. The software has to have an easy-to-use environment, with high interactivity to create a playground for beginners, where they can easily explore the technologies without the big hassle of setting up the environment (as simple as download, install and use). For advanced users, this software can help them to present their programs, as it offers a nice way to visualise them. Lastly, it can be used as a prototyping tool, as some technologies require more thinking about the program than others. Developers write less code but need to think more about its function. An example of such technology can be TensorFlow, which focuses on distributed machine learning. For these kinds of technologies, the IDE-like software can bring valuable visualisation of the program, which helps the developer's mental process and therefore eases the development.

Section 2 introduces concepts and an overview of distributed computation. Section 3 presents the high-level design of the software, which is created as part of this

thesis. Section 4 presents used technologies and implementation details. Section 5 describes the referential implementation of the Spark adapter and how to implement a custom adapter. Section 6 evaluates the results of the software and Section 7 offers concluding remarks.

1.1 Contribution

The main contribution of this thesis is the creation of the IDE-like software which is released under an Open Source licence. Therefore, it is easily accessible to the whole community and ready for further development, if the community will find the software useful.

2 Background

This Section will present the fundamental information to understand the context of the thesis and the software which was created as part of this thesis.

Section 2.1 explains the different types of distributed environments and their properties. Section 2.2 presents the basic overview of Hadoop, which is the main platform for Big Data. After that, the definition of Directed Acyclic Graph (DAG) and its properties are presented in Section 2.3. In following Section 2.4 introduces several frameworks which utilise a DAG in one way or another. The last Section 2.5 surveys projects which are similar or related to this thesis.

2.1 Cloud Computing

The concept of Cloud Computing can be hard to grasp. There are several definitions which specify its attributes. The most widely accepted definition is from the *National Institute of Standards and Technology (NIST)* [20]. It defines five basic Cloud characteristics: *on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service*. Moreover, it defines two models – *service model and deployment model*.

Service model defines what kind of interaction users have with the cloud service. It splits the interaction into three levels based on what area of the cloud infrastructure is accessible to the user.

- Software as a Service (SaaS) – users interact with an application which is deployed on a cloud infrastructure, and they access it through various kinds of devices (for example a web browser or a mobile device). The application behaves as a monolithic unit, so the user is not aware of the deployment setup, nor the application design and implementation.
- Platform as a Service (PaaS) – users create an application, which then they can deploy on to the provided platform. They can manipulate the application (configure the application, update it and so on), but can not affect underlying infrastructure (operating system, storage and other configurations). The platform behaves as a monolithic unit.
- Infrastructure as a Service (IaaS) – users are provided with computing resources (processing units, storage, networks), which they can use for creating their custom infrastructure for deploying their application.

Deployment model defines who manages the Cloud infrastructure and by whom it is accessible.

- Private cloud – the infrastructure is completely managed by a single organisation for the organisation's purpose or usage to granted entities.
- Public cloud – the infrastructure is managed by a single organisation, but its service is accessible to the general public.

- Community cloud – the infrastructure is run by one or more organisations and is intended for a specific community which shares a similar concern.
- Hybrid cloud – the infrastructure is a combination of several distinct types of deployment infrastructure (private, public or community), but are connected for usage of the customer.

2.1.1 Scaling

Cloud Computing as described earlier is more focused on the infrastructure. The infrastructure can be used for a wide variety of tasks. Examples can be web hosting, database platform and more. One important use case is to process a large amount of data. The community started to use the term Big Data for this use case. The size of the Big Data can vary a lot, for example on Flicker, around 611 million pictures were uploaded during the year 2016 [22]. With an average picture size of 2 MB, that makes 3,3 TB of photos per day. Just to store such an amount of data the scalability of the infrastructure is critical. There are two main approaches to scalability in Cloud environments mentioned by *Vaquero et al.* [27] – scale vertically or scale horizontally.

- Vertical scaling – improving the current set up by scaling the machine's resources. For example by improving the power of the CPU or other resources.
- Horizontal scaling – improving the current set up by adding more machines into the cluster.

Vertical scaling has its limits because increasing the power of the machine is restricted by the power of its components. Moreover, adding highly powerful components is often very expensive, as it requires more specialised hardware than the standard commodity components. On the other hand, horizontal scaling can take advantage of using cheap commodity hardware, but it brings high demands on the software to manage the distributed environment.

2.1.2 Challenges of Distributed Environment

Distributed environment for computation brings several problems which need to be tackled by the software which runs on this environment. *Katal et al.* [19] surveyed the main difficulties and issues. They divided them into five categories – Privacy and Security, Data Access and Sharing of Information, Storage and Processing Issues, Analytical challenges, Skill Requirement and Technical challenges.

This Section will focus mostly on the Technical challenges.

Fault tolerance: Because of horizontal scaling the cluster contains a high number of machines, which means that the probability of error of a machine or some of its components increases significantly. Therefore, fault tolerance and recovery need to be taken into consideration when designing software running in such an environment.

Scalability: As many machines work on the same job, there is a need for coordination of the tasks. Also, the programs running the computation need to be created for the distributed environment. As the computation demand might vary, the platform

needs to be flexible about increasing or decreasing the number of workers running the execution.

2.2 Hadoop

In 2003 *Ghemawat et al.* from Google published work on Google File System (GFS) [17] and a year later *Dean et al.* also from Google published work about their distributed computation framework MapReduce [16]. These two papers inspired the open source community to create open source versions of these projects, and so the Apache Hadoop platform was created. It is a platform for distributed computation that tackles challenges mentioned in the previous Section. In its basic version it incorporates several modules:

- Hadoop Distribute File System (HDFS) [23] – storage module which creates a distributed file system and handles fault tolerance.
- Yet Another Resource Manager (YARN) [28] – resource manager which schedules the computation jobs in a cluster.
- MapReduce – a YARN-based system for distributed computation.

As the Hadoop platform was continuously developing, more projects were created compatible with Hadoop such as Apache Spark, Apache Hive [25] and more.

2.3 Directed Acyclic Graph (DAG)

The computation frameworks which will be described in the following Section employ directed acyclic graph (DAG) for defining the dataflow of the program's computation. This Section will define DAG and its characteristics. The definitions follow *K. Thulasiraman and M. N. S. Swamy* [24].

Definition 1. (Graph) Graph $G = (V, E)$, where V is a finite set of *vertices* and E is a finite set of *edges*. Each edge is defined by a pair of vertices.

Definition 2. (Directed graph) Graph $G = (V, E)$ is called *directed graph*, if edges are defined by ordered pairs of vertices.

Definition 3. (Walk) A *walk* in a graph $G = (V, E)$ is a finite sequence of vertices $v_0, v_1, v_2, ..., v_k$, where $(v_{i-1}, v_i), 1 \leq i \leq k$ is an edge in the graph G.

Definition 4. (Closed walk) A walk in a graph $G = (V, E)$ is called a *closed walk* if the starting and ending vertices are the same, otherwise the walk is called *open walk*.

Definition 5. (Cycle) There is a *cycle* in a graph $G = (V, E)$, if a closed walk exists inside the graph.

Definition 6. (Directed acyclic graph) Graph $G = (V, E)$ is called *directed acyclic graph*, if the graph is directed and does not contain any cycles.

2.3.1 Characteristics

One of the significant characteristics of DAG is that it has *topological ordering*. Conversely, if topological order exists in a directed graph, then it is a directed acyclic graph. This characteristic can be used for detecting a DAG, as there does not exist a topological order for a directed graph which contains cycles.

Definition 7. (Topological order) *Topological order* is a labeling of vertices of n-vertex directed acyclic graph G with integers from set $\{1, 2, ..., n\}$, where an edge (i, j) in G implies that $i < j$ and the edge is directed from vertex i to vertex j.

2.4 Frameworks Overview

This Section will list several frameworks for processing Big Data which utilise DAG in some way, describe how they employ it and explain the basic programming paradigms of the frameworks.

2.4.1 Spark

As researchers tried to improve upon MapReduce performance, they realised that there was one main issue – reuse of intermediate data (for example in iterative algorithms). To reuse intermediate data in a MapReduce job, the job needs to write the data into a storage system (for example HDFS) between each MapReduce cycle, which results in expensive I/O operations and slows down the execution.

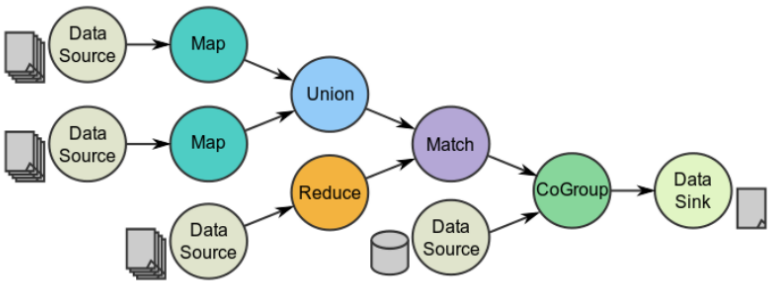

Figure 1: Example of Spark's DAG Dataflow.

Hence *Zaharia et al.* [29] proposed *resilient distributed datasets (RDDs)*, an in-memory, fault-tolerant, parallel data structure, which they implemented into a project call Spark (now under the Apache Foundation). As it is an in-memory data structure, it increases performance and eliminates the I/O bottleneck. When *Zaharia et al.* were solving fault tolerance for this data structure, they had to consider the specific characteristics of the in-memory approach. They could not use a replication approach, which was one of the common approaches, as it would add significant computation overhead and memory usage. Instead of that, they came up with programming model which defines transformations over a data where the data structure is immutable, so every transformation results in a new object. This shift enabled the creation of a

lineage of transformations, which can then be used for re-computation in case of lost data. An important fact is that, when a data loss occurs, Spark recomputes only the lost data.

Spark uses a DAG for defining the dataflow of the computation execution. An example of such a DAG dataflow can be seen on Figure 1. The source code defines through the Spark's API an operator DAG, which is then passed to the DAG Scheduler which performs a set of optimisations. It splits the operators into stages of tasks. A stage consists of tasks based on the partitions of the input data. The scheduler compresses as many tasks as possible into the single stage as all tasks of a stage are performed on single partitions of the data and do not need any exchange of data (shuffling). After dividing tasks into stages, they are passed to the Task Scheduler, which handles the planning of execution in cooperation with the cluster manager.

There are two types of functions in the Spark RDD API – transformations and actions. Transformations take as input an RDD and output also an RDD (for example `map`, filter). Actions take as input an RDD, but the output can be anything. The transformations behave in a lazy manner, and when the code's executor reaches an action, it evaluates all the previously defined transformations up to the action and then continues to the rest of the code. Figure 2 presents a basic list of Spark's functions. In addition to the RDDs API, Spark consists of several other modules which extend the basic RDDs behaviour:

- DataFrames/Dataset [15] – Declarative API, which enables the use of constructs similar to those of SQL (`where`, `groupBy` and so on), even using limited SQL itself.
- Structured Streaming [30] – API to build a streaming application (i.e. application where the flow of data is continuous).
- MLlib [21] – high-level API for using Machine Learning algorithms in a distributed environment.
- GraphX [18] – API for processing graph structures.

2.4.2 TensorFlow

TensorFlow [14] is a project of Google which was open sourced. It is designed for large-scale machine learning computation. One of its advantages is the range of devices which it can operate on, starting from smartphones (Android and iOS), single machine setup to distributed clusters. Moreover, it supports computation on both CPU and more importantly GPU, where computation parallelism is used in a very efficient manner.

The underlying computation is defined as a directed graph, where nodes are *operators* which modify *tensors* that flow along the normal edges in the graph. Tensors are multidimensional arrays that are passed from operation to operation. Operators can have zero or more inputs and zero or more outputs. There are several types of

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Figure 2: Example of Spark's API as presented in [29].

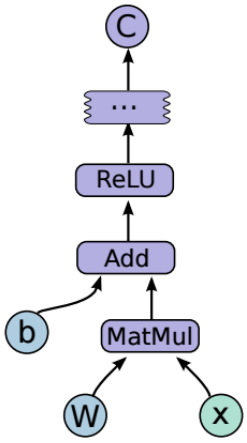

Figure 3: Example of TensorFlow's DAG as presented in [14].

operators and the basic overview can be seen Figure 4. Additionally there is also concept of *variables* that enables to mutate the variable (i.e., special tensor with reference) for example for model's parameters.

Compared to Spark's MLlib, TensorFlow is rather low-level. Instead of being constrained only to several implemented algorithms (as in MLlib), in TensorFlow you define the exact computation yourself. Although TensorFlow also has support for Moreover, in the case of TensorFlow, the underlying representation is not a DAG but just a directed graph as it supports looping.

2.4.3 Storm

Storm [26] is a real-time stream data processing system originally developed at Twitter (now under the Apache Foundation). Twitter developed it to perform real-time analysis of their data.

Storm uses a directed graph to define the dataflow and computation over the

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

Figure 4: An example of TensorFlow's API as presented in [14].

data. It defines two types of nodes – spouts and bolts. Spouts are input nodes, which load the from other systems. Bolts are processing nodes which transform the incoming data and pass the results to the next set of bolts. Similarly to TensorFlow, the representation is not a DAG but a directed graph, as Storm supports loops.

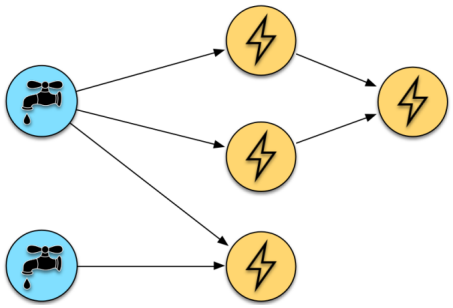

Figure 5: Example of Storm's DAG.

2.5 Related Work

There are several projects which in some way tackle a similar problem or have other similarities with the software created in this thesis. This Section will describe them.

2.5.1 Seahorse

Seahorse [3] is a graphical user interface for creating Spark jobs developed by the company Deepsense, which specialises on Big Data Science.

The editor focuses on high-level programming of Spark jobs as it offers predefined transformations, so the users do not have to write any code, just simple drag&drop nodes, connect them and specify their properties. This simplicity enables the creation of Spark jobs even for people not so proficient in programming, but it still preserves enough flexibility since anybody can define his or her own transformations in Python or R [3].

Aside from defining Spark jobs, Seahorse can execute the jobs in either local or cluster mode (YARN, Mesos, Standalone).

In the end, Seahorse mainly focuses on data science jobs, and for that they adapted the whole user interface and range of features. The main look of the editor can be seen on Figure 6.

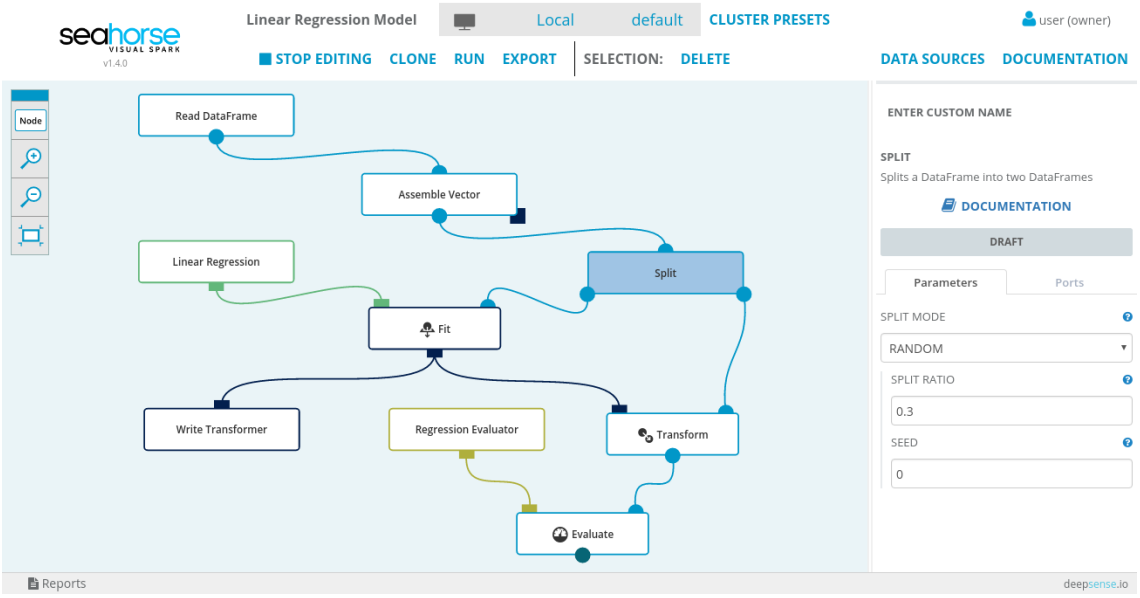

Figure 6: The interface of Seahorse editor [3].

2.5.2 Spark Web Interfaces

In the Spark 1.4 release, the Spark's developers added DAG visualisation to Spark Web Interfaces. When a user submits a Spark job to the cluster, it has its Web interfaces, where the user can monitor the status of the job. To make it easier to debug Spark jobs, the developers added the Execution DAG visualisation, which shows how the code of the job defines the underlying DAG that is used for the computation. It is purely a visualisation tool and does not offer any interactivity. An example of the visualisation can be seen on Figure 7

2.5.3 Dataiku

Whereas Seahorse is a specialised tool for creating Spark jobs, Dataiku [2] is more of a data science Swiss Army knife. It is a collaborative platform for data science, integrating a wide variety of tools: data connectors (HDFS, No-SQL, SQL...), machine learning (Scikit-Learn, MLlib, XGboost), data visualisations, data mining, data workflow and more. All these features are integrated into an easy to use environment, where many of the definitions can be done by "code or click". Moreover, Dataiku created the whole platform with cooperation in mind so the entire team can work in one environment.

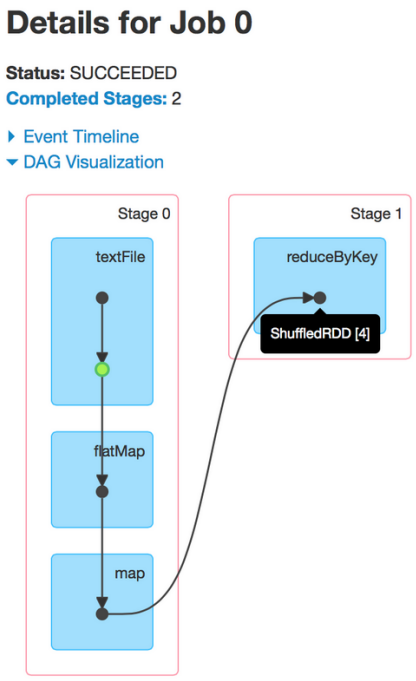

Figure 7: Example of Spark's DAG visualization.

3 Design

This Section will cover the high level details of the software which is developed as part of the thesis. It will describe the basic goals of the software (Section 3.1), the used Adapter design pattern (Section 3.3), graph validation (Section 3.4), code generation (Section 3.5), code execution (Section 3.6) and lastly code parsing (Section 3.7). The software is called **daGui** and its GitHub repository can be found on `https://github.com/AuHau/daGui`.

3.1 Overview

daGui is an integrated development environment (IDE) like software, which is meant to support easy development of programs which are based on frameworks that use directed graphs for program representation. It is a general tool, which provides an extensible platform for working with these frameworks.

Figure 8: daGui's logo.

To have an idea of what daGui does and how it does it, see a mock-up of its basic interface on Figure 9. Users drag&drop nodes from the node's pallet. Then they connect the nodes with directed links, to form a dataflow. After that, they fill the parameters of the nodes (for example the filtering function for the `filter` node in Spark) and if the graph is valid, the code is generated and can be executed locally or on a cluster, based on given settings.

3.1.1 Use-cases and Users

When designing and developing software, it is important to know its purpose and its users. daGui most probably will not be utilised by experienced developers as a primary IDE, because it is more efficient to write the code directly than to drag&drop nodes, link them and fill their properties. Still, there are several valid use-cases for such software.

One valid use-case for daGui is related to teaching these technologies. For students, it might be hard to understand the underlying principles of the technologies, so the graph graphical representation can be very helpful. This use-case assumes users who might not be so skilled in programming or with computer interaction. On the other hand, users will most probably not be complete beginners in computer science either, as the field of Big Data is already a specific subset of computer science, so some level of programming knowledge is assumed.

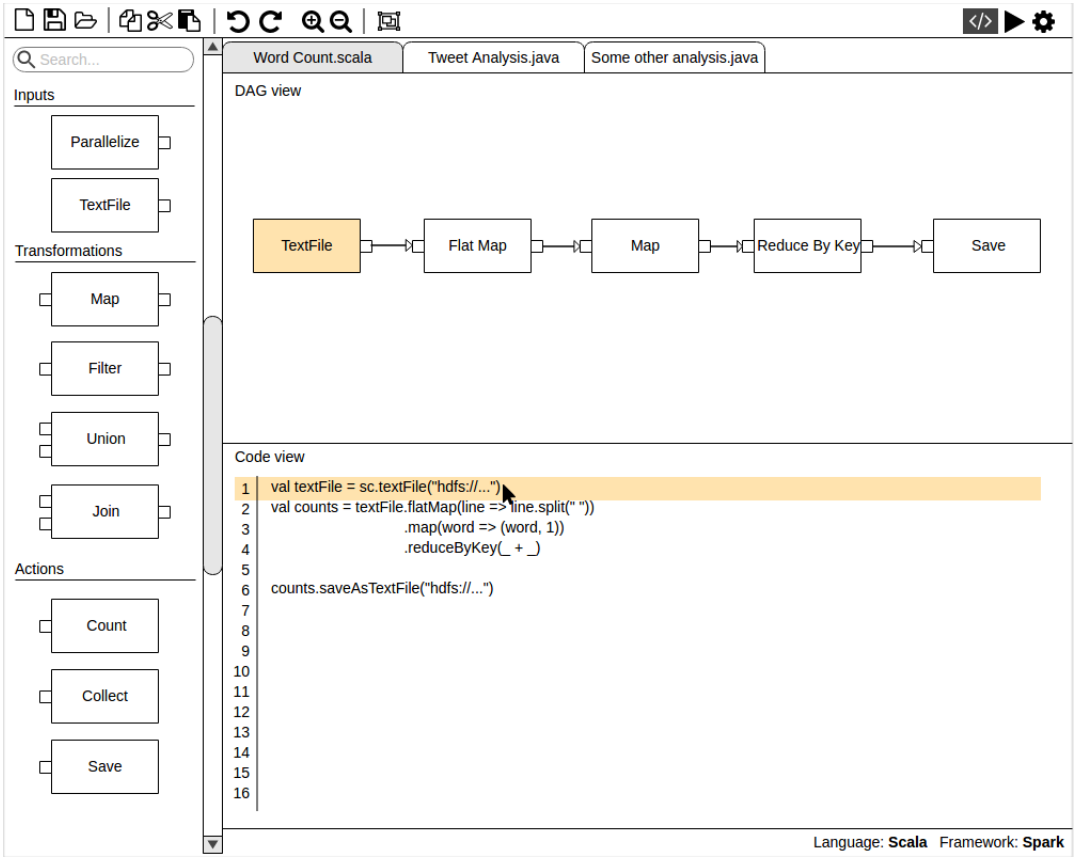

Figure 9: Mock-up of daGui interface with Spark adapter.

Another use-case is the presentation of the programs. Explaining what some piece of code does can be sometimes a bit challenging. With the graph representation of the code, this task can become much easier.

The last use-case is connected to prototyping. Some tasks require more thinking about a problem and playing with the code. An example can be developing machine learning programs in TensorFlow. This type of tasks does not need huge efficiency, but rather an overview of the problem, so new ideas on how to solve a particular challenge can be developed.

3.1.2 Goals

Before starting to work on daGui, there were several goals defined which the program should fulfil.

As it is mainly a graphical user interface program, with high interactivity, the *User Experience (UX)* of the program is critical. It needs to be easy to control, with very natural control flow. Particularly, since it incorporates a graph editor, the interactivity is higher than in a typical IDE. This goal also correlates with the beginner users group identified in Section 3.1.1.

When it comes to the main features set of daGui, there were set three main goals — *code generation, code execution and code parsing*. Code generation (translation of

DAG into runnable code) is the main purpose as it lays at the core of the whole concept. Code execution was derived from the UX goal as it introduces a very convenient way of working with the software. Moreover, typical IDEs provide ways to run and debug code easily. Lastly code parsing is a logical step as it would introduce more flexibility of usage of the software, because it would enable editing source code files which were not created with daGui.

As there are many libraries, frameworks and tools which utilise DAG in some way, daGui aims to be a general platform, which can be easily extended with support for any of these frameworks in the future.

3.2 Graph and its Components

This Section will define and describe the graph and its parts which users create in daGui. On a general level, it is a directed graph with nodes and directed links (edges). It is up to the adapter's authors to give the nodes and links some specific meaning.

Every node has a label, which should express the function of the node. Moreover, it can have an editable field which is placed outside of the node. The adapter's author can utilise that, but is not required to do so. For example, Spark's adapter uses it for naming variables in the generated code.

Node has ports which define the input and output degrees of the node. Ports can be of two types: input ports and output ports. The ports are visualised as small dots on the node with a different colour for each type. The links between nodes are created between ports. daGui restricts the input ports, where one input port can only accept one link, but the output ports are not restricted, so there can be an unlimited number of links going from an output port (hence every node needs only one or zero output ports). This configuration currently meets all requirements of the Spark's adapter, but in the future these settings may be generalised and it may be possible to set these constraints within the adapter's configuration.

In graph validation, the term *input nodes* is used. The adapter's authors define the input nodes. Often input nodes are those nodes which have zero input ports (zero input degree), but it does not always have to be the case.

Figure 10 shows an example of nodes, ports and links.

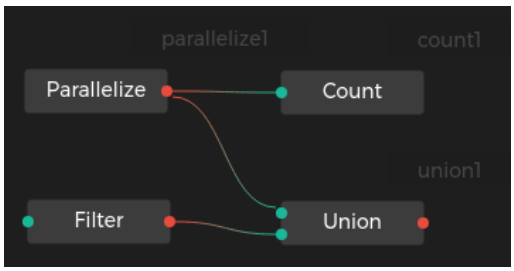

Figure 10: Example of nodes, ports (input ports are green and output ports are red), links and editable fields (grey text outside of the nodes).

3.3 Adapters

To fulfil the extensibility goal of daGui, its architecture needed to be built with this goal in mind from the beginning of its development. daGui uses the *Adapter* design pattern to define a clear interface between the daGui's core, which handles the GUI of the application, and parts which define the framework specifics areas. In this way, every task which is somehow related to the framework is delegated to the frameworks adapter, and daGui's core only processes the results passed back from the adapter. An example of such a delegation can be code generation, where daGui's core passes the user's graph to the adapter and then only presents the adapter's output, which is the generated source code that represents the graph together with some metadata.

Information which defines an adapter:

- Framework's/library's/tool's name.
- Supported programming languages and their versions.
- Supported versions of the framework/library/tool.
- Node templates – definitions of supported nodes.
- Node template grouping – it is possible to group the nodes by their functionality, for a better overview.
- Graph validation – the validation of the graph is not delegated to the adapter. Instead, the adapter defines the criteria, which the graph needs to fulfil so that the adapter can generate valid code. More details in Section 3.4.

Node template defines a type of node in the graph, which is usually translated into a function call during source code generation. The template defines the properties of the node, such as visual look in the graph canvas, the node's type name and label, input and output ports, parameters of the function into which it will be translated and several other details.

Tasks which are delegated to the adapter:

- *Code generation* – the task translates the given graph into runnable source code. More details in Section 3.5.
- *Code execution* – the task takes the generated code from the *Code generation* task and the user's configuration which specifies the parameters of the execution and launches it. More details in Section 3.6
- *Code parsing* – the task takes a source code file and produces a graph representation which is then displayed to the user. More details in Section 3.7.

There are several other tasks which both adapter and node templates perform, but those are mainly related to the implementation side of the software and are detailed in Section 5.

3.4 Graph Validation

To be able to generate code from the graph, the program needs to verify that the graph is valid according to the adapter's definition. As mentioned in Section 3.3, the framework's adapter does not perform the validation itself, it only defines the criteria which the graph needs to fulfil, and daGui core then evaluates them.

The currently implemented criteria are:

- Has Inputs nodes – the adapter defines what node templates are Input nodes and then checks that there is at least one Input node present in the graph.
- Has all ports connected – check that all ports in all nodes inside of the graph are linked with some other port.
- Has all required parameters filled in – check that all parameters of the graph's nodes which are required are filled.
- No cycles are present in the graph.

The cycle detection uses the DAG property which states that every DAG has a topological ordering, as referred in Section 2.3.1. daGui implements a topological sorting algorithm for cycle detection, which works well, but this algorithm does not convey any information about the location of the cycle, only about its presence. For topological sorting, daGui uses an implementation from a JavaScript library written by Saumel Neff called `topsort` [12].

A future improvement will be to implement an algorithm for searching Strongly connected components, which identifies exactly the cycle inside the graph to better convey the error information to the user.

```
def validateGraph(graph, checks):
    inputs = []

    for node in graph:
        if checks.hasConnectedPorts and not
           checkAllPortsConnected(node):
            addError()

        if checks.hasRequiredParamsFilled and not
           checkAllRequiredParamsFilled(node):
            addError()

        if isNodeInput(node):
            inputs.append(node)


    if checks.hasInputNodes and inputs.isEmpty():
```

```
        addError()

    if checks.noCycles and graphContainsCycles(graph):
        addError()
```

Listing 1: Pseudocode of validation of the graph.

3.5 Code Generation

Code generation (i.e., translation of a graph into the runnable source code) is the core feature of daGui. The task can vary significantly between frameworks, which is why it is delegated to the framework's adapter and not implemented in the daGui core. For details about the referential implementation see Section 5.2.2.

3.6 Code Execution

Code execution is another task which is delegated to the framework's adapter, because each adapter can use different dependencies, various process calls and so on.

The execution flow is split into two stages:

- Build – a compilation of the generated source code and linking required libraries.
- Run – executes the computation with specified configurations.

Not all stages have to be used by the authors of adapters as scripting languages such as Python do not require the build stage.

The Run stage usually needs some parameters for the execution itself. For example in Spark, these parameters specify where the job should be launched (local mode, cluster mode, YARN mode and others), how many resources should be allocated for the job, what libraries should be linked with the program and so on. All these parameters need to be able to be set. Otherwise, it will limit the users of daGui. Moreover, from the user experience point of view, it would be convenient if the user could easily switch between sets of parameters, so the user could try something in local mode to validate that the code runs as expected on a limited range of data and then launch it on a cluster with the full data range. daGui has a solution, which is inspired by other IDE software, that is called *Execution configurations*. The user can set up an unlimited number of Execution configurations, each with its set of parameters, and then the user can easily switch between them.

3.7 Code Parsing

Code parsing is the last main feature which was defined to be achieved. Its importance is in the fact that it will enable importing any source file into daGui and therefore it will remove the restriction that only files originating from daGui are compatible with daGui. As this task is again adapter and language specific, it will be delegated to the adapter. When importing the file, there is no information about it, so there will

be an import dialogue where daGui will ask the user about which framework is used in the file, which version of the framework is targeted, and which language version is used. This information is then used for calling the proper adapter's parsing function.

At the beginning of the work on this feature, we realised that it would not be any easy task. There were two possible solutions to this task.

1. Use the framework to generate the graph.
2. Directly parse the code to generate the graph.

The first approach uses the actual framework. It launches the source code with some dummy data on localhost, and as the framework builds the graph for the execution, the graph is saved in daGui and used as the source code representation. This approach has one significant advantage that there is no need of parsing the code in daGui as the framework takes care of that [1]. However, also it has many disadvantages. First of all the generated graph might not fully represent the code in the file. When developers use some dynamic constructs (conditions, looping), then these constructs can change the shape of the graph based on the input data. Therefore the extracted graph can represent only one branch of possible walkthroughs of the source code. Another related problem is deciding what data should be used for the execution. The simple solution is to ask the user as he should have knowledge of the code and therefore should know what data it will need, but this might not always be the case as users might want to explore some unknown source code in daGui. Lastly, this is not very user-friendly as the import process would require the user to provide the dummy data.

The second approach consists of parsing the code directly by daGui (or more accurately by the framework's adapter). The problem with this method is that daGui would have to have support for control flow as the graph will need to be able to express branching situations for conditions in the code, cycle support for looping and all the other language's features. Parsing of the code would consist of building an Abstract Syntax Tree, which represents the structure of code and then analyses the tree to deduce the graph which represents the code. Another issue relates to tools used for the parsing. It is not a trivial task to write a library for building an AST. There are tools for working with AST for a specific language usually written in that language. As daGui supports a broad range of languages, parsing all of them might be very challenging. One possible solution to this problem is to call some external dependency for retrieving the AST and then work with it inside daGui. However, the need for an external dependency brings an extra burden as the dependency might not always be satisfied on the user's system, which can introduce user experience problems with requests to satisfy such a dependencies. We did a basic search for tools written in JavaScript for parsing AST of other languages, and we found several of them, but further research will be needed to compare their functionality and reliability. Lastly, the biggest problem of directly parsing the code is the complexity of the task itself. An example of how the control flow could be expressed in the graph is in Figure 11.

[1] The framework does not parse the code but based on the API calls, it builds the graph.

After doing the research about this feature, we decided that implementation of this feature would be highly complex and the result unsure, as creating a general parser which would process any written code would be very time-consuming. Instead, we decided to put the focus on the previously listed features to ensure that we will deliver reliable and stable software. However, in the future this feature could highly improve daGui's capabilities. Therefore it will be one of the main points of the future work.

```
from pyspark import SparkConf, SparkContext

conf = SparkConf()
sc = SparkContext('local', 'text', conf=conf)

textFile = sc.textFile(...).filter(...).cache()
count = textFile.count()

if count < 10:
    temp = textFile.groupBy(...)
    for i in range(count):
        temp = temp.map(...)

    temp.saveAsText(...)
else:
    textFile.sort(...).saveAsText(...)
```

Listing 2: Example code which could be parsed.

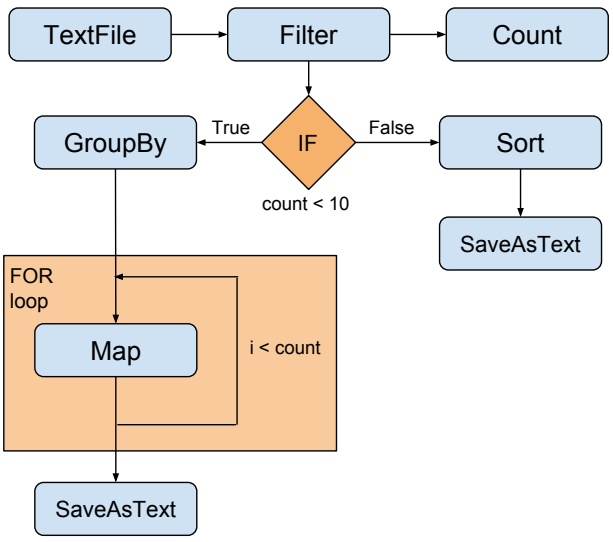

Figure 11: Example of how the code presented in Listing 2 could be parsed and what the graph could look like with the control flow.

4 Implementation

This Section will describe the low-level details of daGui.

4.1 Technologies

During a survey of technologies for daGui, there was one important factor considered: the portability of the software. The core technology has to be platform independent to reach as many users as possible with as little effort as possible. Also, in the beginning of daGui's development, the authors of Hops Hadoop distribution [7] approached the thesis author, requesting that daGui be integrated into their environment. This introduced another requirement of simplicity of porting daGui into a Web environment.

The result of the survey were two possible solutions:

- Packaged Web application with a local server;
- Electron standalone application.

The packaged Web application would consist of a local server written in Python, which would be the back-end of the application and would serve the interactive Web application over HTTP protocol to the user's browser. The Web browser would be the main entry point for the user. The advantage of this approach would mainly be straightforward access to the user's OS and utilising well-known principles of Web development. The big disadvantage would be a distribution of such software as packaging and distributing is possible, but rather hard and inconvenient from a user's point of view.

Electron [4] on the other hand is an entirely stand-alone program. It is essentially a packed Chromium Web browser with Node.JS as the application back-end and a V8 JavaScript engine. Therefore, writing an application with Electron is almost the same as writing a Front-end JavaScript Web application. The main differences with Electron are the additional JavaScript APIs for accessing the underlying OS resources and the application GUI management (e.g., opening windows, dialogues). The advantage of this approach is that it provides a much better user experience, as the application behaves as a monolithic unit. Moreover, as Electron development is almost identical to Web development, it will be simple to convert daGui into a proper Web application in the future. The disadvantage of Electron is the size of distributed program, as it contains a standalone Web browser which adds up to hundreds of megabytes to the final package.

After comparing these two approaches, the chosen one was the Electron solution, for its better user experience and also the fact that nowadays the size of programs is not a big problem, as high-speed internet is becoming standard and most users also have big memory storage.

The next step was to decide which tools, libraries and framework to use for the front-end development. In the end, React + Redux were the main libraries used, in addition to other small tools of which only some will be described here.

React [10] is a rendering library which holds a Virtual Document-Object-Model (DOM) representation and through that tries to minimise the changes in the actual browser's DOM as they are rather expensive. Through React the developers create Components which define some element on the Web page with its full life-cycle. This architecture is highly useful for daGui, as the rendering of some adapter's specific parts can be delegated to the adapter's authors (for example Run Configuration form), where a result of a call to the adapter's function can be a Component which will be rendered through React.

Redux [11] is built on the idea of Facebook's Flux [6] and functional programming. It is a tool which keeps a synchronised state of the whole application. When there is a change in the state of an application, Redux emits a new state with the changes incorporated in it. This design is very useful, as it is very easy to implement history (undo/redo) in the application, because Redux's state is immutable. Therefore the application can easily keep track of the previously states and roll back or forward at the user's request. As in JavaScript objects are generally mutable, another tool which was used was Immutable.JS [8], which has a special API which enforces immutability on its special objects.

The last valuable library was JointJS [9]. daGui needs rich support for diagramming because users will need to create and manipulate the graph. There are several JavaScript diagramming libraries. After comparing their feature sets and especially their licensing, the chosen one was JointJS. It is a high-level library for creating interactive diagrams, with rich event support and easy customization.

To build the whole environment into an executable program with all the previously mentioned libraries, there is a tool which serves as "glue", called Webpack [13]. It is a handy tool which optimises the building process and, in particular, it supports Hot Module Reload. It replaces the changed components directly in the Website, which means that the developer does not have to refresh the whole program (or Website) and the changes propagate immediately.

As setting up all these technologies together takes much time, there are many boilerplate projects for a different combination of technologies. These projects have the basic environment with all technologies already set up and are ready for the developer to start to work with right away. Electron React boilerplate [5] was chosen for daGui as it incorporated all the technologies mentioned earlier. There were several features which were not needed and were therefore removed, such as the React Router. There are some other features which are not actively used in daGui but remain in the project, as they might prove handy in future development. The main feature is support for Flow (static type check for JavaScript) and ESLint (linter for JavaScript, a tool which enforces consistency of the format of the source code).

4.2 Other Features

In addition to the main features which were set in Section 3.1.2, there are several smaller features included in daGui, which this Section will describe.

4.2.1 Nodes Highlighting

Nodes highlighting is a feature which helps in orientation inside of the graph and the generated code. When a user hovers over a graph's node, it highlights the proper part of the code which the node represents. The highlighting also works in the other direction: when a user hovers over a piece of code, the appropriate node is highlighted.

The highlighting is possible because of a special class called *CodeBuilder*. During the code generation part, this class is used for storing the generated code. Its crucial feature is that it internally notes which parts of the code are linked to which node's ID. This information is then used in CodeView together with the Ace editor to create so-called Markers for the Ace editor. They are used for handling the hover action over the code and also to highlight the proper part of the code when needed.

4.2.2 Image Export

The last feature is handy for the presentation of a program. daGui can export the graph as a PNG image. It is easier than taking screenshots as it automatically renders the whole graph and not just the visible part.

4.3 User Interface

As one of the set goals was to have a good UX, the user interface is a critical part of daGui. Moreover, software nowadays also needs to look nice to have good feedback from the users. To make daGui visually appealing Petr Sykora, a graphic designer, helped with the visual design of the editor. He created a dark styled theme and also the logo and icon for daGui. The main daGui window can be seen on Figure 12. Aside from the main editor view, daGui also has modal windows. An example of such a window can be seen on Figure 13.

As an important user target group of daGui are beginner users who might be confused about the parameters they are supposed to configure, daGui tries to help them as much as possible. In several places of daGui, there are icons which on hover display a help tooltip. Some input fields also display a similar tooltip when hovered over. An example of such a help tooltip can be seen on Figure 14 and Figure 13. Moreover, when some error happens in daGui, the program tries to assist the user as best it can with the solution of the error. An example of that is the reporting of the validation errors which can be seen on Figure 15.

4.4 Platform Adapter

As daGui will be ported into a Web environment in the future, the daGui's architecture has to be prepared for this transition already from the beginning of its development. daGui needs a back-end for several tasks: saving and opening files, compiling and launching the execution of files and some other small tasks. These tasks are environment specific as in Electron they will be implemented directly using

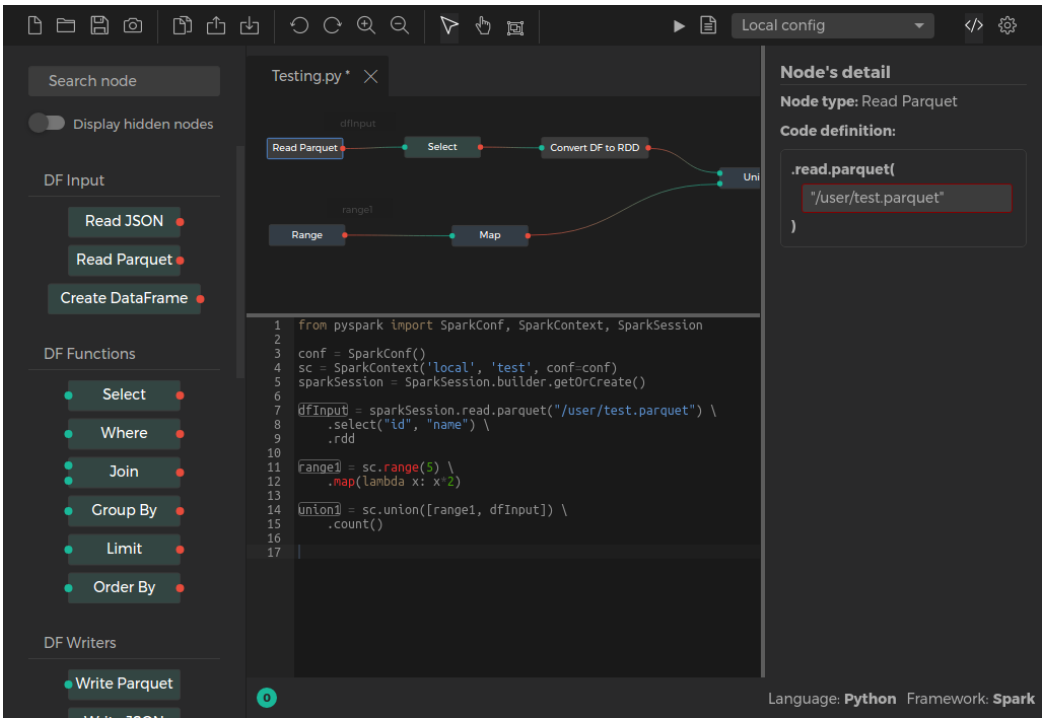

Figure 12: Look of daGui editor.

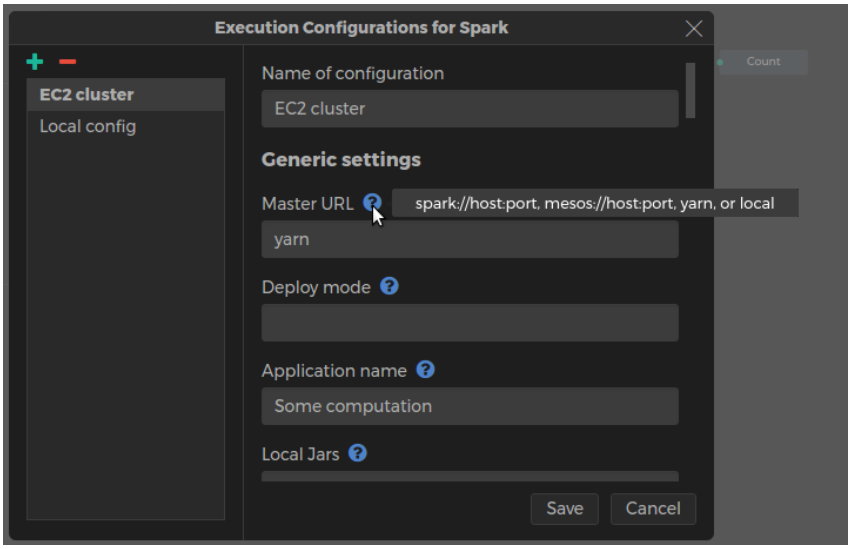

Figure 13: Execution Configuration modal window with displayed help for configuration parameter.

NodeJs, but in the Web environment, they will most likely be delegated to a remote server using an AJAX call.

There is a special adapter called Platform adapter to shield daGui from the back-end's implementation specifics. This adapter is not related to the framework's adapters. Figure 16 shows the role of the Platform adapter in daGui's architecture.

The tasks of the Platform adapter are:

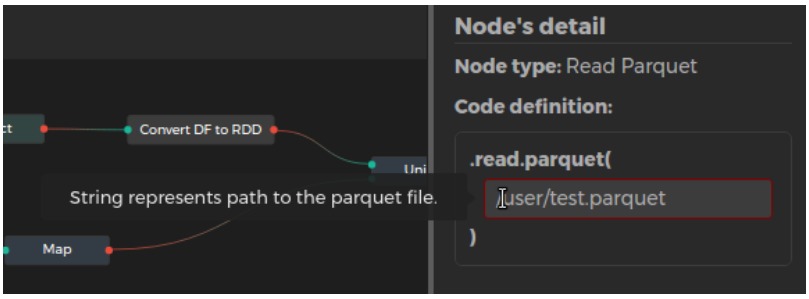

Figure 14: Detail of a node with displayed help for its parameter.

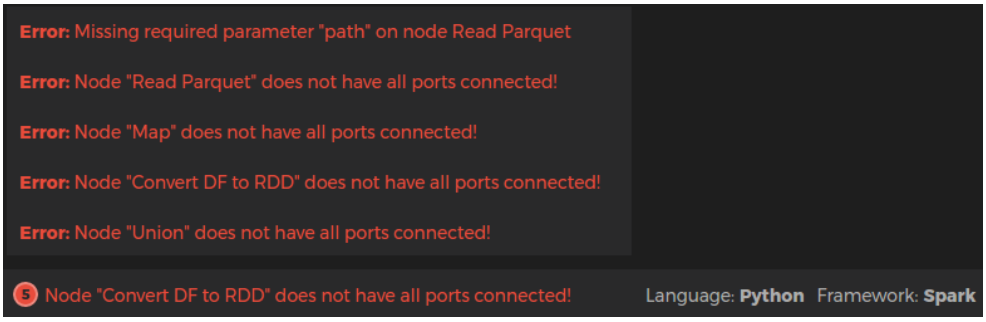

Figure 15: Errors View which informs the user about graph's error.

- Open source files – only those source files which were generated by daGui can be opened.
- Save source files – saves generated source code into the proper source file on the memory storage.
- Launch execution – calls the appropriate *AdapterExecutor* on the backend which handles the whole execution.

4.4.1 Persisting daGui's Files

As the Code parsing feature turned out to be too challenging (as described in Section 3.7) and daGui does not currently support it, there had to be another way to save and load the work. In the end, the work is saved into a proper source file, based on the currently used language. This source file contains the generated source code of the build DAG, and at the end of the file it includes serialised daGui specific meta-data about the work. This serialised meta-data contains:

- Version of daGui which generated the file.
- Hash of the whole file.
- Name of the used adapter and the framework's version.
- Name of the used language and the language's version.

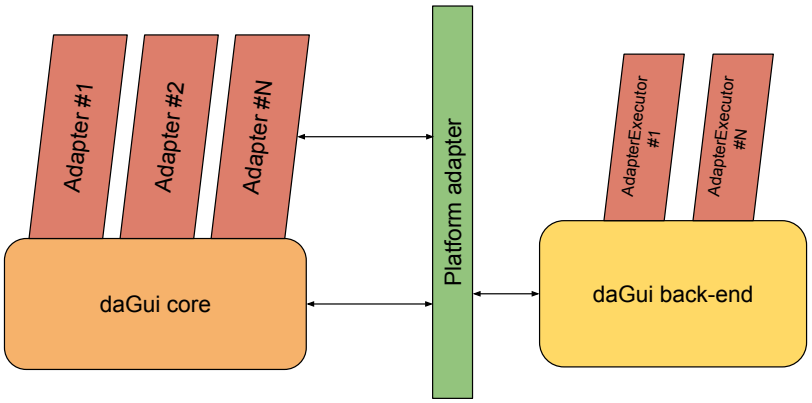

Figure 16: Overview of the architecture of daGui.

- Serialised JointJS object of the built graph.

From this meta-data, the daGui can completely reconstruct the original work. As parsing of the source code is not supported, there is a control mechanism which detects if anybody changed anything inside of the source code. When saving the work, daGui generates a hash of the source code which is then stored along with other daGui's metadata at the end of the source code file. If during the loading of the file any difference is detected, daGui raises a warning to the user, informing that the original DAG and the source code may not match and that loading and subsequently saving it may overwrite any changes in the source code.

Also, when daGui saves the work it regenerates the source code. If there are any validation errors and it is not able to generate the source code, daGui gives the user the possibility that only the meta-data be saved into the file and that the old source code in the file is preserved.

4.5 Components

As mentioned already in Section 4.1, the React library defines Components which can then be used in other Components. This Section will lay out an overview of the main Components which were created for daGui, and it will detail the most important ones.

In addition to Components in React, the concept of Containers is also often used. A container is essentially a Component which introduces some hierarchy into the Components layout. Containers often correlate with different layouts and pages in the application. As daGui is mainly a one-page application, since the editor is always visible and the modal window is used for all other parts (settings, new file dialogue and others), there is only one main container called *App*. This Container encapsulates all other Components and facilitates some interaction which does not need to be incorporated in Redux's state. The container and its functionality will be detailed later.

The overview of the key components can be seen on Figure 17.

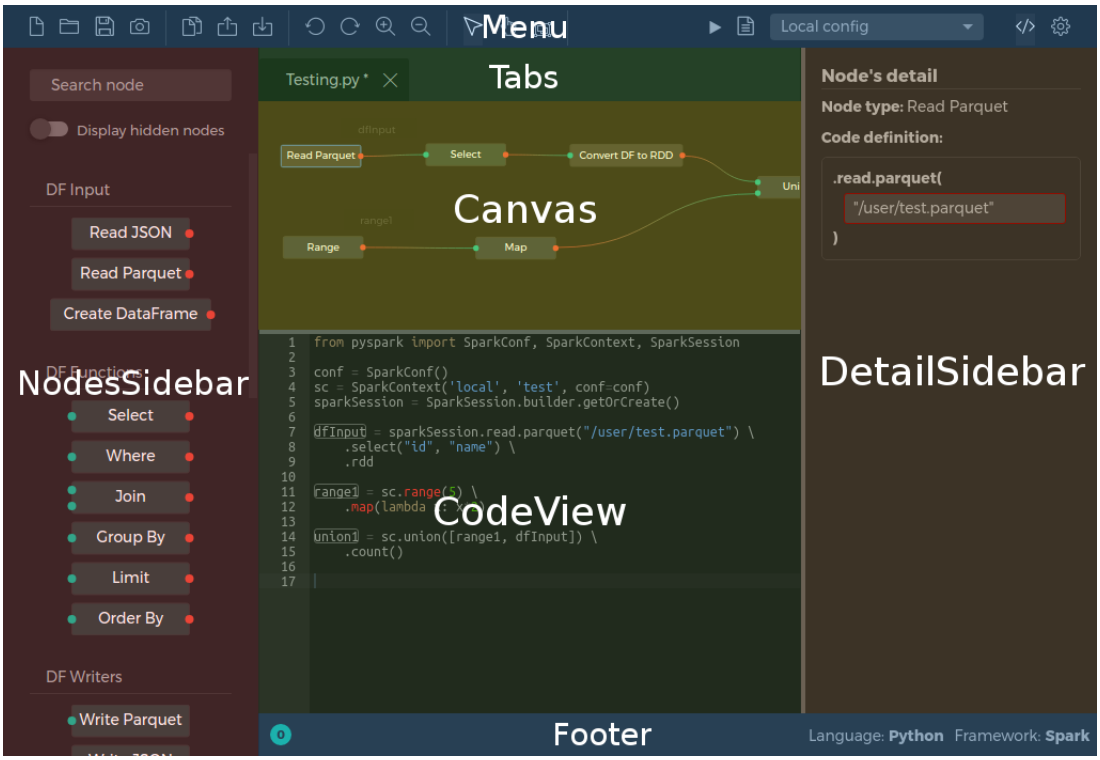

Figure 17: Overview of the main components in daGui.

- *Menu* – control component where all the control icons are placed. Also, it is the main place where the keyboard shortcuts are defined and handled (i.e., fired the proper Redux's action).
- *NodesSidebar* – a component which lists all possible nodes of the adapter of the current file. It features search of the nodes and also hiding/displaying less used nodes. Internally it uses a component called NodesGroup.
- *Tabs* – a component which enables the opening of several files at once and switching between them.
- *Canvas* – the most complex component of daGui. It manages the whole graph drawing and all related functions. It is more detailed in the following subsections.
- *DetailSidebar* – component that displays details of the selected node. The details mainly consist of parameters which are used for the code generation step.
- *CodeView* – an important component which displays the generated code and offers several interactive features. It is more detailed in the following subsections.
- *Footer* – component which shows status information. It displays used language and framework of the active file. Moreover, when there are some errors, it has a sub-component which display them to the user.

- *Modals* – component which is by default hidden. It encapsulates components which display modal windows for different dialogues (new file dialogue, settings, execution configurations and more).

4.5.1 App Container

The App container is the only React container in daGui. It mounts all the other components and therefore creates the layout of the application.

For optimisation reasons, it is good to have as few Redux's connected components (i.e., components that can directly access the Redux's state and fire Redux's actions) as possible. One way to achieve that is to have just a few connected components which distribute the proper callbacks into their sub-components. The App container is the main connected component, where the many callbacks are created and passed to the proper sub-components.

Moreover, this container facilitates some level of interactivity through its state. It manages actions which do not need to be incorporated in Redux's state. The main events the container handles are linked to the highlighting of nodes/code blocks. It distributes the highlighting callbacks to CodeView and Canvas components and based on the information passed through the callbacks it keeps an overview of which nodes should be highlighted and in which component.

4.5.2 Canvas Component

The Canvas component is the most complex component in daGui. It is based on the JointJS [9] library, but since the library only has support for very basic features, many of the features had to be implemented from the bottom up. At the beginning of the development the length and the complexity of the component started to grow very fast, so at one point, when the code of the component began to be impossible to manage, there was a need for better architecture. It resulted in creating *Canvas components*, which are components that are not connected to React in any way. Instead, they manage some part of the Canvas's functionality. It is not the perfect solution as the components have shared state (the Canvas component's state) and therefore there can be error states when several Canvas components try to modify some part of the shared state, which can cause "deadlock" [2]. On the other hand, this architecture helped with the readability of the code and separation of concern, which was the main motivation behind it. Until now there were no major issues with the current solution, but if some problems appear, a better solution will be created.

The list of current Canvas components:

- *Grid* – servers for drawing a grid on the canvas's background.
- *PanAndZoom* – implements panning and zooming support for the canvas.
- *Link* – handles any linking related events: link's creation, link's modification, link's validation and link's deletion.

[2] JavaScript is single-threaded, so the meaning of deadlock is not meant in the multi-threading sense, but rather as an error state after unexpected modification.

- *Nodes* – handles any node's related events: node's movement and node's deletion.
- *Highlights* – servers for highlighting nodes which were passed through Canvas's components properties from the App container.
- *Variables* – handles changes of node's variable name.
- *Selecting* – implements multiple selection of nodes: adding and removing nodes from the selection.

4.5.3 Modals Component

Even though daGui is a single-page based application, there are still some cases which need a slightly different layout (e.g., settings, new file dialogue). daGui follows the example of other IDEs, which use modal windows for this task. However, daGui has a somewhat different implementation. The usual way is to open a new system window and display the content in it. The modal window is separated from the main window. As daGui will be ported to a Web environment in the future, the system windows are not used for the task and instead daGui displays them as an overlay in the main window.

Currently, there are three types of modal windows: new file dialogue, execution configurations and settings view.

4.5.4 CodeView Component

The last critical component is CodeView. It displays the generated code and offers a little degree of interactivity. The component employs Ace Editor [1] for highlighting the code's syntax. Additionally, it implements node's highlighting and it is also possible to rename the variables inside the CodeView.

5 Adapters

5.1 Implementing an Adapter

Implementing a custom adapter is a relatively simple process, but it requires a bit of programming. First of all the author needs to have a good overview of the framework/tool/library which he will write the support for. He needs to understand how the framework relates to the graph, how it uses it and how the framework's API is related to the graph.

To integrate a new adapter with daGui, the author has to implement the adapter's class which extends the `BaseAdapter` class and register it in the daGui's configuration `/app/config/index.js`. Then, daGui will include the new adapter in its selection. If the adapter is supposed to support code execution as well, then the author also has to implement the adapter's execution class which has to extend the `BaseAdapterExecutor` class, and register it in the Electron's configuration `/app/config/electron.js`.

This Section will guide the adapter's author through the implementation. The recommendation is to have a look at the referential implementation of Spark's adapter, as much of the code can be reused in the new adapter. How much depends on the differences between the two frameworks.

5.1.1 Implementing the Adapter's Class

The adapter's class consists of static methods that are called by daGui to retrieve information about the adapter or to delegate some tasks to it. The `BaseAdapter` class serves as an interface definition, and all its methods have to be overridden in the adapter's class, otherwise daGui will raise errors.

The first part relates to the presentation of the adapter. The `getId()` method should return some short unique identifier string. The string should contain only alphanumerical values without spaces and should be reasonably short. This ID is used for example in meta-data which are stored with the graph while saving the work. Method `getName()` on the other hand can return any string with a reasonable length, that will be displayed to the users for example in the footer of the editor. Lastly, the method `getIcon()` should return the path to an image which will be displayed as the representation of the adapter. This should normally be a logo of the framework. This is not mandatory and `getIcon()` can return `null`, in which a case daGui will display the adaptor's name instead.

The next part is associated with supported versions of the framework and its languages. Method `getSupportedVersions()` has to return an array of strings which represent the versions of the framework supported by the adapter. The version is then always given for all other adapter's tasks such as code generation, execution and so on. Next is the method `getSupportedLanguages(adapterVersion)`, which has to return an array of supported programming languages for a given adapter's version. The languages in the array should be imported classes from `/app/core/languages/`. If there is a language missing, then the author can create a new language in the

earlier specified folder by implementing a class which extends the `BaseLanguage` class and then register it in daGui's configuration. However, daGui has already support for the major languages, so this should not be needed. The last method in this area is method `getSupportedLanguageVersions(langId, adaptersVersion)`, which has to return an array of strings which represent the supported language versions for a given language and adapter's version.

The following part is connected to the graph's nodes. daGui uses the term *Node Template*, as it represents only a template and not the node directly [3]. Method `getNodeTemplates(adaptersVersion)` has to return an object which contains all supported node templates for a given adapter's version. The keys of the object are the node template's types, and the values are the classes of the node templates. The next method is `getGroupedNodeTemplates(adaptersVersion)` which enables grouping node templates into groups which represent some similar function of the node templates. It should return an array of objects, which represents the group, but this method can also return `null` if the author does not want to use this feature.

Another part is linked to graph validation. As described in Section 3.4 the adapter only defines the criteria of the validation and daGui then performs the validation. Method `getValidationCriteria(adaptersVersion)` defines the criteria which have to return an array of criteria. The enum `ValidationCriteria` defines all possible criteria. If the author decides to use the "has input nodes" criterion, then he must also implement method `isTypeInput(type, adapterVersion)`, which specifies if for a given adapter's version is a node template's type an input node. The method returns boolean.

The second to last part relates to the adapter's components. daGui currently needs two components from the adapter — `ExecutionConfigurationForm` and `SettingsForm`. They will be detailed in the following Section 5.1.3.

The last part is connected to the adapter's tasks. daGui delegates two tasks to the adapter — code generation and code execution. The Section 5.1.4 will describe these tasks. The last method, connected with code execution, is `hasExecutionSupport()`, which returns a boolean which specifies whether the adapter supports code execution and has all necessary support implemented.

5.1.2 Node Templates

The Node Template contains all the information about what the node looks like in the graph, what parameters it has and more. Node Template is a class which extends `NodeTemplate` class, which defines an interface for the Node Templates. It has the following methods which the author has to implement.

- `getType()` – Returns a string that uniquely identifies the node across all daGui's adapters. It is advised to use adapter's name as prefix to ensure cross-adapters uniqueness.
- `getName()` – Returns a string which represents the node template name and which is displayed as label of the node in Canvas.

[3] It is a similar concept to Objected Oriented Programming: a class versus an instance

- `getModel()` – Returns a JointJs model which represents the look of the node in Canvas. More details will follow.
- `getWidth()` – Returns an integer that represents the width of the node. If the width was not changed from the default one, it does not need to be implemented.
- `getHeight()` – Returns an integer that represents the height of the node. If the height was not changed from the default one, it does not need to be implemented.
- `isNodeHidden()` – Returns a boolean if the node should be hidden by default in the `NodesSidebar` component.
- `getCodePrefix(langId)` – Returns a prefix that precedes the parameters listing for a given language ID. In most cases this means the name of the method the node is converted into. It is important to note that if the node translates into a method call, this prefix should also include an open bracket, for example "`filter(`".
- `getCodeSuffix(langId)` – Similar to `getCodePrefix()`, but returns the suffix instead.
- `getCodeParameters(langId)` – For a given language ID, returns array of objects which represents all possible parameters for the node.
- `getOutputDataType(langId)` – Returns string that represents the data type which the node template emits.
- `isInputDataTypeValid(dataType, langId)` – Returns true if the passed data type is a valid input data type of the node template.
- (requiresBreakChaining()) – Returns true if the presence of the node should interrupt the chain of method calls, otherwise returns false. Does not need to be implemented if it does not need to break the chain.
- `generateCode(parameters, langId)` – The only method which does not need to be implemented. It is a method which is called during code generation with the node's parameters and language ID and has to return a string with source code that represents the node and its parameters. There is a default implementation which uses `getCodePrefix()`, `getCodeSuffix(langId)` and `getCodeParameters(langId)` to deduce the source code which should be generated. But the author can overload this implementation and use his own.

The object that represents the parameters in `getCodeParameters()` can have up to five attributes, but only the *name* is required.

- *name* – name of the parameter.
- *description* – explanation of what the parameter does.

- *required* – boolean which specifies whether the parameter is required or not.
- *template* – string which is placed in the input box by default.
- *selectionStart* – it is possible that when the user focuses on the input field of the parameter, only part of the text is selected. This parameter specifies on which position the selection should start.
- *selectionEnd* – similar to *selectionStart*, but specifies the end position of the selection. If the value is "all" then the rest of the string is selected.

As mentioned earlier, `getName()` requires a JointJs model. In JointJs the developers can define custom shapes of the nodes through defining custom models. To ease the development, there is already the prepared model `DefaultShape`, which has most of the parameters predefined and follows the visual style of daGui. The only things the author has to specify are the name, the type and the ports of the node. It is also essential to add the new shape into the JointJs namespace. To better understand the possibilities, the author should consult the JointJs documentation [4] and the `DefaultShape` model.

5.1.3 Adapter's Components

daGui needs two components from the adapter's author — `ExecutionConfigurationForm` and `SettingsForm`.

`ExecutionConfigurationForm` is a component where the adapter presents to the user possible configuration parameters for the execution configuration. Its lifecycle is simple. The user selects a configuration which he would like to modify. The configuration is passed to this component, which displays all the possible parameters with pre-filled current values if there are any. The component handles the whole modification cycle and only when the user saves the configuration is it also saved in daGui. That means that validation of the parameters is also up to the form component. The important exception is the name of the configuration. As the name has to be unique across the adapter's configuration, there is a special callback dedicated for validation of the name, which is handled by a daGui's wrapper component. When the configuration is saved, it is persisted by daGui and later retrieved for execution. The component is fetched from the adapter's class with method `getExecutionConfigurationForm()`. The component has four properties which are used for passing data and callbacks from daGui.

- `configuration` – a property which holds the currently selected configuration. It can also be `null`, when no configuration is selected.
- `onUpdate` – a callback which is called by the form component when the user decides to save the configuration and the configuration is valid.
- `onClose` – a callback which closes the modal window.

[4] `http://resources.jointjs.com/docs/jointjs/v1.0/joint.html`

- `isNameValid` – a callback for validation of the name of the configuration.

`SettingsForm` is another component which is used for the adapter's specific settings. It is similar to the `ExecutionConfigurationForm`. When the user switches to the adapter's settings, daGui will fetch the already defined settings and pass them to the component, which will display the settings with pre-filled data. The component is responsible for the data validation. It is expected to have the following properties:

- `data` – the user's settings for the adapter, which were already previously set.
- `onUpdate` – a callback which the component is supposed to call with an object that represents the settings that are supposed to be saved.
- `onClose` – a callback which the component can call if it wants to close the Settings modal window.

5.1.4 Adapter's Tasks

There are two main tasks for the adapter — code generation and code execution.

The principle of *code generation* is rather simple. daGui calls an adapter's method `generateCode(...)` which takes a graph and some other parameters as input and returns generated source code. How the source code is generated is up to the adapter's author to decide and implement. Inspiration can be drawn from the referential implementation of the Spark adapter, as there are several issues which need to be solved (e.g., variable dependencies, branching). From the implementation perspective, it is important to know that the `generateCode(...)` does not directly return the code as a string. Instead, it has to use a `CodeBuilder` instance for storing the generated code which is passed as a parameter. The parameters of the `generateCode(...)` method are in the order:

- *output* – CodeBuilder instance for storing the generated code.
- *graph* – the input graph.
- *inputs* – the input nodes of the graph.
- *usedVariables* – an object which contains all used variables, where the key is the variable name and the value is the ID of the node the variable belongs to.
- *conf* – the currently active Execution Configuration, it can be null as the configuration is passed to it only during code generation for execution.
- *language* – the language which the code should be generated for.
- *languageVersion* – the version of the language the code should be generated for.

- *adaptersVersion* – the version of the framework the code should be generated for.

The second adapter's task is the code execution. Electron uses an architecture which splits the backend of the application (the main process, which can reach the operation system resources) and the front end of the application (the renderer process, which renders the web content). These two parts are completely separated, and the code from one part can not be used or imported in the other one. For communication between the parts Electron implements a system called *Inter Process Communication (IPC)*, which sends messages through channels and on each end there are callbacks registered for specific channels. Because of this architecture, the execution part of the adapter is split from the main adapter class and is implemented in a special adapter's executor which is based on the `BaseAdapterExecutor` class. It has the following methods, some of which have to be overridden.

- `getId()` – a method which has to return the same string Id as the adapter class.
- `handleStartExecution(event, generatedCode, conf, settings)` – a method which is called upon the start of the execution and implements the whole execution process of the adapter (compilation of source code if needed, launching the execution). The `conf` parameter consists of the selected execution configuration, the `settings` parameter contains the adapter's settings and the `event` parameter is an Electron object through which the executor can communicate with the renderer process.
- `handleTerminateExecution(event)` – a method which should terminate the running execution when called.
- `bootstrap()` – a method which does not need to be overridden. It is called upon initiation of the Electron to register the IPC channels handlers.
- `sendData(event, type, data)` – a helper method which does not need to be overridden. It sends data over to the renderer process over a channel specified by *type* parameter.

5.2 Spark Adapter

Apache Spark was chosen as a referential implementation of a framework's adapter. It has a simple API which serves well for the development process of daGui. However, as described in Section 2.4.1, Spark consists of several modules, which means that there are several groups of APIs. For the first stage, it was decided to implement the basic RDD and DataSet (DataFrame) API for Python binding.

As these two APIs are not compatible, it must be defined which nodes can be linked with what type of nodes. Therefore when a user starts to drag new link, daGui will allow connecting the link with only compatible nodes.

Moreover, there is a high number of nodes which are not used often. The Nodes Sidebar has a feature which hides these irregular nodes. Currently the definition on whether a node should be hidden or not is hard coded by my judgement. In the future, a user will be able to set this labelling in the daGui's settings, and later on, daGui should automatically learn which nodes are frequently used and which are not.

5.2.1 Graph Definition

The graph is an operator directed acyclic graph, where nodes are an operation performed on data and the links connect an operation's output with next operation's input. A node can have an unlimited number of outgoing nodes (output degree) if it has an outgoing port but can have only as many incoming links as the number of incoming ports (input degree).

A graph is a valid graph for this adapter if it fulfils all the four possible criteria:

- Has Input nodes – the adapter defines which node templates are Input nodes and then checks that there is at least one Input node present in the graph.
- Has all ports connected – check that all ports in all nodes inside of the graph are connected.
- Has all required parameters filled in – check that all parameters of the graph's nodes which are required are filled.
- No cycles are present in the graph.

5.2.2 Code Generation

The basic concept behind the code generation is a walk through the graph in Depth-First-Search (DFS) manner, while the rendering of the code is based on the principle of chaining of method calls which can be seen on Listing 3. Additionally, there are several other issues which needed to be taken into consideration – branching, variable naming and code dependencies. The pseudocode of the code generation algorithm is shown in Listing 4.

```
class Example:
    call(self):
        # ...some code...
        return self

    anotherFunction(self, ...):
       # ...some code...
        return self


variable = Example() \
```

```
            .call() \
            .anotherFunction(...)
```

Listing 3: Example of chaining methods in Python.

```
def processNode(output, node, graph, variableStack):

    if isInBreakSituation(node): # i.e., input degree of
       node > 1
        assignPreviousNodeVariableName(node,
           variableStack.pop())

        if not allPreviousNodesHaveVariableName(node):
            return; # Not all in-break dependencies are
               satisfied => backtrack
        else:
            output.add(generateCodeWithNewVariableName(
               node, variableStack.pop())) # breaks the
               chain and starts new one: newVariable =
               generatedCode(...)
            variableStack.push(theNewVariableName)

    else: # normal or out-break situation
        if afterOutBreakSituation(node): # i.e., output
           degree of previous node > 1
            output.add(generateCodeWithNewVariableName(
               node, variableStack.pop())) # breaks the
               chain and starts new one: newVariable =
               oldVariable.someMethod(...)
            variableStack.push(theNewVariableName)
        else:
            output.add(generateCode(node)) # continues
               the chain

    if isOutBreakSituation(node):
        multiplyTopVariableByOutputDegree(node,
           variableStack)

    if endOfBranch(node):
        variableStack.pop() # As the chain is at the end,
            the top variable won't be needed anymore.
        return # No more next nodes => backtrack

    for nextNode of node.nextNodes:
        process(output, nextNode, graph, variableStack)
```

```
def generateCode(graph, inputs):
    variableStack = Stack()
    output = CodeBuilder()
    output.add(getInitBlock()) # All includes and
       initialisation part of the code

    for inputNode in inputs:
        variableStack.push(inputNode.variableName)
        output += processNode(output, node, graph,
           variableStack)

    return output
```

Listing 4: Pseudocode of the code generation of the graph.

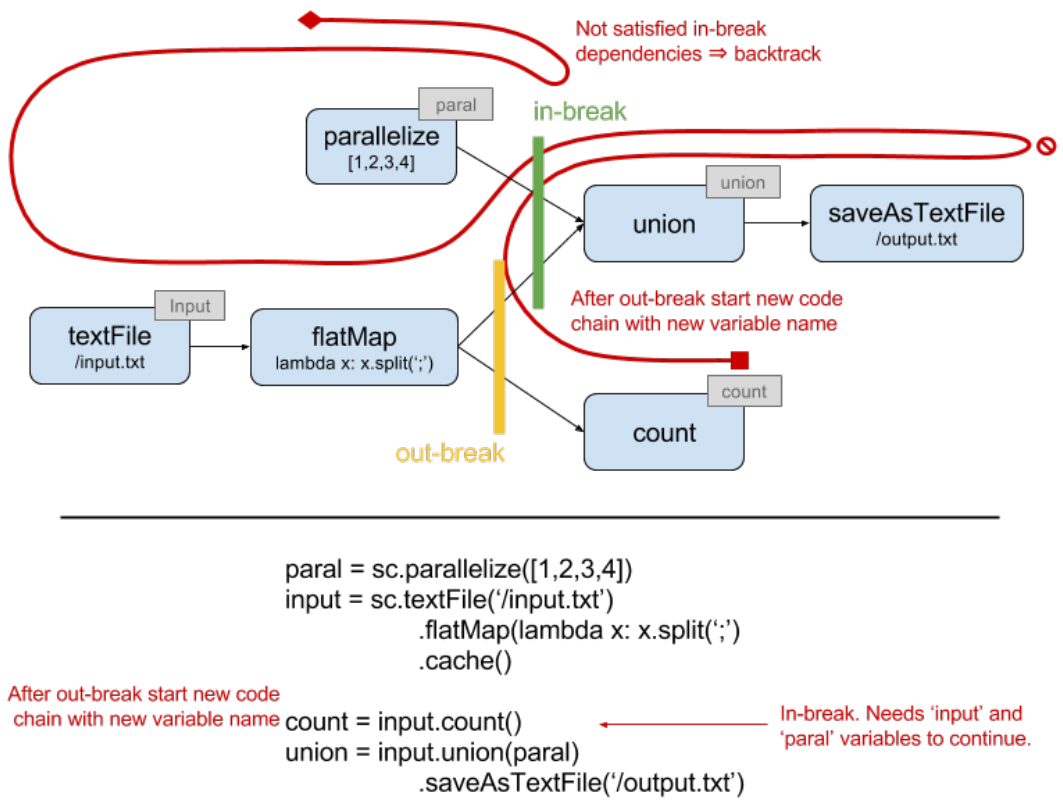

Figure 18: Example of a Spark's DAG with branching. The red line indicates walk-through of the DFS.

Branching and Variable Naming

During the DFS there is a need for generating variable names. These names are stored in a *variable stack* which is used during branching situations as described later. For every Input node, a variable name is generated and placed on top of the variable stack. Moreover, when the DFS reaches the end of the graph, it pops a top variable name out of the variable stack as it will not be needed anymore.

During the DFS walk-through there can be three branching situations based on the input and output degree of the current node:

1. normal situation (input and output degree of the current node is one),
2. out-break situation (output degree of the current node is higher than one), or
3. in-break situation (input degree of current node is higher than one).

During a *normal situation* the node is translated into a method call and appended to previous method calls based on the chaining principle.

When an *out-break situation* happens, the chaining of methods needs to be interrupted. The node where the out-break happens multiplies the variable on top of the variable stack by the output degree of the node and then calls the generation on the following nodes with a special flag, which indicates that there was an out-break situation. These nodes then, based on this flag, pop a variable name from the variable stack which is used for starting a new chain. At the same time, a new variable name for the new branch is generated and pushed on top of the variable stack. An example of this situation is in Figure 18.

When an *in-break situation* happens, it is required that all the previous nodes be processed before the DFS can continue out of the in-break situation. Therefore if some previous node is not processed the recursive process is halted, and backtracking is applied, as this guarantees that all previous nodes will eventually be processed, because the graph is valid as described in Section 5.2.1. The need of all previous nodes to be processed is based on the fact that, for generating the method call for the in-break node, all previous variable names are needed.

Figure 18 shows examples of both in-break and out-break situations with visualising the walk-through of the DFS. It can also be seen from the Figure that both in-break and out-break situations can happen simultaneously.

Code Dependencies

As the order of evaluation in the source code is defined, because the source code is read sequentially, it enables the use of previous evaluations's results in subsequent expressions. Most, if not all, programmers take this for granted and do not think much about it, but since the graph does not behave in a sequential manner, the use of results of some evaluation in the graph becomes more complicated.

There needs to be a way to reuse results of evaluation in some other parts of the graph. As described in Section 5.2.2 the nodes are assigned variable names when the Input node is processed, during in-break and out-break situations. Therefore

these variables can be used to reference the output of some other evaluation, for example in the anonymous functions of `map` or `filter`. This referencing creates code dependencies between parts of a graph, which need to be resolved during the code generation as the referenced variables need to be created and evaluated before its usage. There are two types of dependencies – cross-graph and branch dependencies.

Cross-graph dependencies emerge when there are at least two independent graphs present, and a node of one graph is dependent on a variable from the second graph. Therefore the second graph needs to be evaluated before the first graph. An example of such a dependency can be seen on Figure 19.

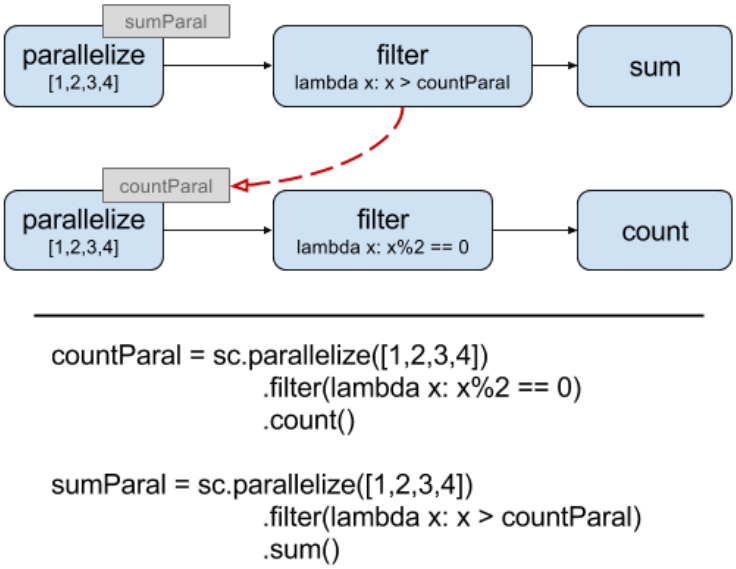

Figure 19: Example of cross-graph dependency between two graphs. The red line indicates the dependency.

Branch dependencies arise when some branch references a variable created by another branch of the same graph. This requires evaluating first the branch which is referenced by the other one. An example of such a dependency can be seen on Figure 20.

To resolve these dependencies, there is a pre-processing step before the actual code generation, which walks through the graph in DFS order and gathers the dependency graph. Then, based on the dependency graph, the order of iterating through Input nodes is defined in order to resolve the cross-graph dependencies and the order of iterating through branches is also defined to resolve the branching dependencies. Orders are determined using topological sorting. The topological sorting ensures detection of circular dependencies. When a circular dependency is found, an error is raised because it is not possible to generate code with circular dependencies.

5.2.3 Code Execution

As the referential implementation of Spark consists only of support for the Python language, it simplifies the code execution because Python is a scripting language

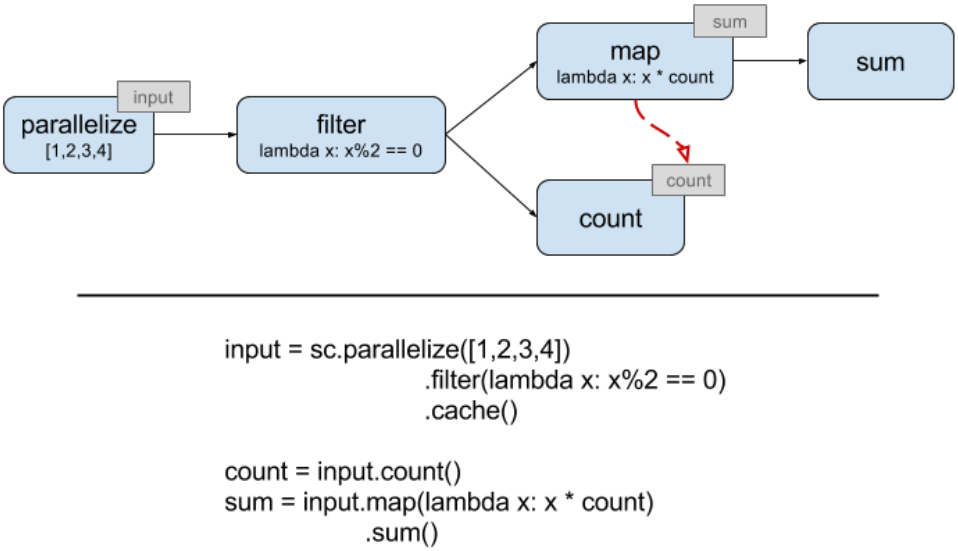

Figure 20: Example of branch dependency between two branches of the same graph. The red line indicates the dependency.

which does not need a compilation step like Scala or Java languages do.

For launching the execution, Spark has a special command line utility called `spark-submit`. This utility submits the Spark job to a given environment (local, cluster, Mesos or Yarn mode). It has several parameters which can be set and all these parameters can be set in the Execution Configuration of the Spark adapter. They are then used when launching the execution.

For the launching and compilation, the Spark's binaries and libraries are needed. Currently, daGui expects that this dependency is fulfilled by the user. It searches the Spark's home folder using the typical environment variable `SPARK_HOME`. If the binaries or the variable is not found an error is raised, and execution is terminated. As leaving this dependency resolution on the user is not user-friendly, in the future daGui will have a system which will automatically download the Spark binaries from Spark's homepage. This will also enable easy switching between the execution of different Spark's version because currently, the user has to set the `SPARK_HOME` environment variable to point to the directory which contains the desired version.

When the execution is launched, daGui asks the adapter to generate the source code for the given execution configuration. This code, together with the execution configuration and adapter's settings, is then passed through the Platform adapter to Adapter Executor. The executor builds the command line command from the Execution Configuration and then spawns a new process with it. All the output from `STDOUT` and `STDERR` is transferred back through the Platform adapter to the `ExecutionReporter` component, which displays the execution process.

In the future, when the Spark's adapter is extended with support for Scala and Java languages, the compilation step will be needed to create a Jar file which will be submitted to `spark-submit`. For that, the Java Development Kit and Scala binaries will be needed, to compile the source code into bytecode and then bundle it into the

needed Jar file. This step will happen prior the execution, and the Jar file will be stored in a temporary directory.

6 Evaluation

This Section will offer an evaluation of daGui through several examples, and also a discussion of its achievements and future work.

6.1 Graph and Generated Code Examples

This Section will present five examples of graphs and the generated source code for them. The reader should keep in mind that the code does not have any useful function, it serves only to demonstrate the code generation possibilities of daGui.

The first example is a simple one, with one out-break situation. It consists only of RDD based nodes and has no code dependencies inside the graph. You can see the graph on Figure 21 and the generated code on Listing 5.

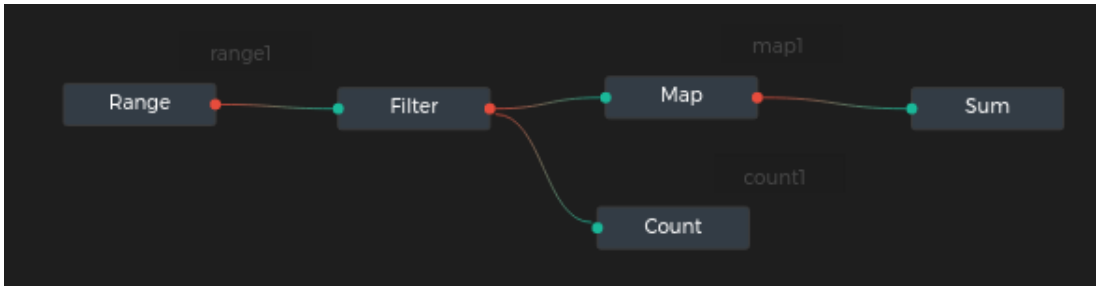

Figure 21: An example of a simple RDD based graph

```
from pyspark import SparkConf, SparkContext, SparkSession

conf = SparkConf()
sc = SparkContext('local', 'test', conf=conf)

range1 = sc.range(5) \
    .filter(lambda x: x%2 == 0) \
    .cache()

map1 = range1.map(lambda x: x*2) \
    .sum()

count1 = range1.count()
```

Listing 5: Generated code for Figure 21.

The second example contains both types of nodes — RDD and DataFrames. It has a conversion of the DataFrame branch into an RDD. Moreover, it has an in-break situation. Notice that SparkSession is created, because daGui detected the presence of the DataFrame's nodes. The graph can be seen on Figure 22 and the generated code on Listing 6

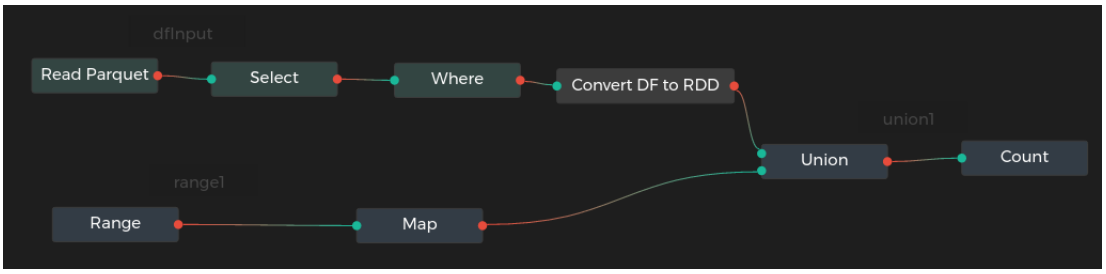

Figure 22: An example with a graph that contains two different types of nodes based on RDD and DataFram API.

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

conf = SparkConf()
sc = SparkContext('local', 'test', conf=conf)
sparkSession = SparkSession.builder.getOrCreate()

dfInput = sparkSession.read.parquet("/user/test.parquet")
    \
    .select("id", "name") \
    .where("name LIKE '%adam'") \
    .rdd

range1 = sc.range(5) \
    .map(lambda x: x*2)

union1 = sc.union([range1, dfInput]) \
    .count()
```

Listing 6: Generated code for Figure 22.

The third example contains a conversion of RDD to DataFrame. This conversion is a special case because it requires breaking the chaining even, although there is no in-break or out-break situation. The need of breaking the chain is detected through the node's definition, where the method of the node's template `requiresBreakChaining()` returns `true`. The graph of this example can be seen on Figure 23 and the generated code on Listing 7.

Figure 23: An example of conversion of an RDD branch into a DataFrame.

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

conf = SparkConf()
sc = SparkContext('local', 'test', conf=conf)
sparkSession = SparkSession.builder.getOrCreate()

load_text_file1 = sc.textFile("some/text/file.txt") \
    .filter(lambda x: x > someValue) \

convert_rdd_to_df1 = sparkSession.createDataFrame(
   load_text_file1) \
    .orderBy(["someColumn"]) \
    .write.parquet("some/path/file.parquet")
```

Listing 7: Generated code for Figure 23.

The fourth example has code dependencies between the branches. The Filter node's function depends on the result of the Count node. Therefore the lower branch which contains Map and Count nodes has to be evaluated first so the result can be used in the Filter node. The graph of this example can be seen on Figure 24 and the generated code on Listing 8. Notice that `mappedValues` precedes the `filter` function.

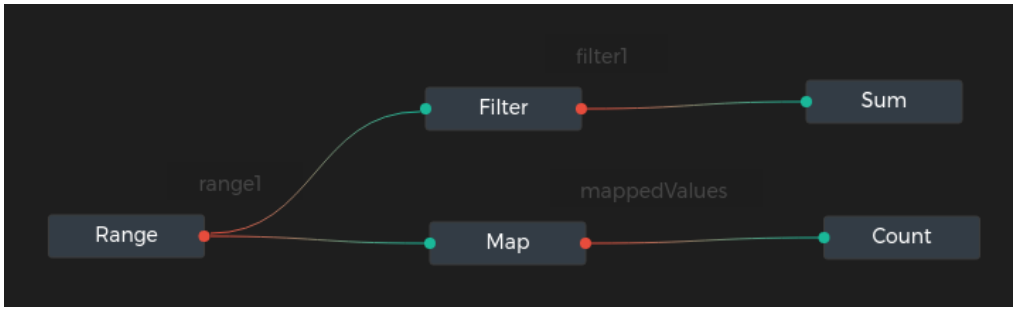

Figure 24: An example which contains code dependencies between the graph nodes

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

conf = SparkConf()
sc = SparkContext('local', 'test', conf=conf)

range1 = sc.range(5) \
    .cache()

mappedValues = range1.map(lambda x: x*2) \
    .count()
```

```
filter1 = range1.filter(lambda x: mappedValues > x) \
    .sum()
```

Listing 8: Generated code for Figure 24.

The last example also contains code dependencies, but between several graphs. In the last graph the Filter's function is dependent on the `load_text_file1` variable and in the first graph the Map' function is dependent on the `range1` variable. Therefore the order of the evaluation has to be: second graph, third graph and first graph. The graph can be seen on Figure 25 and the generated code on Listing 9.

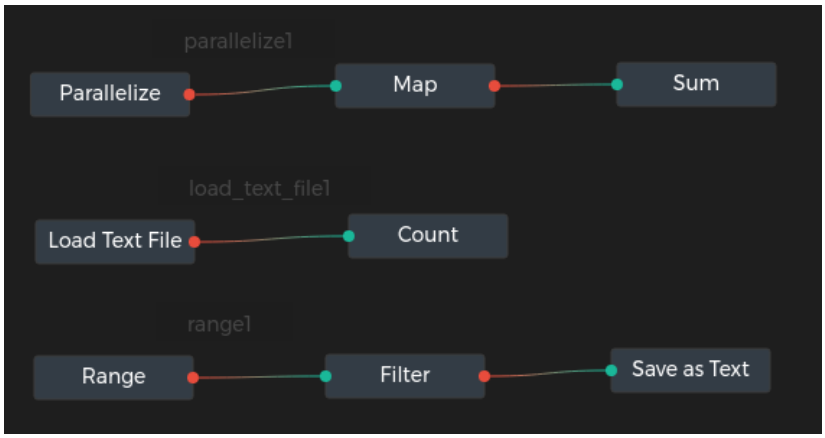

Figure 25: An example that contains several not connected graphs that have code dependencies between them

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

conf = SparkConf()
sc = SparkContext('local', 'test', conf=conf)

load_text_file1 = sc.textFile("some/path/file.txt") \
    .count()

range1 = sc.range(5) \
    .filter(lambda x: x - load_text_file1 > 2) \
    .saveAsTextFile("some/path/newFile.txt")

parallelize1 = sc.parallelize([1, 2, 3]) \
    .map(lambda x: x % range1 == 0) \
    .sum()
```

Listing 9: Generated code for Figure 25.

6.2 Discussion

daGui fulfilled two of the three main features proposed — code generation and code execution. Code parsing was dropped because of its great complexity. The current implementation of daGui presents the concept well, and has basic user-experience features such as history (undo/redo), copy&paste support, multi-node selections and so on. However, it is clear that there is still much work to be done on daGui to have a fully featured IDE-like software with great user experience.

The code generation algorithm sufficiently covers the possible usages as presented in the previous Section. However, we will need to enable the user to add his custom code, so custom libraries and packages can be imported, and custom functions can be created.

The current state of code execution support is sufficient for basic usage, but will also need improvements for more advanced usage, such as support for custom cluster managers.

6.3 Future Work

The current state of daGui represents the minimum required from IDE-like software programs, in order to be usable. During the development, several interesting features were identified which would help the user experience, but which were evaluated as "nice to have" features rather than critical features, and therefore did not make it into this implementation. Examples of such features are grouping nodes for a better overview, visualisation of the code dependencies between nodes, a delegation of the node's detail to the adapter's authors and more.

The main point of the future work is to improve the code base of daGui with proper code documentation, test coverage and extensible refactoring, to ensure a healthy code base.

The following step will be splitting the current code base into two branches — web-based branch and Electron branch. This will enable the Open Source community to incorporate daGui in their projects, such as integration with Hops Hadoop distribution. This step will require debugging the software on all web browsers, as the functionality is currently only guaranteed for WebKit-based browsers. Moreover, implementation of the Platform Connector for web environment will be needed.

The next step will be to focus on the user experience side of the software. Implement the previously mentioned "nice to have" features and improve the usability of daGui.

The last step will be extending the support of daGui with other Big Data frameworks. After finishing the support for all Apache Spark's API, the next framework which is planned to be supported is TensorFlow, which will most likely require implementing additional features so the adapter for the framework can be implemented.

7 Conclusion

The size of data in information systems grows rapidly. In recent years a new trend called *Big Data* emerged. It focuses on storing and processing a vast amount of data in a distributed environment. With the development of this trend, the community created new tools, libraries and frameworks. Several of these tools utilise Directed-Acyclic-Graph (DAG) for the execution in the distributed environment.

In this thesis the essential characteristics of Cloud computing and Big Data systems were gathered, leading frameworks which utilise DAG were surveyed, and several projects which focus on visualisation or visual programming concerning Big Data were listed. The main contribution of the thesis is an Integrated Development Environment (IDE) like software, which was designed from the bottom up and implemented. It is a multiplatform standalone application based on the Electron technology. The application is ready to be ported into a web environment as the core of the application is mostly a web application. During the design phase of the software, three main features were set: code generation, code execution and code parsing. From these three features, the code parsing feature was not implemented for two reasons: firstly the feature proved to be very challenging, mainly because it would require implementing Control flow support to enable parsing conditions and looping in the source code; secondly, the scope of developing the IDE like software was very time-consuming as it needed to have many features to support good user experience, otherwise nobody would use it. Therefore it was decided to drop the code parsing feature and instead focus on delivering easy to use and stable software.

The software was evaluated by implementing several examples in daGui, and then the output code was analysed. Additionally, discussion about the software was presented.

Even though we believe that we have delivered useful software, there is still plenty of work to do. The main things which we need to focus on in the future are increasing the test coverage, extend the support for more adapters for other frameworks, port the core of the application to a fully functional web application and lastly, work on the user experience side of the software.

References

[1] Ace editor homepage. Cited on 15.5.2017. https://ace.c9.io/.

[2] Dataiku. Cited on 15.3.2017. https://www.dataiku.com/.

[3] Deepsense Seahorse product webpage. Cited on 15.3.2017. https://seahorse.deepsense.io/.

[4] Electron homepage. Cited on 5.5.2017. https://electron.atom.io/.

[5] Electron React boilerplate GitHub repository. Cited on 15.5.2017. https://github.com/chentsulin/electron-react-boilerplate.

[6] Flux concept by Facebook. Cited on 15.5.2017. https://facebook.github.io/flux/.

[7] Hops: Hadoop open platform. Cited on 5.5.2017. http://www.hops.io/.

[8] Immutable.JS homepage. Cited on 5.5.2017. https://facebook.github.io/immutable-js/.

[9] JointJS diagramming library. Cited on 5.5.2017. https://www.jointjs.com/.

[10] React: A javascript library for building user interfaces. Cited on 5.5.2017. https://facebook.github.io/react/.

[11] Redux homepage. Cited on 5.5.2017. http://redux.js.org/.

[12] Topsort library. Cited on 15.5.2017. https://github.com/samuelneff/topsort.

[13] Webpack homepage. Cited on 15.5.2017. https://webpack.github.io/.

[14] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43. ACM, 2003.

[18] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613. USENIX Association, 2014.

[19] Avita Katal, Mohammad Wazid, and R. H. Goudar. Big data: Issues, challenges, tools and good practices. In Manish Parashar, Albert Y. Zomaya, Jianer Chen, Jiannong Cao, Pascal Bouvry, and Sushil K. Prasad, editors, *Sixth International Conference on Contemporary Computing, IC3 2013, Noida, India, August 8-10, 2013*, pages 404–409. IEEE, 2013.

[20] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[21] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[22] F. Michel. How many public photos are uploaded to Flickr every day, month, year? Cited on 25.2.2017. `https://www.flickr.com/photos/franckmichel/6855169886/`.

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[24] Krishnaiyan Thulasiraman and M. N. S. Swamy. *Graphs - theory and algorithms.* Wiley, 1992.

[25] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[26] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong

Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@Twitter. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014.

[27] Luis Miguel Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *Computer Communication Review*, 41(1):45–52, 2011.

[28] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[30] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013.