Easy Spark

Y. van den Wildenberg
Eindhoven University of Technology
y.v.d.wildenberg@student.tue.nl

W.W.L. Nuijten
Eindhoven University of Technology
w.w.l.nuijten@student.tue.nl

O. Papapetrou
Eindhoven University of Technology
o.papapetrou@tue.nl

ABSTRACT

Today's data deluge calls for novel, scalable data handling and processing solutions. Spark has emerged as a popular distributed in-memory computing engine for processing and analysing a large amount of data in parallel. However, the way parallel processing pipelines are designed is fundamentally different from traditional programming techniques, and hence most programmers are either unable to start using Spark, or are not utilising Spark to the maximum of its potential. This study describes an easier entry point into Spark. We design and implement a GUI that allows any programmer with knowledge of a standard programming language (e.g., Python or Java) to write Spark applications effortlessly and interactively, and to submit and execute them to large clusters.

1 INTRODUCTION

Currently, data volumes are exploding in research and industry because of the increasing data-intensive applications. As a consequence, more and more disciplines face scalability concerns. However, the barrier to utilize distributed Big Data platforms is high for a multitude of reasons. Firstly, legacy code does not fit very well in Big Data platforms. Secondly, most senior programmers in the field – the ones usually taking the decisions – never had formal training on Big Data platforms. On top of that, the mere number of available Big Data platforms and pay-as-you-go solutions (e.g. cloud solutions) complicate the right choice for the user to scale-out, which increases again the barrier to entry. Because of this, companies are frequently reluctant to invest in Big Data platforms. In the long run, these companies will face either inability to scale or they will face higher cost for maintaining much stronger architectures in-house.

Spark is possibly the most popular Big Data framework to date. The framework implements the so-called master/slave architecture. It includes a central coordinator (the driver), and many distributed executors (the workers). Spark hides the complexity of distributing the tasks and data across the workers, and transparently handles fault tolerance. Nonetheless, the complexity of Spark steepens the learning curve, especially for entry-point Data Engineers [11]. Furthermore, Spark environment allows for several pitfalls, such as using too many collects, no clear understanding of caching and lazy evaluation, etc.

So far, a number of attempts have been made in order to simplify distributed computing. [8] present a web-based GUI to simplify MapReduce data processing, but supports only a small set of pre-determined actions/processing nodes. Also, [10] introduced an extension for RapidMiner[1], Radoop, which runs distributed processes on Hadoop via the RapidMiner UI. Another (streaming) extension to RapidMiner Studio was recently released by the INforE EU project[2], which supports code-free creation of optimized, cross-platform, streaming workflows running on one of the following stream processing frameworks: Apache Flink, Spark Structured Streaming or Kafka [6]. In addition, [9] present RheemStudio, a visual IDE that creates code-free workflows on (a subset of) Spark using RHEEM's ecosystem [4] to easily specify cross-platform data analytic tasks. In the same line of work, StreamSets Transformer[3] offers an UI based ETL pipeline where data transformations can be executed on Spark. Legacy code can be incorporated in StreamSets Transformers by writing custom PySpark or Scala pipelines. Similarly, KNIME[4], a visual programming environment, supports extension of workflows with Spark nodes. Spark code can be added in KNIME as a PySpark script node in the workflow. However, in both cases, the developer needs to understand the Spark API and semantics (RDDs, maps, reduces). At the moment, we still lack a simple, open-source graphical user interface that can be used out-of-the-box, to support Spark newcomers – developers with potentially no training and experience of Spark – to design, develop, and deploy complex workflows in Spark that go beyond standard ETL processes. We explicitly target a stand-alone tool that requires a very simple installation process (e.g., unzipping a file, or clicking an icon) and no servers/spark clusters, so that it can be used from novice users. Such a tool will simplify Spark, lowering the learning curve and initial cost for testing out integration and use of Spark in mission-critical processes.

This work introduces *Easy Spark*, a Graphical User Interface (GUI) for guiding the developer and flattening out Spark's learning curve. Instead of having to write code, the user designs and implements her big data application by designing a Directed Acyclic Graph (DAG), inserting nodes, specifying the input and output of each node, configuring the nodes, and linking them to other nodes. Upon completion of the workflow, the tool translates the model to executable Spark code, which can be submitted for execution to a cluster, executed locally, or even saved for future use. Beyond hiding the complexity of Spark by abstracting the job to the natural DAG model of operators, the GUI itself prevents bugs, e.g., by showing intermediary results to the developer, and by restricting the developer to a model and to specific method signatures that lend themselves to parallelism, *yet without requiring the introduction of Spark-specific concepts like Maps and Reduces. Easy Spark* can also serve more advanced requirements, guiding experienced developers that do not know Spark to write and to integrate code.

The user group of *Easy Spark* is: (a) Spark newcomers and students that want to quickly test out Spark, get a first introduction to Spark's basic ideas and capabilities, and construct a rapid prototype/proof of concept, (b) data scientists and domain scientists that are now hitting the limits of centralized computing, e.g., with python, but do not have the formal training or extensive programming experience to start with Spark, and, finally, (c) educators and researchers that need novel methods to introduce, teach, and

[1] https://rapidminer.com/

[2] https://www.infore-project.eu/

[3] https://streamsets.com/products/dataops-platform/transformer-etl/

[4] https://www.knime.com

advertise Spark and similar Big Data frameworks. It is planned that *Easy Spark* will be integrated in the syllabi of two courses this year, in two different universities/different professors, and it will be released as open-source after the conference.

2 RELATED WORK AND BACKGROUND

2.1 A MapReduce/Spark primer

Apache Spark [14] and Hadoop MapReduce [5] are the two most popular open-source frameworks to date for large scale data processing on commodity hardware.

MapReduce breaks a job to multiple tasks and assigns the tasks to the available workers. The programming API of MapReduce is concentrated on the implementation of two (types of) methods, mappers and reducers. Mappers take a pair (originating from a file) as input and produce a set of intermediate key/value pairs. Typical uses of mappers are filtering and transformations on the input data. The results of the mappers are typically pushed to the reducers. Each reducer instance (running the user-defined reducer function) accepts a key and a list of values for that particular key. Reducers typically serve the purpose of aggregating all data for each key.

Even though MapReduce is generally recognized as a highly effective and efficient tool for Big Data analysis [13], it comes with several limitations. During applications such as machine learning and graph analytics, data needs to pass from several processing iterations, i.e., a sequence of different jobs. MapReduces reads its input data from secondary storage (typically network drives), processes it, and writes it back for every job, posing a significant I/O overhead. Furthermore, expressing of complex programs with a series of maps and reduces proves to be cumbersome. Spark offers a solution to these problems, by the use of Resilient Distributed Datasets (RDDs), and by an expansion of the programming model to enable representing more complex data flows that may consist of multiple stages, namely Directed Acyclic Graphs (DAGs) [12]. RDDs are read-only and can be kept in memory, enabling algorithms to iterate over the same data many times efficiently. RDDs support two kinds of operations organized as a DAG: transformations and actions. Transformations are applied on one or more RDDs and return a new RDD. Examples of transformations are map, flatMap, filter, join, and union. Actions operate on an RDD and return back a non-RDD answer, e.g., a count, or a conversion of the last RDD to an array list. Due to the lazy-evaluation nature of Spark, transformations are executed only when their result needs to be processed by an action. Placement of actions in the Spark code is critical for the performance of the code – invoking unnecessary actions may nullify the benefits of distributed computation. A single Spark DAG may involve a number of transformations and actions, which are completed in a single run and can be optimized by the Spark execution engine. As a result, Spark is generally much faster on non-trivial jobs compared to Hadoop MapReduce [3, 13].

The architecture of Spark is depicted in Fig. 1. A Spark deployment consists of a driver program (SparkContext), many executors (workers), and a cluster manager. The driver program serves as the main program and is responsible for the entire execution of the job. The SparkContext object connects to the cluster manager, which sends tasks to the executors.

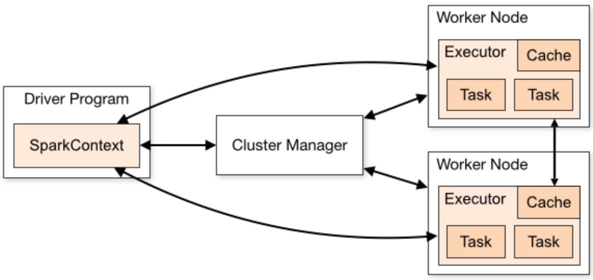


Figure 1: Spark Architecture (figure taken from [1])

2.2 Related work

Even though MapReduce simplifies the complexity of programming for distributed computing, it departs from the traditional programming paradigms, imposing a steep learning curve to programmers. [8] developed a GUI in which users can design their MapReduce workflow intuitively, without writing any code and/or installing Hadoop locally. Users are only required to know how to translate their tasks into target-value-action (TVA) tuples, which reflect data processing into mappers and reducers. Users pick objects as targets, whereas action filters or processes the values with the same target. For example, to implement a word-count code (i.e., count the frequency of each word in a text file), we can identify the following TVA values: each word is a target, *1* is a value, and *sum* is the desired action. The offered GUI offers a predefined list of operations, such as merge, sum, count, and multiply. However, the proposed GUI is inflexible in terms of input data formats, and cannot support arbitrary code.

RapidMiner Studio is an extendable and popular open-source user-interface for data analytics. Radoop is an extension for RapidMiner Studio and supports distributed computation on top of Hadoop [10]. Furthermore, it supports integration with PySpark and SparkR scripts [2]. Nonetheless, the user is still required to know Spark's API to include custom code for execution on top of Spark's distributed environment. Another more recent extension to RapidMiner Studio is [6], which supports code-free creation of optimized, cross-platform, streaming workflows over various clusters, including Spark. To the best of our knowledge, both Radoop and [6] do not include data information available on the operator level, whereas *Easy Spark* tries to reduce common errors by explicitly guiding the user, i.e., extracting data output types at individual nodes, and showing sample intermediate results when constructing the DAG.

StreamSets Transformer is a modern ETL pipelines engine for building data transformation, stream processing, and machine learning operations. It offers an easy drag-and-drop web-based UI in which users can create pipelines that are executed on Spark. It is possible to write custom Pyspark and Scala code as part of the user's data pipeline. However, the user should be familiar to Spark's API to include custom Spark code into their pipeline[5].

KNIME [6] is an open-source platform for drag-and-drop data analysis, visualization, machine learning, statistics, and ETL. KNIME allows the user to create workflows via their GUI without, or with only minimal programming. KNIME consists of an extension[7] that creates Spark nodes, which can execute Spark SQL queries, create Spark MLlib models and allow for data blending and manipulation. Spark Streaming and Spark GraphX are not integrated in the extension. The extension consists of a PySpark

[5] https://streamsets.com/documentation/transformer/latest/help/index.html
[6] https://www.knime.com
[7] https://www.knime.com/knime-extension-for-apache-spark

script node, where users can add their own code. However, similar to StreamSets and RapidMiner Studio, KNIME requires from the user to be familiar with the Spark API.

Lastly, RheemStudio [9] is a visual IDE on top of the open-source platform system RHEEM [4], which enables data processing over multiple data processing platforms, including Spark. RheemStudio helps developers to build applications easily and intuitively. It allows for a drag-and-drop generated RHEEM plan where users can drag-and-drop operators from the RHEEM operators panel to the drawing surface to construct a RHEEM plan. It is then shown to the user how the previously presented RHEEM plan can be specified into RHEEMLatin – the declarative language for RHEEM. Next, the user is invited to select one of the operators to revise and is asked to inject her own logics (couple of lines of code) which is then checked to be syntactically correct. Furthermore, RheemStudio consists of both a dashboard for displaying RHEEM's internals in detail, which allows for immediate interaction with the developer, and a monitor for keeping track of the status of the data analytic tasks. Compared to RheemStudio, *Easy Spark* focuses on simplicity and more guidance for the novice user. It displays intermediary results to the user which are useful for debugging, and it does not require learning an additional language like RHEEMLatin, since it does not need to integrate with multiple data processing languages. This focus to simplicity makes *Easy Spark* an ideal entry point to Spark, for non-Spark programmers.

Our contribution improves the state-of-the-art in multiple directions. First, it supports newcomers to start writing Spark without previous Spark knowledge since it avoids using the Spark semantics (RDDs, maps, and reduces). Instead it uses constructs that are identical, or very similar to the standard programming constructs of traditional languages (python and java). It also supports coding arbitrary code, which is useful for integrating legacy code, and it provides guidance to the user during the design of the workflow, e.g., by providing the expected structure (data format) and sample answers of each intermediary operation during the design of the workflow. This enables the developer to avoid pitfalls (e.g., calling too many collects), and quickly identify bugs. Finally, it can function as a stand-alone tool, not requiring complex installation and maintenance of large clusters.

3 EASY SPARK

Recall that the target group of *Easy Spark* includes a diverse set: Spark newcomers, CS and non-CS students, educators, researchers, and professional data/domain scientists. Naturally, each of these categories comes with different levels of expertise, experience, and problems of different complexity. To support all of them, we need a powerful and intuitive GUI where users can visually control the flow of the data. In *Easy Spark*, every operation on the data is represented by a node, and the user can connect nodes to form a computation path, which is a Directed Acyclic Graph (DAG) of computation nodes and edges that represent the flow of data between nodes. We start by providing a high-level overview of *Easy Spark*, with a special emphasis on how it guides the developer and prevents traditional errors. We then present a detailed discussion of the offered functionality.

3.1 A high-level overview of *Easy Spark*

Easy Spark contains two types of buttons: (a) nodes of different types, and (b) configuration buttons. Figure 2 depicts the currently supported DAG node types and configuration buttons.

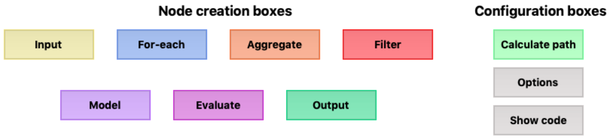


Figure 2: Boxes that create and configure nodes

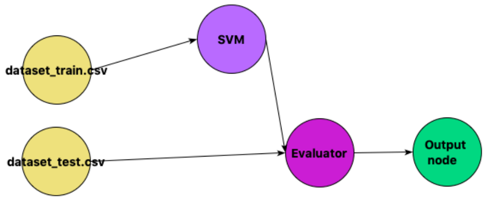


Figure 3: Example graph

Node types. *Input* and *Output* are for choosing the data input and results output files, and configuring how these should be read/written (e.g., format, delimiters). *For-each*, *Filter*, and *Aggregate*, correspond to the Spark functionalities of (flat-)map, filter, and reduce/reduceByKey respectively. *Model* and *Evaluate* allow the user to train an ML model, e.g., an SVM, and to use it for classification/predictions. Clicking on any of the above buttons adds a node with the matching color in the DAG, and allows the user to configure it (e.g., copy-paste legacy code into an editor, or configure the filter predicates). Section 3.3 discusses the precise meaning and configuration parameters of these node types.

Configuration buttons. Button *Options* allows the user to re-open the configuration panel on the selected node. *Calculate Path* executes the DAG on the full dataset – or submits it for execution to a cluster – and writes the results to the output node, whereas *Show code* depicts the generated Spark Code. The precise functionality of these buttons will be detailed in Section 3.4.

Example. Fig. 3. depicts an example DAG that executes a fairly common ML task: training a ML model on a part of the dataset, and testing the produced model on the remaining part. The user first configures the two input nodes, by selecting the correct file names, and then adds a model training node (in this case, selected to generate an SVM). The output of the model node is the model itself, which is then passed to an evaluator node, together with the testing partition of the dataset. The output of the evaluator is finally saved into the output node.

Notice that the described workflow does not include any Spark-related concepts. We will see soon that the Spark fundamentals remain hidden from the user.

3.2 Guiding the user

Different mechanisms are in place to guide the user through the DAG design.

- *Showing sample intermediary results.* When adding a node, the developer is able to immediately preview a small number of input and output results of that node (see Fig. 4 for an example). The preview is computed locally on a small part of the data such that results can be shown with zero waiting time.
- *Identifying and naming the attributes, and propagating the data formats and structures.* The developer configures a structure and format for the data input when configuring the input nodes (part of this is inferred if the data files contain headers). This information is propagated in the following nodes, i.e., the user can see and use the attribute names. When the code of an intermediary

node (e.g., a *For-each* node) modifies the data format, the developer is supported to update the data format accordingly.

• *Disabling buttons that are incompatible with the current state.* The current state and current selection determines the buttons that are enabled and/or disabled. For example, when an output node is selected, only the *Calculate path* button is enabled.

• *Hiding the Spark semantics from the developer.* Notice that the used operations are not specific to Spark. E.g., the developer does not need to understand map, reduce, and RDDs. She does not need to write Spark boilerplate code, or directly submit the code for execution on a server. All semantics of *Easy Spark* can be easily understood by an everyday Python developer/data scientist.

• *Preventing common errors.* Besides hiding the complexity of parallelism which is frequently a source of error, *Easy Spark* supports the developer on using named and typed data structures, and shows result samples at each intermediary node. Therefore the developer can quickly detect most types of errors.

3.3 Node types

Nodes have to represent operations that are both familiar to the user and useful in the context of data processing. To serve this purpose, *Easy Spark* comes with a core set of node types, and allows for an easy extension by implementing additional nodes. The current node types are:

• *Input.* An input node serves as a data source. On creation of an input node, the user chooses the data source, and the input node handles the creation of the corresponding RDD, and parallelizes the data to the available worker machines. *Easy Spark* also prompts the user whether or not the data source contains header data, and if so, the header is parsed and propagated to subsequent nodes on the computation path. If no header is in the data, the user is prompted to supply the related information.

• *For-each.* The for-each node allows the developer to enter code that would normally be put in the body of a for-loop over all records. The developer is prompted with a box for specifying the desired behaviour of the node, i.e., to include the code that needs to be executed. The node itself then chooses what Spark code can be executed in order to replicate this behaviour, either through `map` or `flatMap` functions.

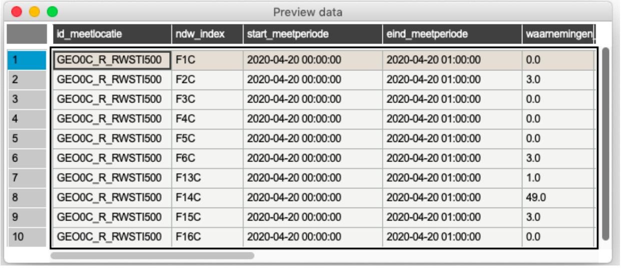
• *Aggregate.* The aggregation node is responsible for aggregating data over different groups in the data. The user is prompted to supply the level of aggregation and the type of aggregation, and the node itself generates key-value pairs and employs a `reduce` or `reduceByKey` function in order to aggregate the data over the given level of aggregation.

• *Filter.* The filter node handles the filtering of unwanted data. The user can supply the tool with a boolean expression (or code that will return a boolean expression) being evaluated for every row in the data. This way, data is excluded from the dataset.

• *Model.* The model node is the gateway to the MLLib library in Spark. The model node prompts the user to supply the split between features and labels, and trains a Machine Learning (ML) model on the given data. The complexity of training a ML model is concealed from the user, such that the user can intuitively create ML pipelines that run on Spark clusters.

• *Evaluate.* This node is responsible for evaluating the results of a ML model trained by a model node on data that comes from another node. This node supplies the user with the possibility of evaluating a model on a large dataset, since evaluation will be parallelized across nodes.

• *Output.* The output node executes each node within the path of the graph and writes the results to a text file from which the user picks the preferred location.



	id_meetlocatie	ndw_index	start_meetperiode	eind_meetperiode	waarnemingen
1	GEO0C_R_RWSTI500	F1C	2020-04-20 00:00:00	2020-04-20 01:00:00	0.0
2	GEO0C_R_RWSTI500	F2C	2020-04-20 00:00:00	2020-04-20 01:00:00	3.0
3	GEO0C_R_RWSTI500	F3C	2020-04-20 00:00:00	2020-04-20 01:00:00	0.0
4	GEO0C_R_RWSTI500	F4C	2020-04-20 00:00:00	2020-04-20 01:00:00	0.0
5	GEO0C_R_RWSTI500	F5C	2020-04-20 00:00:00	2020-04-20 01:00:00	0.0
6	GEO0C_R_RWSTI500	F6C	2020-04-20 00:00:00	2020-04-20 01:00:00	3.0
7	GEO0C_R_RWSTI500	F13C	2020-04-20 00:00:00	2020-04-20 01:00:00	1.0
8	GEO0C_R_RWSTI500	F14C	2020-04-20 00:00:00	2020-04-20 01:00:00	49.0
9	GEO0C_R_RWSTI500	F15C	2020-04-20 00:00:00	2020-04-20 01:00:00	3.0
10	GEO0C_R_RWSTI500	F16C	2020-04-20 00:00:00	2020-04-20 01:00:00	0.0

Figure 4: Preview of the data via the options box for the input node

3.4 Configuration buttons

This subsection lists the purpose of the configuration buttons.

• *Calculate path.* By clicking on this button, the data of the node that was previously selected is collected and written to disk. This serves as a clear endpoint of the calculations for the user, and supplies the tool with a clear target node for which we can apply the transformations defined by the nodes on the path from the data source nodes to the previously selected node that activates the calculation. Note: it is possible to press this button multiple times, for different nodes (and paths) in order to derive multiple outputs from the DAG.

• *Options.* The options button enables the developer to show additional information or set configuration parameters for the selected node. The options for all nodes are as follows:

– *Input* presents a preview of ten rows from the data source (see Fig. 4)

– *For-each* consists of a drop-down menu for the output type (one output or multiple outputs), a drop-down menu for the level of the for-loop based on the (provided) header data, an entry box where the user can enter the code that needs to be execute for every level and an entry box with the structure of the output.

– *Aggregate* shows a drop-down menu for the aggregation function (sum or count), a drop-down menu for the key (one of the headers) to aggregate on and an entry box to provide the structure of the output.

– *Filter* includes a drop-down menu for the filter level (entire row or attribute in a row) and an entry box where the filter condition should be entered.

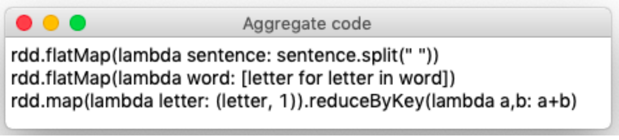
– *Model* has a drop-down menu for the type of model (SVM) and two entry boxes where the statement that retrieves the label and features from each data entry should be entered.

– *Evaluate* has two drop-down menus from which the model node and data node should be selected and two entry boxes to enter statements that retrieve the label and features.

• *Show code.* By clicking this button, the user can see the generated Spark code on the selected node (see, e.g., Figure 5). If no node is selected, the user gets the generated Spark code for the whole program.

4 CASE STUDIES

We now discuss three case studies that are supported by the tool. The goal of our discussion is to illustrate the simplicity and expressive power of the tool, and to demonstrate the ease of use with which Spark architectures can be designed.



Aggregate code

```
rdd.flatMap(lambda sentence: sentence.split(" "))
rdd.flatMap(lambda word: [letter for letter in word])
rdd.map(lambda letter: (letter, 1)).reduceByKey(lambda a,b: a+b)
```

Figure 5: Generated Spark code based on the aggregation node in LetterCount (subsection 4.1)

Algorithm 1: LetterCount.

```
Input: Textfile f, dictionary Counts;
for line in f do
    for word in line do
        for letter in word do
            if letter in Counts then
                Counts[letter] += 1;
            else
                Counts[letter] = 1;
            end
        end
    end
end
```

4.1 Applying Easy Spark to LetterCount

We start by parallelizing a workflow that counts the number of appearances of each letter in a large text file. The pseudocode of the algorithm (algorithm 1) involves a triple nested loop – first the text is broken into lines, then each line is broken into words, and then each word is broken into letters.

To implement this using *Easy Spark*, we need three nodes: a (yellow) Input node which contains the data source, an (orange) Aggregation node to count, and a (green) Output node. Figure 6 depicts the drawn graph in the GUI for LetterCount.

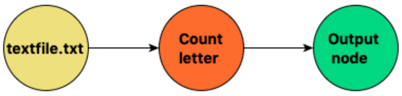
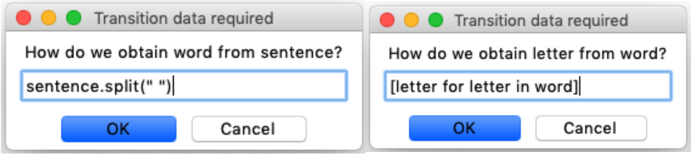


Figure 6: Drawn graph for LetterCount.

When creating the Input node, the user is asked to provide the header (since a plain text file does not contain a header). For the sake of this example, we choose the header to be of the following structure: sentence - word - letter. This means that the expected output of the input node we just added will contain three different representations of the data. Notice that the names are arbitrarily chosen by the user, but give a reference point for the next nodes.

Next, we create the Aggregation node and set the right options: aggregation type is set to count and aggregation key is set to letter. Before we can calculate the computation path, the GUI needs to know how to get from sentence to word, and from word to letter. This information is provided by the user through two pop-up windows (Figure 7). Lastly, we add the Output node from which we can initiate code generation/execution.



Transition data required

How do we obtain word from sentence?

sentence.split(" ")

Transition data required

How do we obtain letter from word?

[letter for letter in word]

(a) Sentence to word. **(b) Word to letter.**

Figure 7: Required transition data for LetterCount.

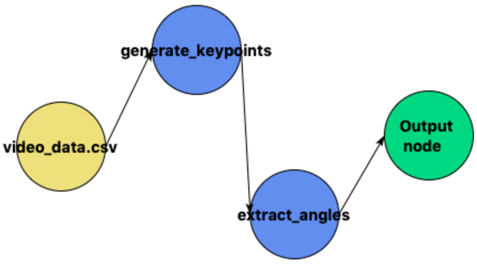


Figure 8: DAG Corresponding to the video analysis

4.2 Distributed Video Analysis

In this use case we will apply our tool to parallelize the data of individual frames in an input video, and perform necessary computations on each frame. In particular, we will apply an ML model that extracts keypoint predictions from a pre-trained network [7], and use these predictions to extract relevant angles between keypoints for motion analysis. Since currently our Input node does not natively support breaking of videos to frames, we use a third-party tool to extract and save all frames from the video as a csv file, and provide this as an input. After importing the data, we use a For-each node to extract the keypoints by running the pre-trained network, followed by another For-each node that extracts the relevant information from the keypoints. The internal code of the two For-each nodes is copied from the centralized implementation. The final DAG appears in Fig. 8.

4.3 Engineering and evaluating a Machine Learning pipeline

Our next use case considers a standard problem when engineering an ML pipeline: different models (or possibly the same model with different hyperparameters) need to be trained and tested with the training and the holdout datasets, in order to compute the accuracy of each model and choose the best one. Figure 9 presents a simple DAG that trains and evaluates an SVM model. The first step is to configure the data input – the training and the hold-out data set. Both input files are passed through a For-each node for pre-processing. The training data is used for building a model (an SVM), which is then passed into an Evaluate node together with the test data. To count the misclassifications we add a Filter node that is responsible for filtering out all correctly-classified cases, followed by an Aggregation node which will count the number of results. Observe that the DAG is sufficient to obtain a clear idea of the data flow, and that we can represent complex operations on the data by a relatively small amount of nodes, i.e., the nodes have a high expressive power.

5 CURRENT AND FUTURE WORK

Easy Spark is under development. It can already generate code for complex workflows and submit it for execution, but it still misses some of the envisioned functionalities. In this section, we discuss our current and planned work and the involved challenges.

- *Adding new node types.* Our effort with the current version of *Easy Spark* was to provide a zero-learning-curve proof-of-concept tool that can be used in teaching, and as the entry point

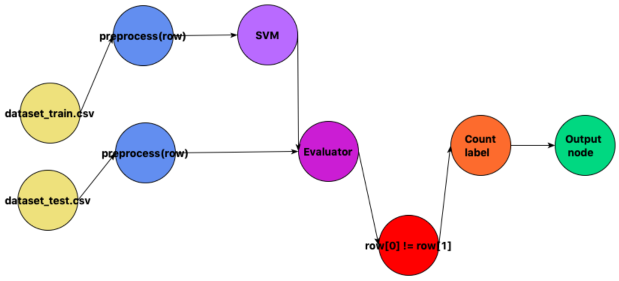


Figure 9: DAG Corresponding to the ML pipeline

of a newcomer in Spark. It was out of our scope to provide a complete visual programming language equivalent to the Spark capabilities. Being convinced with the usefulness of the tool, we are now extending it with more node types to cover frequently-used functionality, e.g., Spark SQL, GraphX, Spark Streaming, and the full MLlib library. It is fairly straightforward to add more node types. We are also redesigning the user interface for helping the users to find the desired node types quickly, e.g., have an ML tab, where all the nodes related to MLlib will be added.

• *Improved support for legacy code. Easy Spark* includes support for including hand-written legacy code, e.g., when writing code for *For-each* nodes. Currently, *Aggregate* nodes enable selecting from a pre-defined set of functions, such as sum and count, but it is not challenging to enable arbitrary code. Support for legacy code can be further improved by recognizing the signature of the functions, and automatically extracting the output data types, instead of asking the user to update the data types manually whenever these change. Notice that this functionality is feasible, as it is already present in different IDE tools.

• *Improved error detection.* Automatic extraction of function signatures can be leveraged to identify and present errors to the user that relate to the data types and formats, e.g., the output of one node does not agree with the expected input of the next node. Again, this is supported by most modern IDEs.

• *Showing sample intermediate results.* Presentation of sample intermediate results (a sample of the output of each node) during design time is often very useful, for getting a rough idea on what is happening up until that point, and for keeping track of the data format that the next node will receive. Our current implementation presents sample intermediary results (the first 10 results) only at some nodes. Notice, however, that some node types, e.g., Filter, For-each, can change this number. The number of sample results shown at each node should be adapted in order to still get meaningful samples throughout the workflow. Furthermore, in some cases, the distribution of the sample results is also important to get a meaningful sample output, e.g., training of a binary SVM classifier requires representative samples from both classes. Starting with a huge number of samples to ensure that all nodes will have an output is also not an option, since this will decrement the performance of *Easy Spark*. We are now developing methods that adaptively choose the samples at each node, in order to derive meaningful sample results at all nodes.

• *Supporting more data input formats and streaming data.* This will further reduce the complexity of loading the data, and offer better representations of the data.

• *Improved support for Spark-related errors.* Some common Spark errors, e.g., OutOfMemory exceptions, can (mostly) be addressed with a few standard steps, e.g., increase the RAM available to the executors or driver, avoid collects, or increase the number of partitions. In the future we will detect these exceptions and propose these standard steps to the user.

• *Integrating the Spark UI.* It will be useful – and with educational value – to integrate Spark's Web UI in *Easy Spark* in order to allow the user to monitor the status of the developed Spark application, resource consumption, Spark cluster, and Spark configurations.

6 CONCLUSIONS

In this paper we proposed *Easy Spark*, a Graphical User Interface for easily designing Spark architectures for Big Data Engineering. The GUI enables the developer to design complex Spark DAGs with arbitrary functionality, by masking Spark constructs and concepts behind traditional programming constructs that any trained data scientist or computer scientist is able to understand, use, and configure. We examined three use cases and showed how *Easy Spark* can be used to generate executable Spark code. We also discussed the mechanisms that are currently implemented in *Easy Spark* to guide the user and prevent bugs, and elaborated on our current and future work to extend and improve it.

7 ACKNOWLEDGEMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041).

REFERENCES

[1] 2020. Spark Cluster Mode Overview. https://spark.apache.org/docs/latest/cluster-overview.html.

[2] 2021. RapidMiner Radoop Feature List. https://rapidminer.com/products/radoop/feature-list/. [Online; accessed 27-January-2021].

[3] Bansod A. 2015. Efficient Big Data Analysis with Apache Spark in HDFS. *International Journal of Engineering and Advanced Technology* 4, 6 (2015).

[4] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing: May the Big Data Be with You! *Proc. VLDB Endow.* 11, 11 (July 2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

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

[6] Nikos Giatrakos, David Arnu, T. Bitsakis, Antonios Deligiannakis, M. Garofalakis, R. Klinkenberg, Aris Konidaris, Antonis Kontaxakis, Y. Kotidis, Vasilis Samoladas, A. Simitsis, George Stamatakis, F. Temme, Mate Torok, Edwin Yaqub, Arnau Montagud, M. Leon, Holger Arndt, and Stefan Burkard. 2020. INforE: Interactive Cross-platform Analytics for Everyone. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (2020).

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2018. Mask R-CNN. arXiv:cs.CV/1703.06870

[8] C. S. Liao, J. M. Shih, and R. S. Chang. 2013. Simplifying MapReduce data processing. *International Journal of Computational Science and Engineering* 8 (2013). https://doi.org/10.1504/ijcse.2013.055353

[9] Ji Lucas, Yasser Idris, Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, and S. Chawla. 2018. RheemStudio: Cross-Platform Data Analytics Made Easy. *2018 IEEE 34th International Conference on Data Engineering* (2018), 1573–1576.

[10] Zoltán Prekopcsák, Gabor Makrai, T. Henk, and Csaba Gáspár-Papanek. 2011. Radoop: Analyzing Big Data with RapidMiner and Hadoop.

[11] S. Salloum, R. Dautov, P. X. Chen, X.and Peng, and J. Z. Huang. 2016. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* (2016).

[12] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2110–2121. https://doi.org/10.14778/2831360.2831365

[13] A. Verma, A. H. Mansuri, and N. Jain. 2016. Big data management processing with Hadoop MapReduce and spark technology: A comparison. *2016 Symposium on Colossal Data Analysis and Networking (CDAN)* (2016), 1–4.

[14] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664