Visually Programming Dataflows for Distributed Data Analytics

Lauritz Thamsen, Thomas Renner, Marvin Byfeld, Markus Paeschke, Daniel Schröder, Felix Böhm
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

***Abstract*—Distributed dataflow systems like Spark and Flink allow to analyze large datasets using clusters of computers. These frameworks provide automatic program parallelization and manage distributed workers, including worker failures. Moreover, they provide high-level programming abstractions and execute programs efficiently. Yet, the programming abstractions remain textual while the dataflow model is essentially a graph of transformations. Thus, there is a mismatch between the presented abstraction and the underlying model here. One can also argue that developing dataflow programs with these textual abstractions requires needless amounts of coding and coding skills. A dedicated programming environment could instead allow constructing dataflow programs more interactively and visually. In this paper, we therefore investigate how visual programming can make the development of parallel dataflow programs more accessible. In particular, we built a prototypical visual programming environment for Flink, which we call *Flision*. Flision provides a graphical user interface for creating dataflow programs, a code generation engine that generates code for Flink, and seamless deployment to a connected cluster. Users of this environment can effectively create jobs by dragging, dropping, and visually connecting operator components. To evaluate the applicability of this approach, we interviewed ten potential users. Our impressions from this qualitative user testing strengthened our believe that visual programming can be a valuable tool for users of scalable data analysis tools.**

***Index Terms*—Scalable Data Analytics, Distributed Dataflows, Visual Programming, End-user Development**

I. Introduction

Distributed dataflow systems such as MapReduce [1], Spark [2], and Flink [3] enable users to process large datasets. These frameworks allow to create programs from sequential building blocks, which are then executed massively parallel. In particular, users construct programs using a set of pre-defined operators. They configure these operators, which includes providing user-defined functions (UDFs) for second-order functions like Map and Reduce, and connect them to form graphs of data transformations. The frameworks automatically parallelize, schedule, and deploy these jobs on to cluster nodes. Many of these analytics frameworks also provide higher-level declarative programming abstractions, including for example SQL-like query languages [4], [5], [6], [7] as well as APIs for specific use cases such as graph processing [8] or machine learning [9], [10]. Some of these systems also automatically optimize programs [4], [11], [12] and provide mechanisms for fault tolerance. MapReduce, for example, exchanges data through a distributed file system with built-in replication such as the Google File System (GFS) [13], while Spark uses linage for the fault tolerance of its Resilient Distributed Datasets (RDDs) [14]. In summary, these frameworks save a lot of effort for users. Users can concentrate on their analysis tasks and leave efficient as well as fault-tolerant distributed execution to the frameworks. However, despite all these steps towards eased distributed analytics of large datasets, programming abstractions are usually still on the level of textual programming languages, specifically in the form of APIs or embedded languages [15], even though the dataflow model is graph-based. This results in a representation mismatch between source code and mental models. Programmers effectively glue operators together using method calls. The programs also mix data processing with repeated setup code. Seeing a clear representation of the actual dataflow graph instead, including what components are available as building blocks and how different components can be connected, might be helpful to users, especially users new to distributed dataflow systems. Furthermore, some potential users of data analysis might also lack knowledge in language concepts, syntax, and libraries. Enabling these users to express their own data analysis programs would be highly beneficial. After all, such users often understand the domain- and business-specific questions best. Moreover, the analytics frameworks do provide additional tools for job submission and execution monitoring, yet these are separate tools from the editors and Integrated Development Environments (IDEs) developers use.

For these reasons, we propose to use visual programming techniques and a dedicated programming system for distributed data analytics. In particular, we argue that a dedicated dataflow programming system should present users with an editable graph representation of their analysis job and should provide integrated job submission as well as monitoring facilities. Towards this goal, we built a Web-based prototype, which we call *Flision*. Flision is integrated with Flink and allows to create, submit, and monitor jobs. Users can create these jobs by dragging, dropping, connecting, configuring, and programming operators–all while viewing a graphical representation of the job graph. Flision also provides users with a view of what components are available and hints on how components can be connected. For second-order operations that need to be configured with UDFs, users can either load UDFs or develop them within Flision. Created dataflow jobs can then be exported, compiled, and executed on a connected cluster. In case jobs are executed directly from Flision, users can follow the execution through an integrated console view.

Outline. The remainder of this paper is organized as follows. Section II provides the necessary background on distributed dataflow systems. Section III first presents a motivating example and then derives requirements for a programming system. Section IV presents our solution, including our prototype. Section V describes user feedback to the prototype. Section VI discusses related work. Section VII concludes this paper.

II. Background

Distributed dataflow systems process data using parallel data transformations on clusters of connected worker nodes. A job graph constructed from data transformation tasks is shown in Figure 1. Tasks are versions of a set of pre-defined operators. The set includes, for example, Map, Reduce, and Join. Map and Reduce are second-order functions. These two operators are configured with UDFs that are called for either a single element or a group of elements. Specific variants of Map and Reduce are operators for filtering and aggregation. Other operators like Join or Cross can be used to combine multiple dataflows. This way multiple datasets can be merged using matching keys as join-predicate.

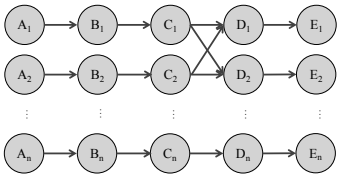


Fig. 1: A job graph with *n* data-parallel task instances of five subsequent task.

The tasks of dataflow job graphs are executed in parallel on many worker nodes. In the job graph in Figure 1, for example, each task has *n* data-parallel instances. These data-parallel task instances typically run on shared-nothing commodity servers, which are connected by a datacenter network and provide execution slots. Execution slots represent computing capabilities. A node with eight hardware threads could, for example, expose eight slots for task execution. Often, execution slots represent equal amounts of the resources of a worker machine in homogenous cluster setups.

Parallel instances process partitions of the data. Partitions are created while reading distributed input files or by shuffling the dataset between subsequent operators. Parallel instances of source operators usually create partitions by reading parts of the input in parallel from a distributed file system. Shuffling is all-to-all communication between all data-parallel task instances of subsequent tasks: elements with the same key are moved to the same instances. This is necessary as some operators require specific partitions of the input data. In cases, in which the data is not already partitioned correctly, data is therefore shuffled before such operators. Operators require, for example, elements of the same group or with identical keys to be available at the same parallel instance to be able to correctly perform their data transformations. The job graph in Figure 1 shows this kind of communication between the instances of task *C* and *D*. Two other important data exchange patterns are all-to-one and one-to-one communication. One-to-one communication forwards elements between subsequently connected task instances without a repartitioning of the data. This communication pattern is visible between all other operators in the job graph shown in Figure 1. One-to-one communication is applicable when no particular partitions are required by the receiving tasks or when the necessary partitions have been established before. All-to-one communication is typically used for global aggregates.

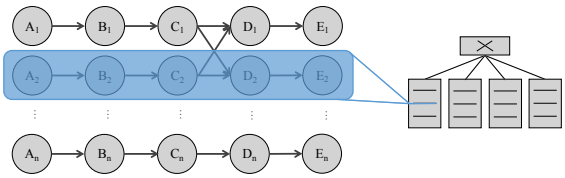


Fig. 2: Subsequent task instances scheduled onto the execution slots of a worker node.

Figure 2 shows a chain of subsequent task instances that is scheduled onto a worker. Except for certain operators, chained tasks can also be executed in parallel, adding pipeline parallelism to the data parallelism of parallel task instances. However, as described above, certain operators require shuffling. That is, these operators require all elements with a certain key to be available before they can start outputting results. Such operators therefore synchronize all parallel threads of the dataflow. This is, for example, often necessary for operators like Reduce and Join. The worker nodes in Figure 2 provide three execution slots, yet multiple task instances can be scheduled and deployed onto one slot. Since each task instance usually maps to at least one thread, the operating system effectively schedules the task instances running on a worker.

III. Motivation

This section presents and discusses the code of an example dataflow program in its textual code representation. We use this example to motivate more graphical and interactive programming systems.

A. WordCount Example

An example of a Flink program is the WordCount application. This simple application counts the occurrences of each word within a text. Listing 1 shows the Java implementation of this application. The listing shows the `main` method of the central `WordCount` class. The code was taken from the examples of batch programs that come with Flink [1]. We changed it in two ways. First, we changed this application to always read input from and write outputs to a path instead of using built-in example data and stdout. Second, we stripped comments and some empty lines to make the listing more concise.

[1] Apache Flink batch examples in version 1.1.2, https://github.com/apache/flink/tree/master/flink-examples/flink-examples-batch/src/main, accessed 2016-10-01

```
1  public static void main(String[] args) throws
       Exception {
2
3    final ParameterTool params = ParameterTool.
       fromArgs(args);
4    final ExecutionEnvironment env =
       ExecutionEnvironment.getExecutionEnvironment();
5    env.getConfig().setGlobalJobParameters(params);
6
7    DataSet<String> text = env.readTextFile(params.get
       ("input"));;
8
9    DataSet<Tuple2<String, Integer>> wc =
10     text.flatMap(new Tokenizer()).
11         groupBy(0).sum(1);
12
13     wc.writeAsCsv(params.get("output"), "\n", " ");
14
15     env.execute("WordCount Example");
16 }
```

Listing 1: The main method of the *WordCount* application.

The main method starts with parsing the arguments and getting the execution environment. The `ExecutionEnvironment` returned can be a local machine or a Flink cluster, depending on the configuration. The actual dataflow then starts in Line 7 with reading in a text file. The method `readTextFile` yields a dataset of strings, containing the lines of the text file. This dataset is processed by a sequence of operators, from Line 9 to Line 11. First, each line of the text file is split into words using a user-defined FlatMap class called `Tokenizer`. The words are then grouped and summed up. The program ends with code writing out the results as CSV into a file, shown in Line 13 of Listing 1.

```
1  public static final class Tokenizer implements
       FlatMapFunction<String, Tuple2<String, Integer>>
        {
2
3    @Override
4    public void flatMap(String value, Collector<
     Tuple2<String, Integer>> out) {
5      String[] tokens = value.toLowerCase().split("
     \\W+");
6
7      for (String token : tokens) {
8        if (token.length() > 0) {
9          out.collect(new Tuple2<String, Integer>(
     token, 1));
10       }
11     }
12   }
13 }
```

Listing 2: The *Tokenizer* UDF class of *WordCount*.

A look at the implementation of the UDF class `Tokenizer`, helps to understand the arguments to the GroupBy and Sum operators. The `Tokenizer` class is an inner class of the class `WordCount` and shown in Listing 2. Each line of text file is passed to the `flatMap` method as the `value` parameter. First, all characters of the line are changed to lower case. Then, each line is split at whitespaces, yielding all individual words. Subsequently, empty words are filtered out before the result is collected. A `Tuple2` is a key-value pair. In this case the key is a word and the value is the number of occurrences. Before equal words are grouped, each word occurs exactly once, visible in Line 9 of Listing2. In Listing 1, the argument `0` passed to the `groupBy` operator refers to the first type of the `Tuple2`, which is the key and in this case the words. This is why the `groupBy` yields one group per word. The argument `1` passed to the `sum` operator then refers to the second type of the `Tuple2`, which is the value and in the case of this program the number of occurrences of each word. The `sum` operator thus sums up the number of occurrences of each word.

B. Discussion

Several drawbacks are the consequence of working with a textual code representation of dataflow programs such as previously shown for the Flink *WordCount* application:

1) *Representation mismatch*: A textual language is arguably not well suited to represent a graph. This is well visible for dataflow components with more than one input or output such as joins and forks.
2) *Operator set invisible*: Which pre-defined operators are available and can be used to extend the dataflow program is generally not visible. Users either have to consult the documentation or need to use code completion facilities of IDEs.
3) *UDF clutter*: UDFs can be defined inline using lambda functions or in additional classes, which can be inner classes or external class definition. Users thus essentially have the choice between cluttering the dataflow program with UDFS or having to explicitly scroll to referenced UDF class definitions.
4) *Repeated setup code*: Setup statements at the beginning–such as the ones for getting the execution environment–and submission statements at the end of the programs–such as the actual execution of the job–are repeated per job.
5) *Coding Environment*: The features of general purpose editors and IDEs focus on producing and compiling code. Important features include, for example, syntax highlighting and code completion facilities, yet does usually not include specific features for using distributed dataflows to analyze large datasets.

Given these five drawbacks, we see the following requirements for a dedicated visual dataflow programming environment:

- *Job graph view*: A visual representation of a job graph can provide better overview of the overall job structure. Besides clearly seeing which operator is connected to which as well as fork/join patterns, freely positionable components could also allow users to visually group components that are closely related.
- *Visible operator set*: A pane showing the set of pre-defined operators can help users learn which operators are available. Further, each of these operators could clearly show how many inputs are necessary.
- *Windowed operator configuration*: A collapsible view of operator configurations, including editing UDF code

in windows, provides a middle ground between inline functions and definition elsewhere. Having these windows pop up close to the operator they configure, but also freely positionable allows to mix overview with a view of particular details at one's discretion.

- *Abstracted setup code*: A visual environment can abstract setup code that is connecting the dataflow job to its execution environment using user interface elements like buttons or selection boxes. There can be for example a button that executes the job in the configured execution environment.
- *Dedicated integrated environment*: A dedicated programming environment for dataflow jobs can provide job submission and monitoring facilities otherwise found in additional tools.

These features can be implemented using a visual programming interface for creating programs for distributed dataflow frameworks.

IV. Solution

This section presents our idea and describes the features as well as implementation of our prototype.

A. Idea

With visual programming languages, users manipulate programs graphically instead of textually. Thereby, the program construction becomes more visual and more interactive. In fact, visual programming is known to improve programming efficiency for a broad range of users with different levels of understanding of computer science and programming concepts [16]. It is language independent and instead uses a intuitive graphical representation [17]. Consequently, it has often been used for rapid prototyping and for end-user development [16], [18], [19].

The idea we have for a visual programming system for distributed data analytics resolves around users being able to visually create programs by interactively developing job graphs. This happens via drag and drop, connecting elements graphically, and using interface elements for actions as well as configuration. Such actions and configuration options as well as usable components are directly visible. Moreover, users do not need to leave the environment to code new UDFs or to run their jobs.

An interesting aspect of such an environment is that users can freely position components on an infinite workspace. Following research on freely positionable code elements for development [20] and debugging [21], this increases code readability and understanding significantly. Users can use positions to express relations between components. For example, they can use component positions to highlight specific transformations in a longer pipelines.

Following this vision, we decided to investigate using visual programming techniques for distributed dataflow programs. In particular, we built a prototypical visual programming system for Flink.

B. Prototype

We implemented a prototype of a visual programming environment which we call Flision. With Flision, users can create, modify, and run Flink jobs. The prototype provides a Web-based user interface for these tasks, shown in Figure 3.

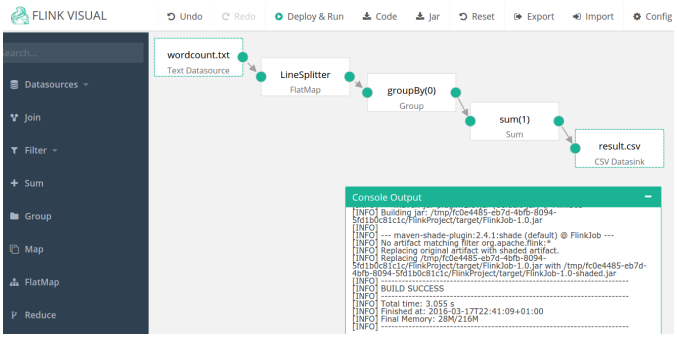


Fig. 3: The user interface of the Flision programming environment, showing the WordCount application.

The interface is designed to increase both readability and comprehensibility. In particular, users are provided with a graphical representation of the dataflow graph. That is, on Flision's main canvas, users can modify the dataflow graph to be executed as a Flink job. In particular, Users can add components such as data sources and data sinks as well as Map, Filter, Reduce, Join, and GroupBy operator via drag and drop. The set of pre-defined components is visible on the left side of the environment, as shown in Figure 3. Furthermore, users can visually connect these components. This defines the dataflow. Each component has a fixed number of inputs, so users can not create connections that would result in an invalid Flink program such as adding more than two inputs to a join component.

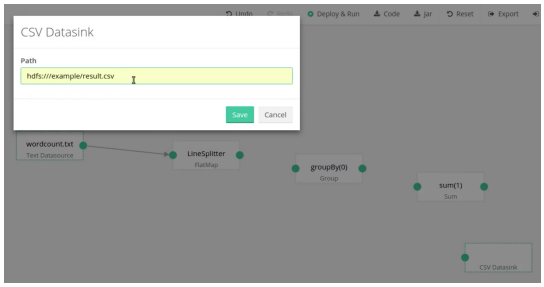


Fig. 4: Configuration of a data sink component in Flision.

Flision also allows to configure the properties of the operator. For example, they can configure the properties that need to match if two datasets are to be joined. Another example is shown in Figure 4. For a sink, users can specify the path to which the operator is supposed to store the dataflow results. As visible in the figure, such configuration is done in overlaying windows in Flision. Moreover, users can program UDFs for second-order function operators like Map and Reduce. The editor for this is shown in Figure 5. It also opens itself in

a window and overlays the workspace. The editor provides standard features such as syntax highlighting.

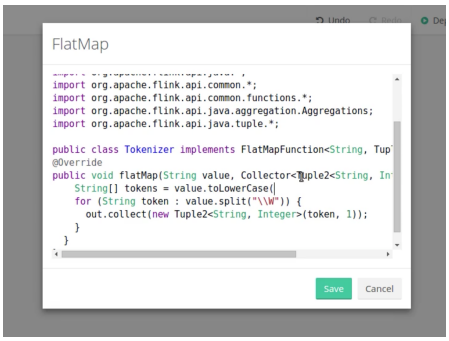


Fig. 5: Developing a custom FlatMap transformation in Flision.

Users can directly execute the job on a connected cluster or can export it. Export options are a JAR with compiled classes or a ZIP with source code. Overall, Flision gives users three options:

- execute the job,
- download a ZIP file containing the generated source code,
- or download a precompiled JAR file of the current job.

When executing the job, errors or misconfigurations will also be displayed. Flision shows the console output in the lower right corner of the user interface.

C. Implementation

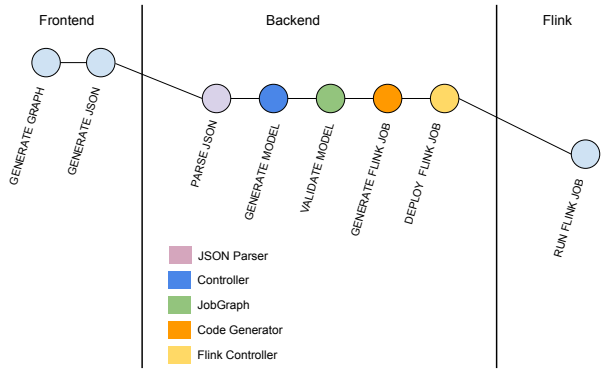


Fig. 6: Steps a dataflow graph takes before running as Flink job on the cluster.

Figure 6 shows the different stages that a job graph passes through when the users triggers its execution in Flision. This process starts in the frontend, with the job graph created by a user. The frontend generates a JSON representation of this job graph and adds the Apache Flink environment variables. It then sends this object via a REST API to the backend. Flision's backend receives this representation, parses it, and creates a corresponding internal job graph model. This model contains the boilerplate as well as the user-specific code for each operator. First, this model is validated for syntactical correctness. Then, this model is used to generate the Java code for the job to be able run the program on a Flink cluster. Finally, it starts the Maven build process and triggers the remote execution of the output JAR on the cluster.

V. User Tests

This section describes the user tests we did to evaluate the acceptance and usability of our prototype. First, we give information about the participants including their background and experience in programming and data analytics. Afterwards, we describe the experimental setup and observations we made during the testing. A questionnaire and a discussion complete this section.

A. Participants

The user tests were done with ten participants that were between 20 and 43 years old. The average age was 29.7 years. Each participant works in the field of or studies computer science. One participant has a diploma, one a master degree, seven a bachelor degree, and one participant holds no degree yet. The average participant of our user study has 7.85 years of programming experience. However, the range of experience varied significantly as the participants had between 3.5 and 20 years of experience. All participants said they have good knowledge in at least one of the programming languages Java, Scala, or Python. Figure 7 illustrates how the participants rate their own Java programming skills on a 1 - 10 scale, with 1 being the lowest and 10 the highest value. In addition, 80 percent of the participants had experience with the MapReduce programming paradigm. As shown in Figure 8 most participants rate their knowledge of MapReduce between medium and low.

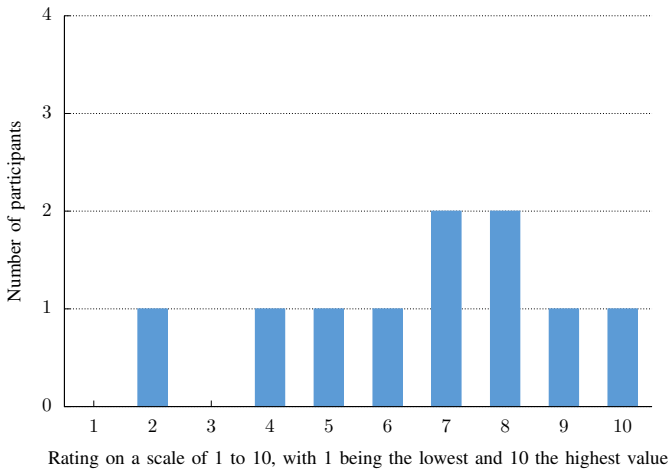


Fig. 7: Results of the question: How do you rate your Java programming skills on a 1 - 10 scale?

B. Experimental Setup Description

The experiment was done in private sessions with each participant. At the beginning, an introduction to the prototype was given. Afterwards, the participants were asked to implemented

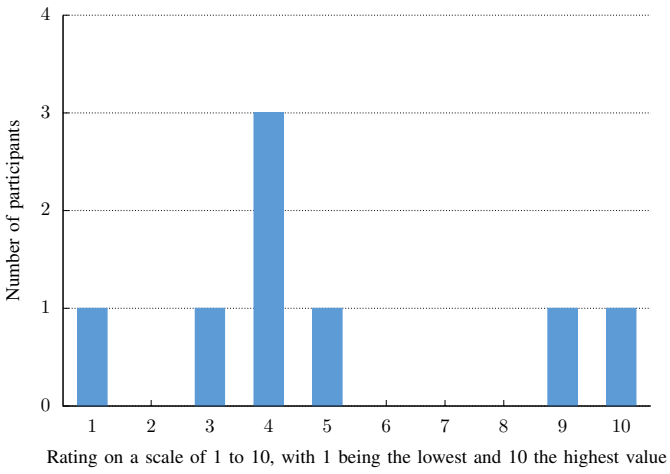


Fig. 8: Results of the question: How do you rate your MapReduce skills on a 1 - 10 scale?

the WordCount example as described in Section III-A with our prototype. In particular, the participants were asked to use Flision's user interface components and write necessary UDFs for WordCount. Afterwards, they were asked to review their generated code and submit the job to a running Flink cluster connected to the prototype. If the participant got stuck, the examiner asked for the reason and any uncertainties before providing assistance. For instance, one participant had no experience with the MapReduce paradigm and got a short introduction to the required functions and the paradigm itself. The user's behavior as well as comments from each were noted and afterwards discussed with the participants. After the prototype had been evaluated by the participants, they were asked a series of questions about the usability of the visual programming paradigm prototype. The results of the observation and the questionnaire are presented in the following subsections.

C. Observations

All participants successfully finished the given task during the test. Therefore, each created a WordCount job and submitted it to a Flink cluster using our prototype. However, the level of required assistance varied between the participants. Participants who were familiar with Flink or similar systems, required only minor assistance in handling the user interface. The participants with less experience had problems in choosing the right functions in the right order, but have been able to create the WordCount program in an assisted trial and error process.

The participants commented their impressions during the testing. These comments included multiple statements that the environment is fairly easy to use and provides a good overview over dataflow programs. At the same time, multiple participants were concerned that the level of increased overview is highly dependent on the complexity of the dataflow program. Some participants also said that they expect such an environment to be most helpful to users with only little coding skills but a good understanding of data analytics.

Besides these judgements, many participants suggested directions for future work. The main ideas for improvements are listed below:

- Zooming of the canvas and the dataflow program
- Grouping of elements and abstraction of entire subgraphs to new components
- Possibility to share custom UDFs, including using namespaces and packages
- Highlighting of the job progress and presentation of intermediate results
- Complete cluster configuration from within the environment
- Documentation, tutorials, and interactive help

D. Questionnaire

This section covers the results of the questions we asked the participants right after using our prototype.

Figure 10 summarizes the results of the following four questions.

a) How do you rate the user interface in terms of readability?: 50 percent of all participants rated the readability of Flision with the highest score. They highlight that the prototype is good for testing and prototyping, as well as for smaller explorative data analytic tasks. However, some participants commented that the advantage of increased overview diminishes when the job becomes more complex and dependent on other jobs. One participant suggested the introduction of expandable and collapsible graph components that could aggregate parts of the job graph or even full jobs to increase the overview for more complex jobs.

b) How do you rate the user interface in terms of reusability?: A majority of the participants rated the reusability with a high score. In particular, one participant that professionally implements MapReduce jobs considered project setup and skeleton generation as a promising reusability feature, as it replaces the boilerplate and glue code he has to have for every job.

c) How do you rate the time-saving potential through an improved overview?: Most participants estimated the time-saving potential for applying visual programming to distributed data analytics high on a 5 point scale, where 1 equals "no time-saving" and 5 equals "very time-saving". In general, the participants concluded that the increased overview of the visual representation is potentially very time-saving, but only for simple jobs. A majority of participants found that the combination of visual programming elements with traditional coding offers the best utility for most use cases or as one participant stated: "the combination would make a powerful tool".

d) How do you rate the user-friendliness for inexperienced users?: A majority of the participants rate the user-friendliness for inexperienced users with a high score. They point out that Flision makes it easier to get

started with dataflow systems such as Flink. In addition, new users only need to interact with the frontend and do not have to care about the Flink execution environment as well as cluster configuration. Flision makes it easy to obtain first results as it handles all the work of compiling and executing a job-graph on a Flink cluster.

Figure 9 presents the results of the question: *Can you imagine using a similar system in future projects?*

70 percent of all participants stated that they can imagine using elements of visual programming in future data analytics projects. However, most of the participants also stated that suitability of the visual approach depends much on the complexity of the task and experience. In the open feedback part after the task, one participant highlighted that: "if you understand the principle, you can reduce development time and increase productivity.". Most of the participants stated that suitability of the visual approach depends much on the complexity of the task and experience with MapReduce or other dataflow frameworks.

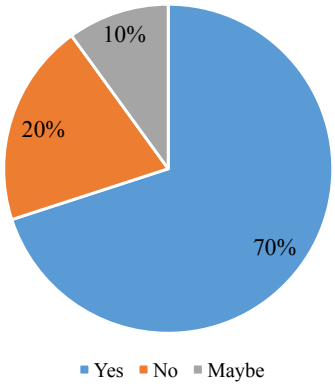
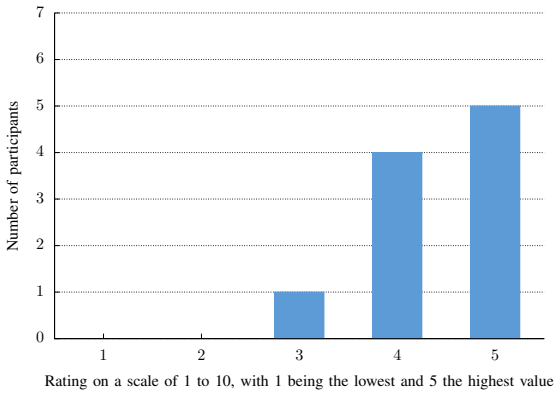


Fig. 9: Question: Can you imagine using a similar system in future development projects?

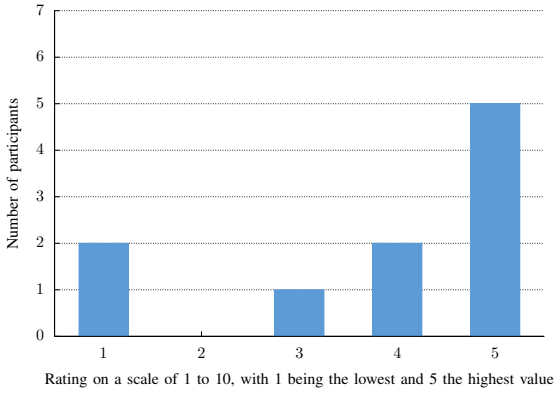
E. Discussion

The user tests show that the visual programming paradigm has large potential as a tool for data analysis. It can flatten the learning curve for users new to using distributed dataflow technology. For advanced users the approach offers improved overview, boiler- and glue-code generation, as well as fast prototyping. In addition, it has also the potential to be a powerful tool when combined with the possibility to reuse and share components.

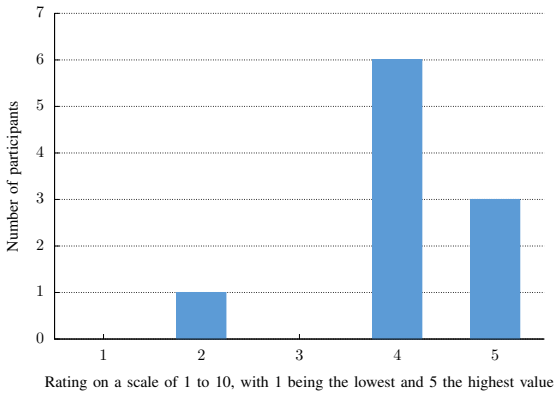
However, the user tests also point out challenges for the integration of visual programming with existing tools. In particular, it was noted that the utility is extremely dependent on the integration with existing coding environments. In fact, based on the experiences described in this paper, we are of the opinion that visual programming is best used as an additional element to existing IDEs. Visual programming has to be integrated as seamlessly as possible though. Moreover, users should have the possibility to choose between the different program representations at any time, even if this kind of



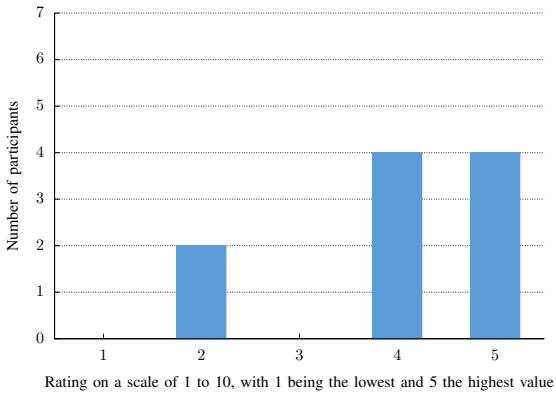
(a) Question: How do you rate the user interface in terms of readability?



(b) Question: How do you rate the user interface in terms of reusability?



(c) Question: How do you rate the user interface regarding overview?



(d) Question: How do you regarding user-friendliness for inexperienced users?

Fig. 10: Results of the questionnaire.

integration is not easily implemented. When designing visual programming elements for frameworks like Flink, it is also important to keep in mind that complex jobs require additional possibilities to aggregate complex structures such as entire graphs into single components, otherwise the advantage of an improved overview will diminish when the job becomes more complex.

VI. Related Work

This section first presents work on distributed dataflow systems. Subsequently, it discusses both programming abstractions and visual programming systems for such systems.

A. Distributed Dataflow Systems

Many systems implement distributed dataflows as their execution model.

a) MapReduce: MapReduce [1] proposed a model for scalable and fault-tolerant data processing using parallel dataflows over interconnected commodity hardware. In MapReduce's execution model data is exchanged through a distributed file system in-between alternating stages of Map and Reduce tasks. Using a fault-tolerant distributed file system such as the Google File System [13] effectively secures the results of each states. Consequently, if a node fails only the computation happening on that node needs to be restarted.

b) Dryad: Dryad [22] extended this execution model by allowing arbitrary directed acyclic graphs of user-defined tasks. Dryad also provides multiple mechanisms for data exchange between tasks, including direct network transmission without storing data on disk. Compared to MapReduce, in Dryad users can provide arbitrary task code instead of being bound to providing code for the second-order functions Map and Reduce.

c) Nephele: Nephele [23] implements the same flexible graph-based execution model as Dryad does, yet adds operators with Nephele/PACTs [24]. Therefore, users can create arbitrary directed acyclic dataflow graphs using operators like Map, Reduce, Filter, Joins, and Cross. The outputs of these tasks can also exchange data directly across the network, without storing data to disk in-between.

d) Scope: SCOPE [4] also offers a large set of predefined operators including, for example, Joins. However, Scope's key characteristic is applying features of parallel databases for its dataflow model. First, on top of the operator abstraction, Scope provides a declarative SQL-like programming languages called SCOPEScript. Second, Scope has a built-in automatic plan optimizer to compile efficient query plans from these scripts.

e) Spark: Spark [2] adds a linage-based mechanism for fault tolerance to distributed dataflows and operators with its RDDs [14]. RDDs effectively maintain enough linage information for each intermediate result to recompute specific partitions in case of failures. This serves as a low overhead alternative to disk- and replication-based fault tolerance in the failure free case. Furthermore, Spark allows to cache results to support repeated usage of the same dataset as, for example, from an iterative dataflow program.

f) Flink: Flink[2] [3] provides batch and stream processing in a single system. Dataflows are built using graphs of predefined operators and UDFs. In contrast to many systems, these graphs do not have to be acyclic. Instead Flink provides dedicated support for iterative programs, especially incremental processing for algorithms with sparse computational dependencies [25].

g) Google Dataflow: Google's Dataflow [26] is a dataflow system similar to Spark and Flink, also provides batch and stream processing, yet provides an especially large number of features with a focus on stream processing. Google's Dataflow system, for example, provides allows to explicitly handle data that arrives out-of-order.

h) Discussion: Even though we built a prototype for Flink, extending Flision to generate programs for other distributed dataflow systems would be fairly straightforward. Moreover, we argue that a visual programming system could abstract some of the specifics of each distributed dataflow framework such as the particular names and options of operators. Therefore, a single visual dataflow programming system could in principle serve as development environment for multiple of the described distributed dataflow systems.

B. Programming Abstractions for Dataflow Systems

There has been a lot of effort to provide more declarative and domain specific programming abstractions.

a) Declarative High-level Programming Languages: Many systems provide declarative SQL-like programming abstractions for distributed dataflow systems. Examples include SCOPEScript for SCOPE [4], Hive [5] for Hadoop[3], Meteor/Supremo [27] for Stratosphere [7], as well as Shark [6] and Spark SQL [28] for Spark. These scripting abstractions are provided as libraries and translate to usage of the operators that the distributed dataflow systems provide. Some of these systems such as SCOPE, Spark, and Stratosphere automatically choose among different plans using heuristic or cost-based plan optimizations [4], [11], [12].

b) Domain-specific Libraries: Besides programming abstractions for analyzing large sets of relational data with SQL-like languages, there have been multiple efforts to provide libraries for specific use cases such as machine learning and graph processing. Recent efforts to make applying machine learning methods easier and more efficient include MLLib [9] and SystemML [10], which both use Spark as runtime system. GraphX [8] on the other hand is an example of a graph library. GraphX provides a vertex-centric programming abstraction, implements many commonly used graph algorithms, and runs on Spark.

c) Discussion: These efforts aim at providing the right level of abstraction and, thereby, try to make programming distributed dataflow systems easier. These libraries, however, remain textual programming languages, requiring users to be familiar with syntax, language concepts, and APIs. In

[2]Flink originated from the Stratosphere project [7]

[3]https://hadoop.apache.org, accessed 2016-10-08

comparison, we propose to use visual programming, so less coding is required and users also get a better overview.

C. Visually Programming Data Analytics Pipelines

Multiple projects have applied visual programming to make data analytics more accessible.

a) Knime Analytics Platform: The Knime Analytics Platform[4] [29] includes a graphical user interface to construct programs visually out of available data transformations. Knime provides a visual workspace where datasets and analysis modules can be dragged and dropped onto an interactive canvas. These modules can then be connected to form a dataflow program. Modules provide interfaces for further customization of their behavior. Pre-defined modules exist for many use cases, especially for data preprocessing, statistics, and data mining. More modules can be provided by users.

b) Azure Machine Learning Studio: Azure Machine Learning Studio[5] is a graphical programming system for constructing and running Machine Learning workflows based on Microsoft's Azure Machine Learning service [30]. Built-in transformations that can be configured and used for programs include operators for classification, regression, ranking, and clustering. Jobs can be created completely visually and executed on the Azure cloud.

c) Sparkflows.io: Sparkflows[6] is a system with a drag and drop interface to interactively build end-to-end data pipelines. Created dataflow programs can be executed on a Spark cluster. The pre-defined transformations focus on machine learning. Results can also be visualized with Sparkflows.

d) Discussion: The Knime Analytics Platform, Microsoft Azure Machine Learning Studio, and Sparkflows are comparable to our vision and to our prototype. All three solutions provide visual environments for a dataflow analytics framework. A key difference is, however, that all three of the described platforms focus on data mining and machine learning applications, especially through pre-defined transformation components. Flision is a general purpose programming environment for creating analytical distributed dataflow programs. For this reason, Flision provides not only focusses on standard dataflow operators, but also provides integrated configuration of operator options and editing of operator code.

VII. Conclusion

In this paper we proposed to use visual programming for distributed dataflow systems like MapReduce, Spark, and Flink. To evaluate this proposal and see whether this combination actually provides benefits compared to textual programming abstractions, we developed a prototype of a visual programming system for Flink, called Flision. In contrast to traditional solutions where the operations are programmed using a programming language, with a system like Flision the user adds operators to a graphical workspace and connects them visually to form a graph. Each of the operators in the graph can be configured within this workspace so that Flink jobs can be developed completely visually. For operations that need to be configured with UDFs, users can either load existing UDFs into the environment or program them within Flision. However, even though new UDFs still have to be expressed using code, the code connecting the components is generated automatically by Flision, reducing the overall amount of coding in any case.

Our user testing showed that using a visual programming system for parallel dataflow programming increases the overview, supports new users, and helps to prototype. It also reduces the amount of boilerplate and glue code. At the same time, our user testing made it obvious that users still need to be able to write UDFs for their components or have a large number of pre-defined transformations available. Furthermore, they also need to have a sound understanding of the semantics of the available data transformations.

The participants of our user tests also suggested future work. Some participants suggested the possibility of abstracting and reusing entire subgraphs. This would increase the readability of larger programs as well as make entire sequences of components reusable at once. An example would be separating three distinct steps of a pipeline that starts with operators for data cleansing, then applies multiple operators that implement a machine learning method, before the results are finally aggregated for human consumption. A high level view of this program could just show the three steps explicitly. Components should also be easily shareable among communities of users of a platform like Flision. A possible strategy for this could be based on shared repositories. Another interesting direction brought forward by our test participants resolves around the idea of presenting intermediate results when a job is executed. We believe that a combination of such features with effective sampling could make the analysis of large datasets much more interactive and immediate.

Even though these ideas make it obvious that much research remains to be done, the results presented in this paper show that the combination of visual programming and distributed dataflow programming is very promising. Based on our experiences, we think the best option is integrating visual programming elements into existing IDEs in a way that users can freely choose between the different program representations.

Acknowledgments

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

We also thank Markus Aurich, Nico Bahlau, Adrian Bartnik, Iwailo Denisow, Alexander Kumaigorodski, Raphael Rieprecht, and Fabian Schürer for their contributions to this research, especially to the prototype presented in this paper.

[4]http://www.knime.org, accessed 2016-06-12

[5]https://studio.azureml.net, accessed 2016-06-12

[6]http://www.sparkflows.io, accessed 2016-10-07

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. USENIX Association, January 2004, pp. 10–10.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USENIX Association, June 2010, pp. 10–10.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, July 2015.

[4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, August 2008.

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a Map-reduce Framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, August 2009.

[6] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and Rich Analytics at Scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. ACM, June 2013, pp. 13–24.

[7] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, December 2014.

[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USENIX Association, September 2014, pp. 599–613.

[9] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, January 2016.

[10] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda, "SystemML: Declarative Machine Learning on Spark," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1425–1436, September 2016.

[11] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing Data-parallel Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, April 2012, pp. 21–21.

[12] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the Black Boxes in Data Flow Optimization," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1256–1267, July 2012.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, October 2003, pp. 29–43.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, 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 Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, April 2012, pp. 2–2.

[15] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl, "Implicit Parallelism Through Deep Language Embedding," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, pp. 47–61.

[16] R. Jamal and L. Wenzel, "The Applicability of the Visual Programming Language LabVIEW to Large Real-world Applications," in *Proceedings of the 11th IEEE International Symposium on Visual Languages*. IEEE, Sep 1995, pp. 99–106.

[17] E. Baroth and C. Hartsough, "Visual Object-oriented Programming," M. M. Burnett, A. Goldberg, and T. G. Lewis, Eds. Manning Publications, January 1995, ch. Visual Programming in the Real World, pp. 21–42.

[18] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle, "Fabrik: A visual programming environment," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '88. ACM, December 1988, pp. 176–190.

[19] J. Lincke, R. Krahn, D. Ingalls, and R. Hirschfeld, "Lively Fabrik: A Web-based End-user Programming Environment," in *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*. IEEE, January 2009, pp. 11–19.

[20] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. ACM, April 2010, pp. 2503–2512.

[21] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE, May 2012, pp. 1064–1073.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, March 2007, pp. 59–72.

[23] D. Warneke and O. Kao, "Nephele: Efficient Parallel Data Processing in the Cloud," in *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '09. ACM, November 2009, pp. 8:1–8:10.

[24] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, June 2010, pp. 119–130.

[25] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning Fast Iterative Data Flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, July 2012.

[26] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, August 2015.

[27] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, "Meteor/Sopremo: An Extensible Query Language and Operator Model," in *Proceedings of the International Workshop on End-to-end Management of Big Data (BigData) in conjunction with VLDB 2012*. VLDB Endowment, January 2012.

[28] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, May 2015, pp. 1383–1394.

[29] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME - the Konstanz Information Miner: Version 2.0 and Beyond," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 26–31, November 2009.

[30] R. Barga, V. Fontama, and W. H. Tok, *Predictive Analytics with Microsoft Azure Machine Learning*, 2nd ed. Apress, 2015.