DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

daGui: A DataFlow Graphical User Interface

ADAM UHLIR

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Abstract

Big Data is growing trend. It focuses on storing and processing a vast amount of data in distributed environment. There are many frameworks and tools which enable to work with this data. Many of them utilise Directed Acyclic Graph (DAG) in some way. It is often used for expressing the dataflow of computation as it offers a possibility to optimise the execution because it contains the overview of the whole computation and not only its limited scopes. This thesis aims at creating 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 then can be transformed into a launchable source code. Moreover, the software offers simple way how 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 at helping beginners with learning these technologies, but experts can also use it as visualisation for their workflow or as a prototyping tool. The software was implemented using Electron and Web technologies, which ensures its platform independence. Its main features are code generation (e.g., translation of a DAG into the 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 future.

Big Data är en växande trend. Det fokuserar på att lagra och bearbeta stora mängder data i en distribuerad omgivning. Det finns flera ramverk och verktyg med vilka man kan arbeta med denna data. Flera av dem använder Direct Acyclic Graph (DAG) på något sätt. Det används ofta för att uttrycka dataflödet av beräkningen tack vare möjligheten att optimera utförandet i och med att det innehåller en överblick över hela beräkningen och inte bara en begränsad del. Detta arbetets syfte är att skapa en Integrated Development Environment (IDE) programvara, vilken är användarvänlig, interaktiv och lätt att utvidga. Programvaran gör det möjligt att rita en DAG som representerar ett programs dataflöde. DAG:en kan sedan omvandlas till en utförbar källkod. Dessutom erbjuder programvaran ett simpelt sätt att köra den skapade källkoden. Den kompilerar koden (ifall nödvändigt) och kör den baserat på användarens konfiguration som localhost eller cluster. Programvaran syftar primärt på att hjälpa nybörjare att lära sig dessa teknologier, men experter kan också använda den som en visualisation för deras arbetsflöde eller som ett prototypsverktyg. Programvaran implementerades med Electron och web teknologier vilka försäkrar plattformens självständighet. Huvudfunktionerna är skapande av kod (t.ex. översättning av DAG till källkod) och utförande av kod. Programvaran har skapats så att en utvidgning är möjlig, så att plug-ins för mer strukturer och verktyg kan stödas i framtiden.

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 hands and was always ready to spend a 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 feedbacks during the writing process and offered me many bits of advice.

I would also like to thank Jim Dowling for making 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.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Data — the primary drive of our time. The birth of Internet enabled easy communication and exchange of data across any distances. In its beginning, the usage was very limited, but after several decades 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 possible to share small bits of everybody's daily lives with their surroundings. But this visible type of data is just a 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 serves 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.

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, a price of hardware and so on). To support such a new paradigm, the community created new projects, tools and frameworks, which requires a bit different mindset while working with them as the distributed environment possess specific restrictions and characteristics. To tackle these challenges authors of several of the frameworks used Directed-Acyclic-Graph (DAG) dataflow, for example, Apache Spark, TensorFlow 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 at creating 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 any big hassle of setting up the environment (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 the programs. Lastly, it can be used as 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 focus 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.

Chapter 2 introduces concepts and overview of distributed computation. Chapter 3 presents the high-level design of the software, which is created as part of this thesis. Chapter 4 presents used technologies and implementation details. Chapter 5 describes the referential implementation of the Spark adapter and how to implement custom adapter. Chapter 6 evaluates the results of the software and Chapter 7 offers concluding remarks.

1.1 Contribution

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

Chapter 2

Background

This Chapter 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 its 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 by Section 2.3. In following Section 2.4 introduces several of 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)* [4]. 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 mobile devices). The application behaves as a monolithic unit, so the user is not aware of the deployment setup, nor 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 (operation 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 by who is the Cloud infrastructure managed 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 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 wide variety of tasks. An example 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 there was uploaded around 611 millions of pictures during the year 2016 [5]. With 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 environment mentioned by *Vaquero et al.* [6] – 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 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, then 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 in this environment. *Katal et al.* [7] surveyed the main difficulties and issues. They categorised 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 component increase enormously. Therefore the fault tolerance and recovery need to be taken into consideration when designing a 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 variate, the platform needs to be flexible about increasing or decreasing the numbers of workers running the execution.

2.2 Hadoop

In 2003 *Ghemawat et al.* from Google published work on Google File System (GFS) [8] and a year later *Dean et al.* also from Google published work about their distributed computation framework MapReduce [9]. These two papers inspired 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 basic version it incorporates several modules:

- Hadoop Distribute File System (HDFS) – storage module which creates distributed file system and handles fault tolerance.
- Yet Another Resource Manager (YARN) – resource manager which schedule 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 and more.

2.3 Directed Acyclic Graph (DAG)

The computation frameworks which will describe 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* [10].

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 *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 there exists a closed walk 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* and vice versa, if for directed graph exists topological order, then the graph 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 enlist several of frameworks for processing Big Data, which in some way utilise DAG, describe how they employ it and explain the basic programming paradigm of the frameworks.

2.4.1 Spark

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

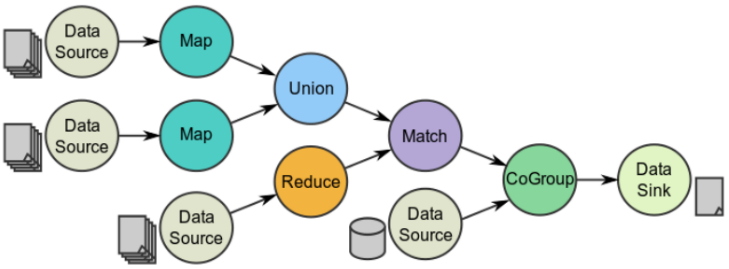


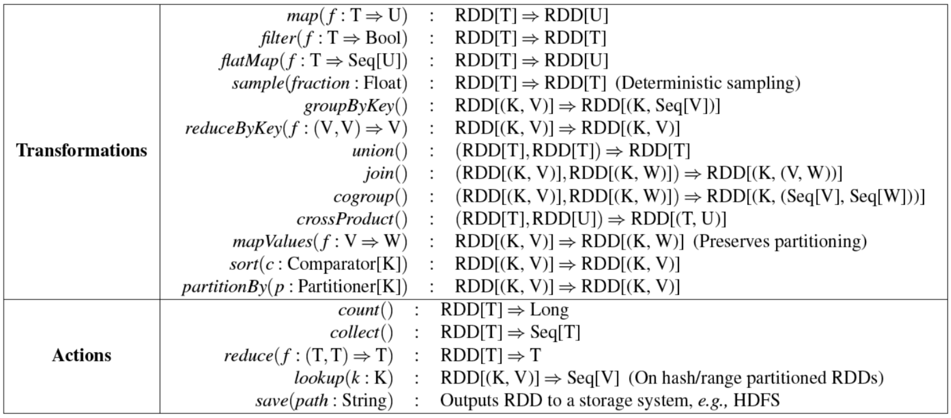
Figure 2.1: Example of Spark's DAG Dataflow.

Hence *Zaharia et al.* [1] proposed *resilient distributed datasets (RDDs)* that is an in-memory, fault-tolerant, parallel data structure, which they implemented into project call Spark (now under 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 of this data structure, they had to consider the specific characteristics of the in-memory approach. They could not use 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 to create a lineage of transformations, which then can be used for re-computation in a 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. Through the Spark's API, the code defines an operator DAG, that is then passed to DAG Scheduler which performs 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 compress as most tasks as possible into the single stage as all tasks of a stage are performed on single partitions of the data and does not need any exchange of data (shuffling). After dividing tasks into stages, they are passed to Task Scheduler, which handles the planning of execution in cooperation with the cluster manager.

There are two types of functions in 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 output can be anything. The transformations behave in a lazy manner, and when the code's executor reach an action, it evaluates all the previously defined transformations up to the action and then continues to rest of the code. Figure 2.2 presents a basic list of Spark's functions. Except for RDDs API, Spark consists of several other modules which extend the basic RDDs behaviour:

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



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 2.2: Example of Spark's API as presented in [1].

2.4.2 TensorFlow

TensorFlow [2] 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 very efficient manner.

Compared to Spark's MLlib, TensorFlow is rather low-level. Instead of being constrained only to several implemented algorithms (as in MLlib), in TensorFlow

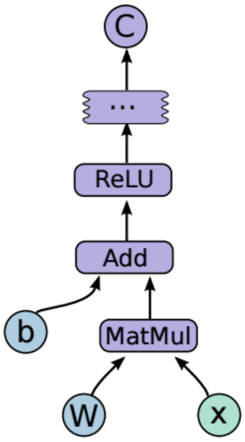
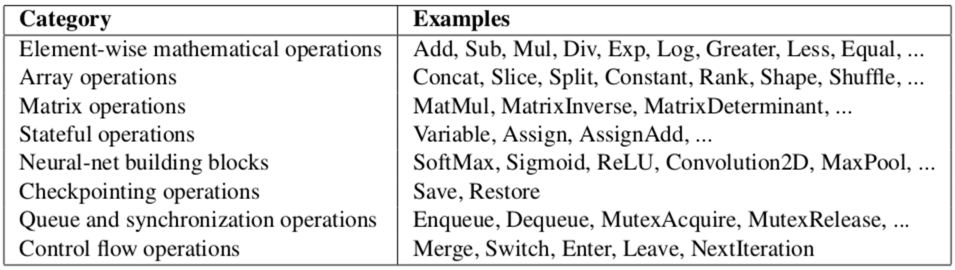


Figure 2.3: Example of TensorFlow's DAG as presented in [2].

you define the exact computation yourself. The computation is defined as a directed graph, where nodes are *operators* which modifies *tensors* that flow along the normal edges in the graph. Operators can have zero or more inputs and zero or more outputs. In the case of TensorFlow, the underlying representation is not a DAG but just a directed graph as it supports looping.



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 2.4: Example of TensorFlow's API as presented in [2].

2.4.3 Storm

Storm [11] is a real-time stream data processing system originally developed in Twitter (now under 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 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 next set of bolts. Similarly, as in TensorFlow, the representation is not a DAG, but directed graph as Storm supports loops.

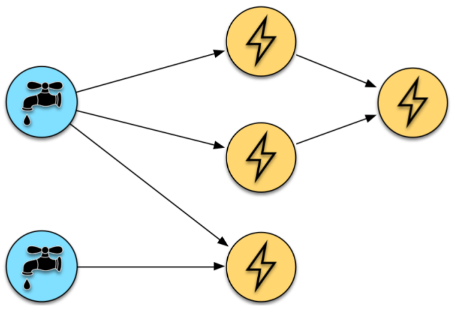


Figure 2.5: Example of Storm's DAG.

2.5 Related work

There are several projects which in some way tackle a similar problem or has 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 company Deepsense, which specialise on Big Data Science.

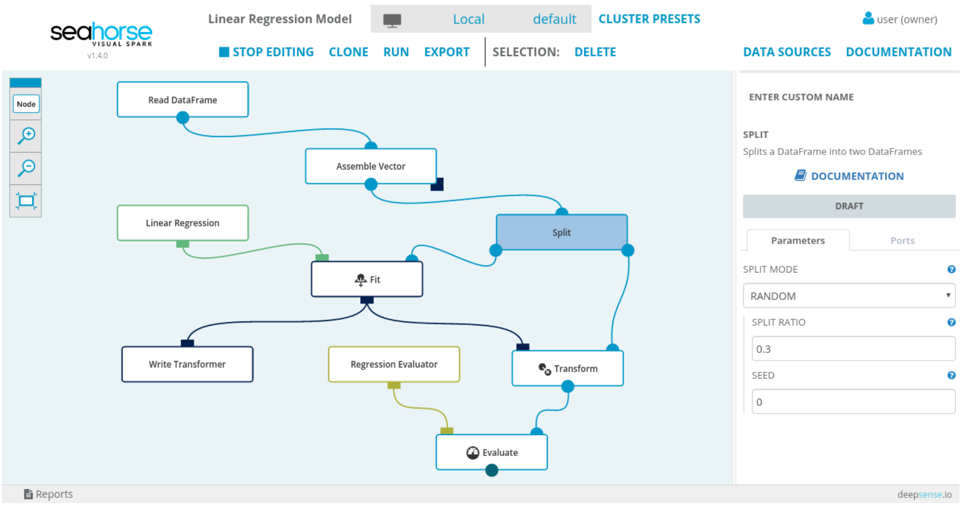
The editor focuses on high-level programming of Spark jobs as it offers predefined transformations, so the users do not have to write any code, just simple drag&drop nodes, connect them and specify its properties. This simplicity enables to create Spark jobs even for people not so proficient in programming,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

Except defining the 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 whole user interface and range of features.

2.5.2 Spark Web Interfaces

In Spark 1.4 release, the Spark's developers added DAG visualisation to Spark Web Interfaces. When a user submits Spark job to cluster, it has its Web interfaces, where the user can monitor the status of the job. To make it more easy 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 and does not offer any interactivity.



ENTER CUSTOM NAME

SPLIT
Splits a DataFrame into two DataFrames

DOCUMENTATION

DRAFT

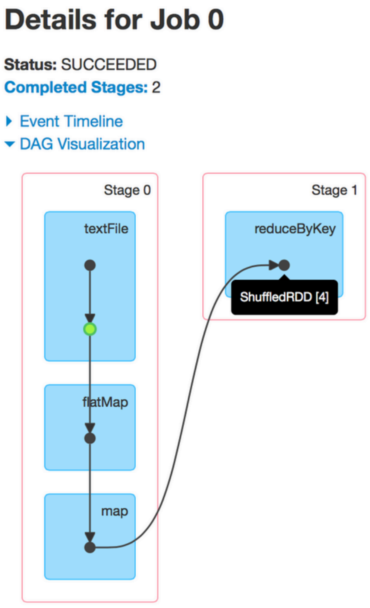
Parameters | Ports

SPLIT MODE: RANDOM
SPLIT RATIO: 0.3
SEED: 0

Figure 2.6: The interface of Seahorse editor [3].

2.5.3 Dataiku

Where Seahorse is the specialised tool for creating Spark jobs, Dataiku [12] is more a data science Swiss knife. It is a collaborative platform for data science integrating wide variate 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.


Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

Figure 2.7: Example of Spark's DAG visualization.

Chapter 3

Design

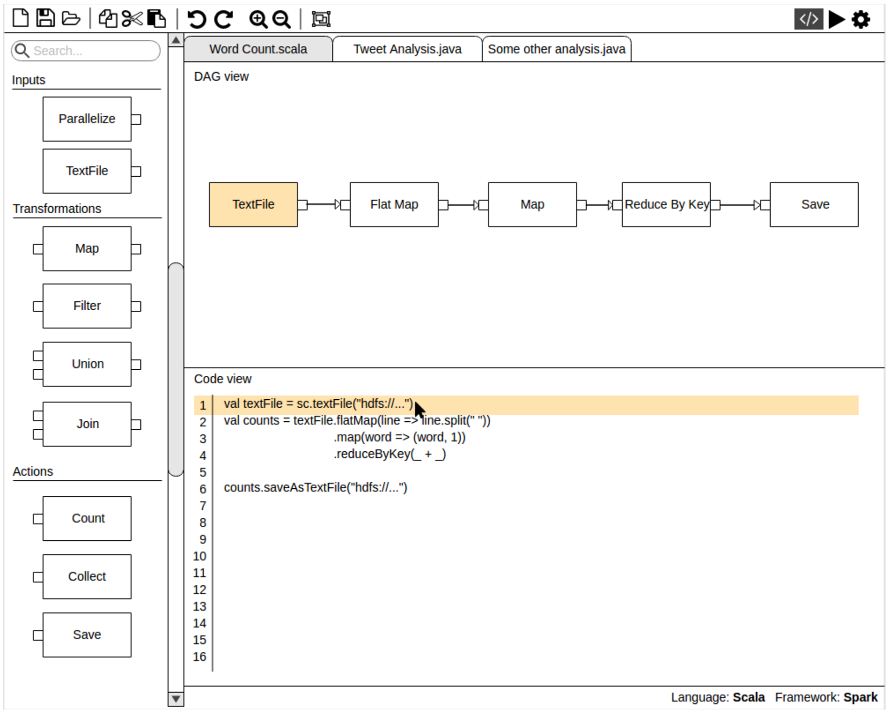
This Chapter will cover the high level details of the software which is developed as part of the thesis. It will describe the basic goals of the software (Section 3.1), the used Adapter design patter (Section 3.3), graph validation (Section 3.4), code generation (Section 3.5), code execution (Section 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 an easy development of programs which are based on frameworks that use directed graph for program representation. It is a general tool, which provides an extensible platform for working with these frameworks.

Figure 3.1: daGui's logo.

To have an idea what daGui does and how is it doing see a mock-up of its basic interface on Figure 3.2. Users drag&drop nodes from the node's pallet. Then connect the nodes with directed links, to form a dataflow. After that, fill the parameters of the nodes (for example filtering function for `filter` node in Spark) and if the graph is valid, the code then is generated and can be executed locally or on a cluster, based on a settings.



```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Language: **Scala** Framework: **Spark**

Figure 3.2: Mock-up of daGui interface with Spark adapter.

3.1.1 Use-cases and users

When designing and developing software, it is important to need to know its purpose and its users. daGui most probably will not be utilised by experienced developers as primary IDE, because it is more efficient to write the code directly over drag&drop nodes, link them and fill their properties, but 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 graphical representation of graph can be very helpful. This use-case implies users who might not be so skilled in programming or with computer interaction, on the other hand, the users most probably will 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.

Another use-case is a presentation of the programs. Explaining what some piece of code does, can be sometimes 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 more an overview of the problem so new ideas of how to solve a particular challenge can be developed.

3.1.2 Goals

Before the start of works on daGui, there were several goals defined, that the program should fulfil.

As it is mainly 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 graph editor, the interactivity is higher than in any other typical IDE. This goal also correlates with the beginner users group identified in Section 3.1.1.

When 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 in 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 how 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 to edit 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 future.

3.2 Graph and its components

This Section will define and describe the graph and its parts that users create in daGui. On a general level, it is a directed graph with nodes and directed links (edges). It is up to 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 are not required to do so. For example, Spark's adapter uses it for naming variables in generated code.

Node has ports which define the input and output degrees of the node. Ports are two types: input ports and output ports. The ports are visualised as small dots on the node with a different colour for each type of the ports. The links between nodes are created between ports. daGui restricts the input ports, where one input port can accept only 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 it might happen that in future these settings will be generalised and it will be possible to set these constraints with the adapter's configuration.

In graph validation, there is used term *input nodes.* 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 have to be always the case.

Figure 3.3 shows an example of nodes, ports and links.

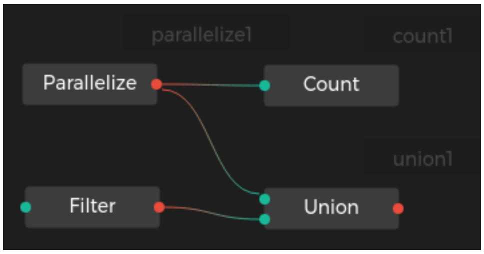


Figure 3.3: Example of nodes, ports (input ports are green and output ports are red), links and editable fields (grey text outside of the nodes).

3.3 Adapters

To fulfil the extensibility goal of daGui, its architecture needed to be built with this goal kept in mind from the early beginning of its development. daGui uses *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 process the results passed back from the adapter. An example of such a delegation can be a 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 criteria, which the graph needs to fulfil so that the adapter could generate valid code. More details in Section 3.4.

Node template defines a type of node in the graph, which is usually translated into 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 which it will be translated into 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 a generated code from *Code generation* task and user's configuration which specifies the parameters of the execution and launch it. More details in Section 3.6
- *Code parsing* – the task takes a source code file and produces graph representation which is then displayed to a 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 Chapter 5.

3.4 Graph validation

To be able to generate code out of the graph, it needs to be verified, that it is valid by the adapter's definition. As mentioned in Section 3.3, the framework's adapter does not perform the validation itself, but 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 check that there is at least one Input node in the graph present.
- 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, that are required are filled.
- No cycles are present in the graph.

The cycle detection uses the property of DAG that every DAG has a topological ordering as stated in Section 2.3.1. daGui implements 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. daGui uses for topological sorting an implementation from JavaScript library written by Marcel Klehr called `toposort` [1].

Future improvement will be to implement an algorithm for searching Strongly connected components, which exactly identifies 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 3.1: 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.

[1]`https://github.com/marcelklehr/toposort`

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 does not require the build stage.

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 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 can have 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 to import 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 [2]. However, also it consists of 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 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, it is not much 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 other language's features. Parsing of the code would consist of building 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 external dependency brings 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 how the control flow could be expressed in the graph is in Figure 3.4.

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 future this feature could highly improve daGui's capabilities. Therefore it will be one of the main points of the future work.

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

```
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 3.2: Example code which could be parsed.

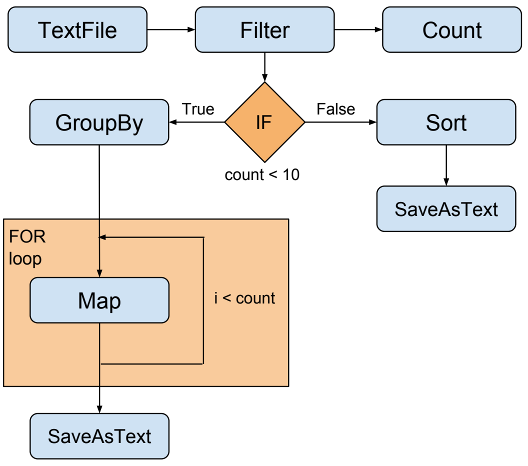


Figure 3.4: Example how the code presented in Listing 3.2 could be parsed and how the graph could look like with the control flow.

Chapter 4

Implementation

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

4.1 Technologies

During a survey of technologies for daGui, there was one important factor, and it was a 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 already in the beginning of daGui's development, the authors of Hops Hadoop distribution [13] approached the thesis author, that they would like to integrate daGui into their environment. This introduced another requirement of simplicity of porting the daGui into the 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 user's browser. The web browser would be the main entry point for the user. The advantage of this approach is mainly straightforward access to the user's OS and utilising well-known principles of web development. The big disadvantage is a distribution of such software as packaging and distributing is possible, but rather hard and inconvenient from user's point of view.

Electron [14] on the other hand is an entirely stand-alone program. It is essentially packed Chromium web browser with Node.JS as the application back-end and V8 JavaScript engine. Therefore writing an application with Electron is almost same as writing Front-end JavaScript web application. The main difference with Electron is 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 much better user experience as the application behaves as a monolithic unit. Moreover, as Electron development is almost identical as web development, it will be simple to convert daGui into a proper web application. 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 the high-speed internet is becoming standard and most of the users also have big memory storage.

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

React [15] is a rendering library which holds Virtual Document-Object-Model (DOM) representation and through that tries to minimise the changes in actual browser's DOM as they are rather expensive. Through React the developers create Components which defines some element on the web page with its full life-cycle. This architecture is highly useful for daGui, as a rendering of some adapter's specific parts can be delegated to 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 [16] is built on the idea of Facebook's Flux [1] 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 as the user requests. As in JavaScript objects are generally mutable another used tool is Immutable.JS [17], which has special API that enforces immutability on its special objects.

Last valuable library is JointJS [18]. 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 is 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" with name Webpack [19]. It is a handy tool which optimises the building process and especially 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.

[1] `https://facebook.github.io/flux/`

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 it right away. Electron React boilerplate [20] was chosen for daGui as it incorporates all the technologies mentioned earlier. There are several features, which were not needed and because of that they were removed, mainly it concerns React Router. There are some other features which are not actively used in daGui but remains in the project as in future development they might prove handy. 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

Except for 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 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 is 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 proper part of the code when needed.

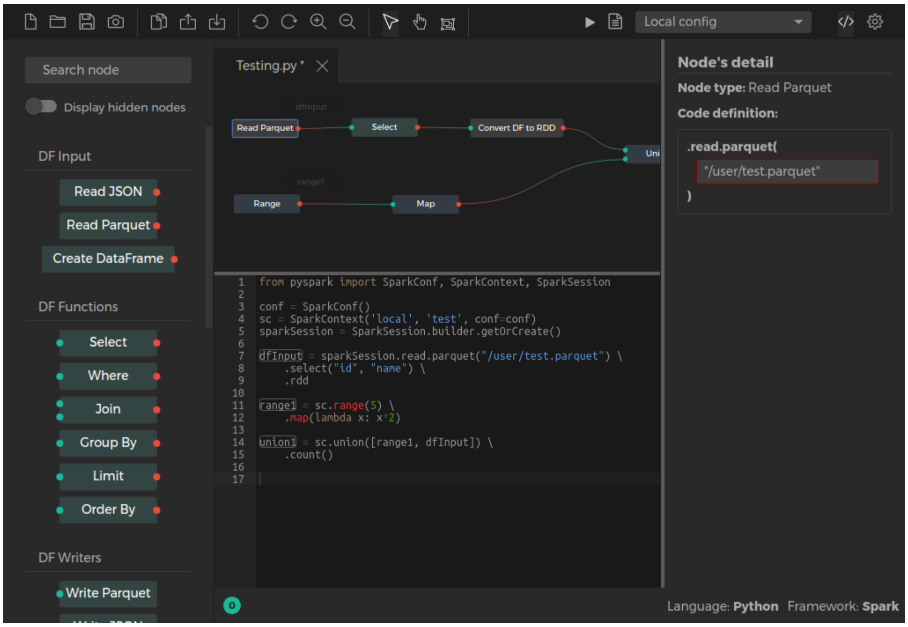
4.2.2 Image export

The last feature is handy for a presentation of a program. daGui can export the graph as a PNG image. It is easier than taking screenshots as it automatically renders 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 4.1. Except for the main editor view, daGui also has modal windows, an example of such a window can be seen on Figure 4.2.

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 prepared either icons that on hover display a help tooltip, or similarly help tooltip is shown on hovering over some input fields. An example of such a help can be seen on Figure 4.3 and Figure 4.2.



```
from pyspark import SparkConf, SparkContext, SparkSession

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

dfInput = sparkSession.read.parquet("/user/test.parquet") \
    .select("id", "name") \
    .rdd

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

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

```

Node's detail

Node type: Read Parquet

Code definition:

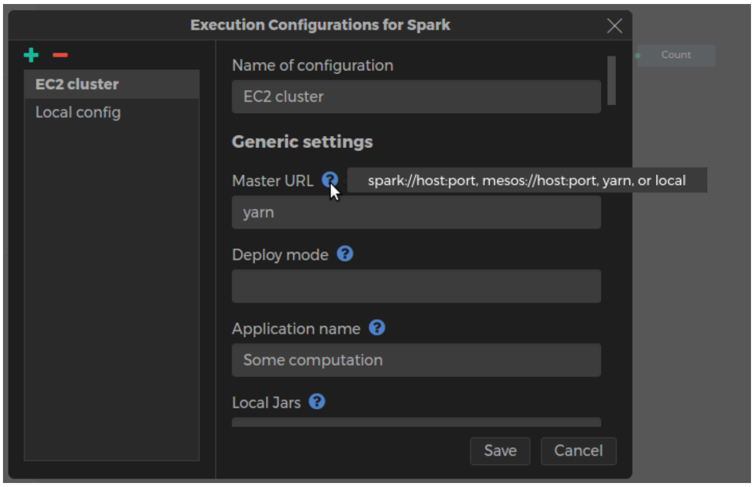
```
.read.parquet(
    "/user/test.parquet"
)
```

Language: **Python** Framework: **Spark**

Figure 4.1: Look of daGui editor.

4.4 Platform adapter

As in future, daGui will be ported into a web environment the daGui's architecture have 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 NodeJs, but in the web environment, they will be most likely delegated to a remote server using AJAX call.



Name of configuration
EC2 cluster

Generic settings

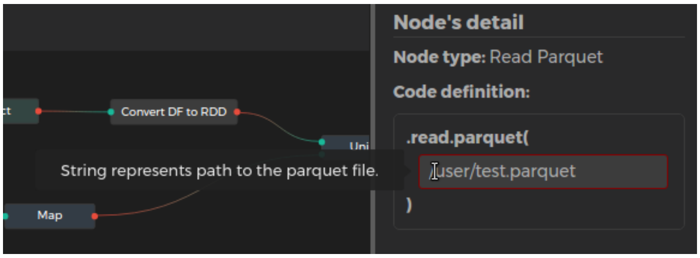
Master URL
spark://host:port, mesos://host:port, yarn, or local
yarn

Deploy mode

Application name
Some computation

Local Jars

Figure 4.2: Execution Configuration modal window with displayed help for configuration parameter.



String represents path to the parquet file.

Node's detail

Node type: Read Parquet

Code definition:

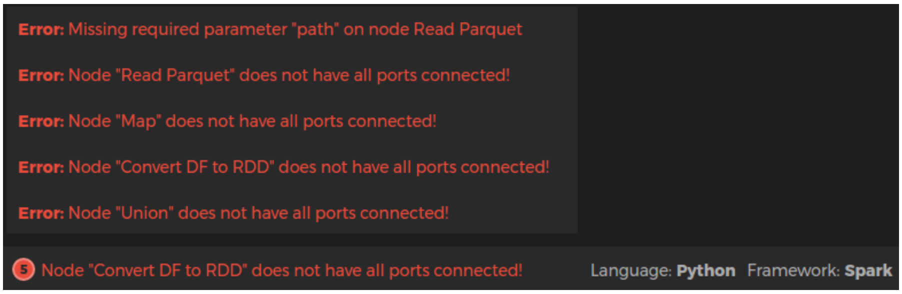
```
.read.parquet(
  /user/test.parquet
)
```

Figure 4.3: Detail of a node with displayed help for its parameter.

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 4.5 shows the role of Platform adapter in daGui's architecture.

The tasks of the Platform adapter are:

- 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.



```
Error: Missing required parameter "path" on node Read Parquet
Error: Node "Read Parquet" does not have all ports connected!
Error: Node "Map" does not have all ports connected!
Error: Node "Convert DF to RDD" does not have all ports connected!
Error: Node "Union" does not have all ports connected!
```

5 Node "Convert DF to RDD" does not have all ports connected! Language: **Python** Framework: **Spark**

Figure 4.4: Errors View which informs the user about graph's error.

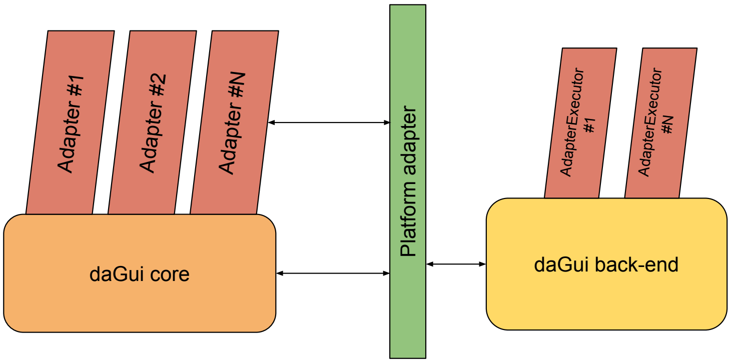


Figure 4.5: Overview of the architecture of daGui.

4.4.1 Persisting daGui's files

As Code parsing feature turned out as too challenging (as described in Section 3.7) and daGui is currently not supporting it, there has to be another way how 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, there is 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.
- Serialised JointJS object of the built graph.

From this meta-data, 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. During saving of the work, daGui generates a hash of the source code which is then stored alongside with other daGui's metadata at the end of the source code file. If during loading of the file is detected any difference, daGui raises warning to the user, that the original DAG and the source code might not match and loading and sequentially saving it might 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 therefore it is not able to generate the source code, daGui offers to the user, that only the meta-data will be saved into the file and the old source code in the file is preserved.

4.5 Components

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

Excepts Components in React there is also often used a concept of Containers. 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 one-page application, because the editor is always visible and for all other parts (settings, new file dialogue and others) is used the modal window, 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.

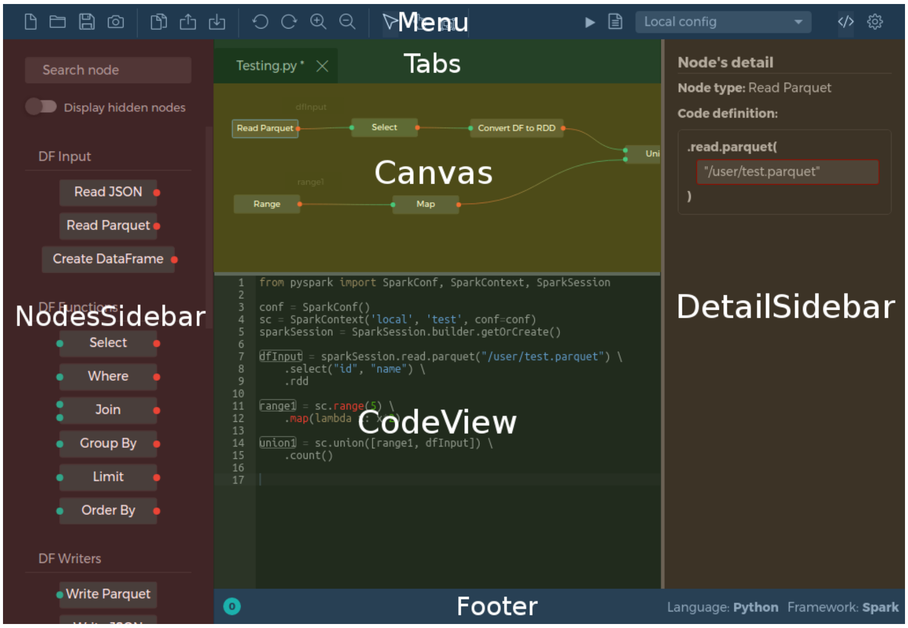
The overview of the key components can be seen on Figure 4.6.

- *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 to have opened 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.



```
from pyspark import SparkConf, SparkContext, SparkSession

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

dfInput = sparkSession.read.parquet("/user/test.parquet") \
    .select("id", "name") \
    .rdd

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

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

```

Figure 4.6: Overview of the main component in daGui.

- *DetailSidebar* – component that displays details of 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 reason, it is good to have as least as possible Redux's connected components (i.e., components that can directly access the Redux's state and fire

Redux's actions). One way to achieve that is to have just a few connected components which distribute the proper callbacks into its 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 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 overview 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 [18] library, but as the library has support only 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 begun 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.
- *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.

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

- *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 bit different layout (e.g., settings, new file dialogue). daGui follows the example of other IDE's, which use modal windows for this task. However, daGui has bit different implementation. The usual way is to open new system window and display the content in it. The modal window is separated from the main window. As daGui will be ported to web environment in 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

Last critical component is CodeView. It serves for displaying the generated code and offers a little degree of interactivity. The component employs Ace Editor [3] for highlighting the code's syntax. Additionally, it implements node's highlighting and also it is possible to rename the name of the variables inside the CodeView.

[3]`https://ace.c9.io/`

Chapter 5

Adapters

5.1 Implementing an adapter

It is a relatively simple process to implement a custom adapter, 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 does the framework relates to the graph, how does it use 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 also code execution, then the author also has to implement adapter's execution class which have to extends the `BaseAdapterExecutor` class and register it in the Electron's configuration `/app/config/electron.js`.

This Section will guide the adapter's author with the implementation. The recommendation is to have a look at the referential implementation of Spark's adapter as many 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. `BaseAdapter` class serves as 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 presentation of the adapter. `getId()` method should return some short unique identifier string. The string should contain only alphanumerical values without spaces and should be reasonable 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, method `getIcon()` should return path to an image which will be displayed as the representation of the adapter. Usually, it should be a logo of the framework. This is not mandatory and `getIcon()` can return `null`, in such a case daGui will display the adaptor's name instead.

Next part is associated with supported versions of the framework and its languages. Method `getSupportedVersions()` has to return an array of strings which represents the versions of the framework, which the adapter supports. The version is then always given for all other adapter's tasks such as code generation, execution and so on. Following is method `getSupportedLanguages(adapterVersion)`, that have to return an array of supported programming languages for given adapter's version. The languages in the array should be an 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)`, that have to return an array of strings that represents the supported language versions for given language and adapter's version.

Following part is connected to the graph's nodes. daGui uses a term *Node Template* as it represents only a template and not the node directly [1]. Method `getNodeTemplates(adaptersVersion)` have to return an object that contains all supported node templates for 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. Next method is `getGroupedNodeTemplates(adaptersVersion)` which enables to group node templates into groups that represents some similar function of the node templates. It should return an array of objects, which represents the group, but this method also can 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 given adapter's version is a node template's type an input node. The method returns boolean.

The second to last part relates to adapter's components. daGui currently needs two components from the adapter — `ExecutionConfigurationForm` and `SettingsForm`. They will be detailed in 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. With code execution is connected the last method called

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

`hasExecutionSupport()`, which returns boolean that specifies if the adapter supports code execution and has all necessary support implemented.

5.1.2 Node Templates

Node Template contains all information about how the node looks in the graph, what parameters it has and more. Node Template is a class which extends `NodeTemplate` class, that defines an interface for the Node Templates. It has following methods that the author have 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.
- `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 by default hidden in the `NodesSidebar` component.
- `getCodePrefix(langId)` – Returns a prefix that precede the parameters listing for given language ID. In most cases that means the name of the method the node is converted into. Important is that in case the node translates into method call, this prefix should also have start of the brackets, for example „`filter(`".
- `getCodeSuffix(langId)` – Similar to `getCodePrefix()`, but returns suffix instead.
- `getCodeParameters(langId)` – For given language ID returns array of objects that 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 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 return false. Does not need to be implemented if it does not need to break the chain.

- `generateCode(parameters, langId)` – 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 have to return a string with source code that represents the node and its parameters. There is 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 does the parameter do.

- *required* – boolean which specifies if the parameter is required or not.

- *template* – string which is by default placed in the input box.

- *selectionStart* – it is possible that when the user focus 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 as *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 prepared model `DefaultShape`, which has most of the parameters already predefined and follows the visual style of daGui. Only things that the author have 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 [2] and the `DefaultShape` model.

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

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 prefilled current values if there are any. The component handles the whole modification cycle and only when the user saves the configuration it is 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 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 on retrieved for the execution purposes. 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.
- `isNameValid` – callback for validation of the name of the configuration.

`SettingsForm` is another component which serves for the adapter's specific settings. It is similar with the `ExecutionConfigurationForm`. When the user switches to the adapter's settings, daGui will fetch the already defined settings, pass them to the component and the component display the settings with prefilled data. The component is responsible for the data validation. It is expected to have these 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 is the source code generated is up to the adapter's author to decide and to implement. Inspiration can be the referential implementation of Spark adapter as there are several issues which need to be solved (e.g., variable dependencies, branching). From the implementation perspective, there 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 to which 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 for which the code should be generated for.
- *languageVersion* – the version of the language for which the code should be generated for.
- *adaptersVersion* – the version of the framework for which 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, that renders the web content). These two parts are completely divided, 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 special adapter's executor which is based on the `BaseAdapterExecutor` class. It has following methods which some of them have to be overridden.

- `getId()` – a method which have to return the same string Id as the adapter class.
- `handleStartExecution(event, generatedCode, conf, settings)` – a method which is called upon start of the execution and implements the whole execution process of the adapter (if needed compilation of a source code, 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 is not needed to override. It is called upon initiation of the Electron to registrate the IPC channels handlers.
- `sendData(event, type, data)` – a helper method which is not needed to override. It sends data over to renderer process over 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 mean that there are several groups of APIs. For the beginning, it was decided to implement the basic RDD and DataSet (DataFrame) API for Python binding.

As these two APIs are not compatible, there has to 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 if node should be hidden or not is hard coded by my judgement. In future, a user will be able to set this labelling in the daGui's settings, and in further future, 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 a 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 much of 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 Inputs nodes – the adapter defines what node templates are Input nodes and then check that there is at least one Input node in the graph present.
- 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, that are required are filled.
- No cycles are present in the graph.

5.2.2 Code generation

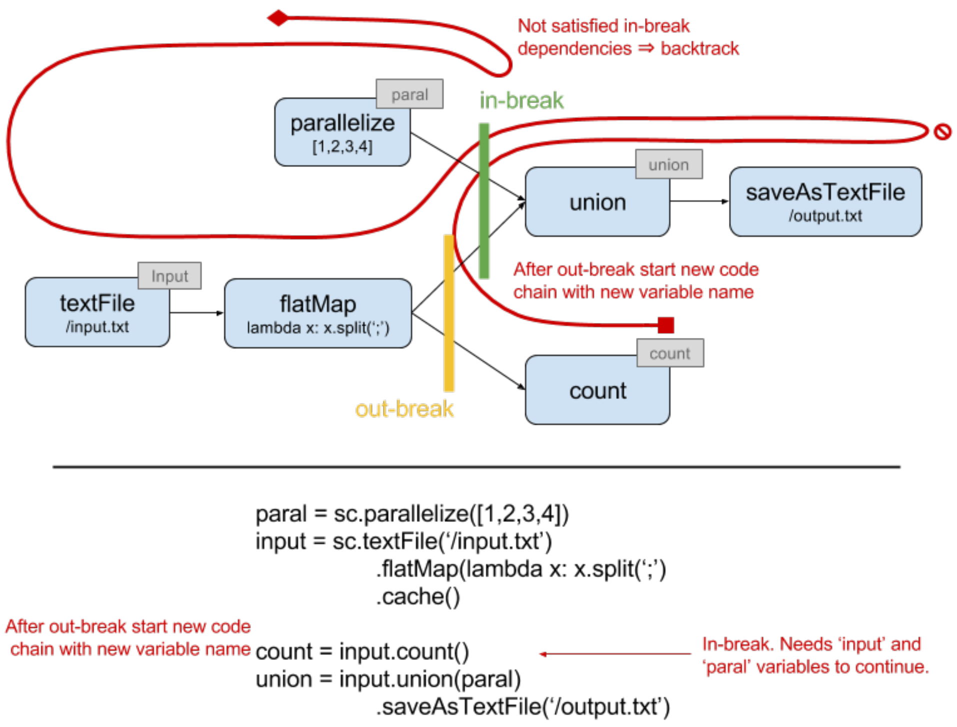
The basic concept behind the code generation is a walk through the graph in Depth-First-Search (DFS) manner, while the rendering of the code is based on the principle of chaining of method calls which can be seen on Listing 5.1. Additionally, there are several other issues which needed to be taken into consideration – branching, variable naming and code dependencies. The pseudocode of the code generation algorithm is shown in Listing 5.2.

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

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


variable = Example() \
            .call() \
            .anotherFunction(...)
```

Listing 5.1: 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 5.2: Pseudocode of the code generation of the graph.


Not satisfied in-break dependencies ⇒ backtrack
paral
parallelize
[1,2,3,4]
in-break
union
union
saveAsTextFile
/output.txt
Input
textFile
/input.txt
flatMap
lambda x: x.split(';')
After out-break start new code chain with new variable name
count
count
out-break
After out-break start new code chain with new variable name
In-break. Needs 'input' and 'paral' variables to continue.

```
paral = sc.parallelize([1,2,3,4])
input = sc.textFile('/input.txt')
             .flatMap(lambda x: x.split(';')
             .cache()

count = input.count()
union = input.union(paral)
             .saveAsTextFile('/output.txt')
```

Figure 5.1: Example of a Spark's DAG with branching. The red line indicates walk-through of the DFS.

5.2.2.1 Branching and variable naming

During the DFS there is a need for generating variable names. These names are stored in *variable stack* which is used during branching situations as described later. For every Input node, there is generated a variable name and is 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.

While the DFS walkthrough 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)
3. in-break situation (input degree of current node is higher than one)

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

When an *out-break situation* happens, then 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 the there was an out-break situation. These nodes then based on this flag pops a variable name from the variable stack which is used for starting a new chain. In the same time, a new variable name for the new branch is generated and pushed on top of the variable stack. Example of this situation is in Figure 5.1.

When an *in-break situation* happens, it is needed that all the previous nodes were 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 back-tracking is applied as it is guaranteed that all previous nodes will be eventually processed because the graph is valid in a sense 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.

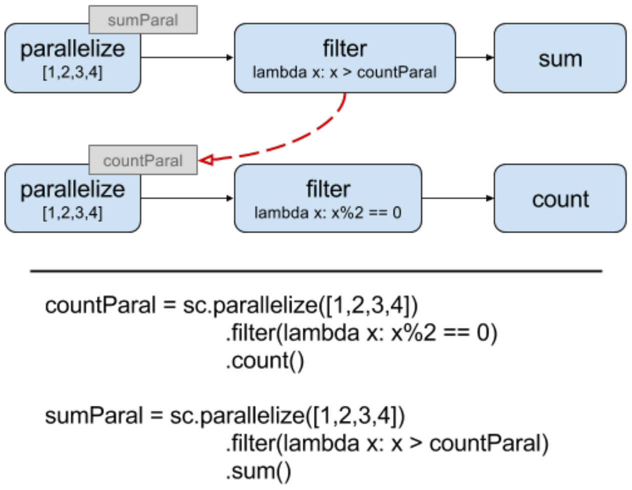
Figure 5.1 shows examples of both in-break and out-break situations with visualising the walkthrough of the DFS. Also from the Figure is visible that both in-break and out-break situations can happen simultaneously.

5.2.2.2 Code dependencies

As the order of evaluation in the source code is defined, because the source code is read sequentially, it enables to use the results of previous evaluations in later on expressions. Most, if not all, programmers take this as 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 way how to reuse results of evaluation in some other parts of the graph. As described in Section 5.2.2.1 the nodes are assigned variables name when 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 needs 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 some 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 5.2.



```
countParal = sc.parallelize([1,2,3,4])
                        .filter(lambda x: x%2 == 0)
                        .count()

sumParal = sc.parallelize([1,2,3,4])
                        .filter(lambda x: x > countParal)
                        .sum()
```

Figure 5.2: Example of cross-graph dependency between two graphs. The red line indicates the dependency.

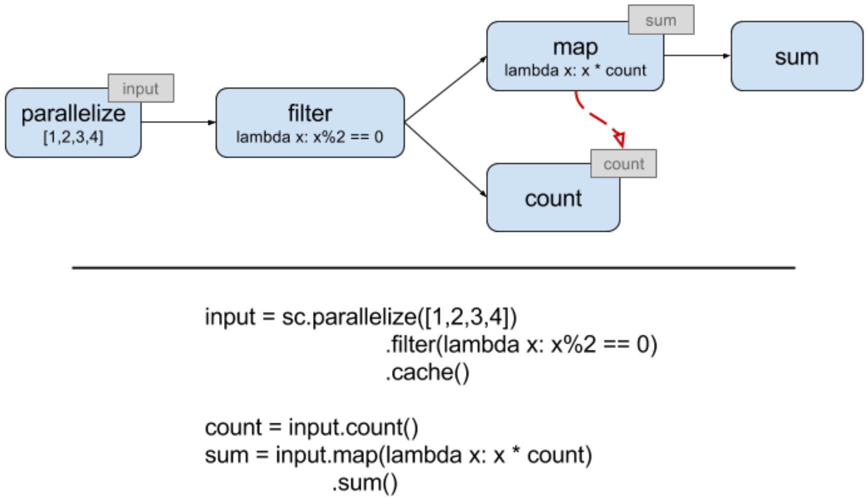
Branch dependencies arise when some branch reference a variable created by another branch of the same graph. This requires evaluating first the branch which is referenced by the other one. Example of such a dependency can be seen on Figure 5.3

To resolve these dependencies, there is a pre-processing step before actual code generation, which walks through the graph in DFS way and gather the dependency graph. Then based on the dependency graph the order of iterating through Input nodes is defined to resolve the cross-graph dependencies and order of iterating through branches is also defined to resolve the branching dependencies. Orders are determined using a topological sorting. The topological sorting ensures to detect circular dependencies. When a circular dependency is found, then 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 Python language, it simplifies the code execution because Python is a scripting language which does not need a compilation step like Scala or Java languages.

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



```
input = sc.parallelize([1,2,3,4])
                          .filter(lambda x: x%2 == 0)
                          .cache()

count = input.count()
sum = input.map(lambda x: x * count)
                     .sum()
```

Figure 5.3: Example of branch dependency between two branches of the same graph. The red line indicates the dependency.

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 then an error is raised, and execution is terminated. As leaving this dependency resolution on the user is not user-friendly, in 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 ask the adapter to generate the source code for 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 spawn a new process with it. All the output from `STDOUT` and `STDERR` is transferred back through Platform adapter to `ExecutionReporter` component, which displays the execution process.

In future when the Spark's adapter will be expended with support for Spark 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 bundled into the needed Jar file. This step will happen prior the execution, and the Jar file will be stored in a temporary directory.

Chapter 6

Evaluation

This Chapter will offer an evaluation of daGui through several examples and also discussion about 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 rational function, it only serves to demonstrate the code generation possibilities of daGui.

The first example is a simple one, with one out-break situation. It fully consists of RDD based nodes and has no code dependencies inside the graph. You can see the graph on Figure 6.1 and the generated code on Listing 6.1.

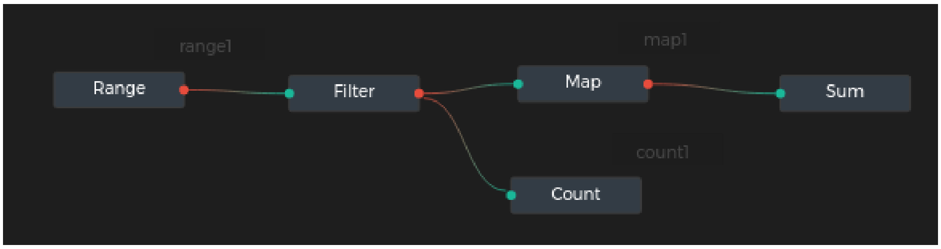


Figure 6.1: An example of simple RDD based graph

```
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 6.1: Generated code for Figure 6.1.

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 6.2 and the generated code on Listing 6.2

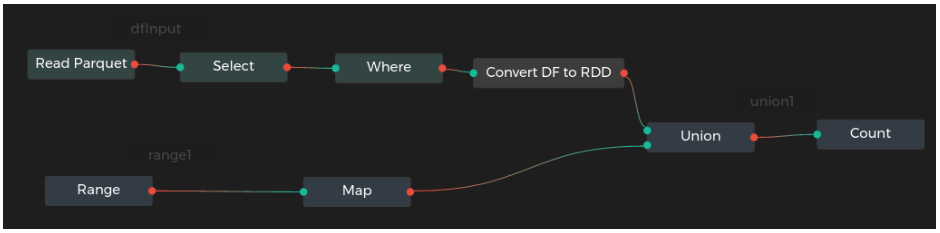


Figure 6.2: An example with graph that contains two different types of nodes based on RDD and DataFram API.

```
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 6.2: Generated code for Figure 6.2.

The third example contains conversion of RDD to DataFrame. This conversion is a special case because it requires breaking the chaining even though 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 6.3 and the generated code on Listing 6.3.

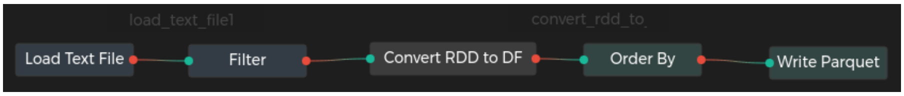


Figure 6.3: An example of conversion an RDD branch into DataFrame.

```
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 6.3: Generated code for Figure 6.3.

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 have to be evaluated first so the result could be used in the Filter node. The graph of this example can be seen on Figure 6.4 and the generated code on Listing 6.4, notice that `mappedValues` precedes the `filter` function.

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

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

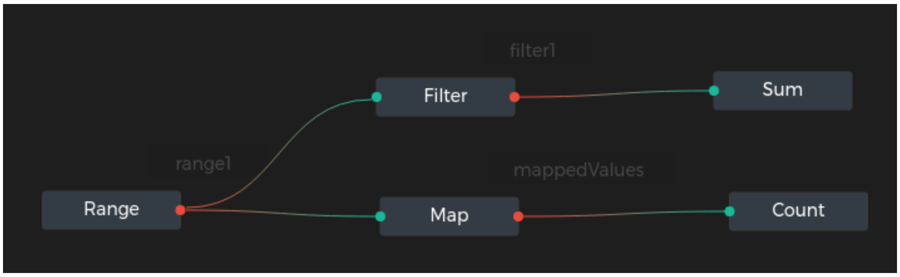


Figure 6.4: An example that contains code dependencies between the graph nodes

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

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

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

Listing 6.4: Generated code for Figure 6.4.

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 6.5 and the generated code on Listing 6.5.

```
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) \
```

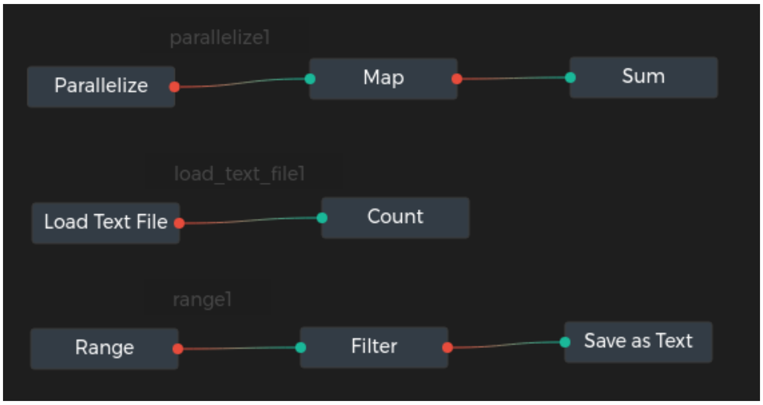


Figure 6.5: An example that contains several not connected graphs that has code dependencies between them

```
    .sum()
```

Listing 6.5: Generated code for Figure 6.5.

6.2 Discussion

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

There are three categories which will need to be worked up in the future — code quality, user experience and additional features.

Current code quality is not the best one. In software development, one third of the coding time should be devoted to refactoring and maintaining the code so that it would be well manageable and clear. During daGui's development, some time for this maintaining was spent, but as the complexity of daGui is rather high, the main development was focused on the defined feature set. Additional better code coverage will be needed to ensure stability in future development.

As mentioned several times, the user experience is critical for daGui's success. Even though there was much time devoted to creating daGui in a user-friendly way, there is still space for improvement. Especially the execution part of daGui will need to be improved, so the user does not have to resolve some of the dependencies himself.

Lastly, there are still many features which can be added in future to daGui. Support for more languages and frameworks will be crucial. The left out parsing feature and many more small, but handy features waits to be added.

Chapter 7

Conclusion

The size of data in information systems grows rapidly. In recent years new trend called *Big Data* emerged. It focuses on storing and processing a vast amount of data in distributed environment. With a 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 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 web environment as the core of the application is mostly web application. During designing 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 as very challenging mainly because it would require implementing Control flow support to enable parsing conditions and looping in a source code. Secondly, the scope of developing IDE like software is very time-consuming as it needs to have many features to support good user experience because otherwise, nobody will use it. Therefore it was decided to drop the code parsing function, 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 need to be focused on in future are increasing the test coverage, extend the support for more adapters for other frameworks, port the core of the application to fully functional web application and lastly, and work on the user experience side of the software.

Bibliography

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (S. D. Gribble and D. Katabi, eds.), pp. 15–28, USENIX Association, 2012.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous distributed systems,” *CoRR*, vol. abs/1603.04467, 2016.

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

[4] P. Mell, T. Grance, *et al.*, “The NIST definition of cloud computing,” 2011.

[5] 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/`.

[6] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, “Dynamically scaling applications in the cloud,” *Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[7] A. Katal, M. Wazid, and R. H. Goudar, “Big data: Issues, challenges, tools and good practices,” in *Sixth International Conference on Contemporary Computing, IC3 2013, Noida, India, August 8-10, 2013* (M. Parashar, A. Y. Zomaya, J. Chen, J. Cao, P. Bouvry, and S. K. Prasad, eds.), pp. 404–409, IEEE, 2013.

[8] S. Ghemawat, H. Gobioff, and S. Leung, “The Google file system,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003,*

SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003 (M. L. Scott and L. L. Peterson, eds.), pp. 29–43, ACM, 2003.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

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

[11] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (C. E. Dyreson, F. Li, and M. T. Özsu, eds.), pp. 147–156, ACM, 2014.

[12] "Dataiku. Cited on 15.3.2017." `https://www.dataiku.com/`.

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

[14] "Electron homepage. Cited on 5.5.2017." `https://electron.atom.io/`.

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

[16] "Redux homepage. Cited on 5.5.2017." `http://redux.js.org/`.

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

[18] "JointJS diagramming library. Cited on 5.5.2017." `https://www.jointjs.com/`.

[19] "Webpack homepage. Cited on 15.5.2017." `https://webpack.github.io/`.

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