Journal of Big Data

RESEARCH

Open Access

Graphical Flow-based Spark Programming

Tanmaya Mahapatra*† and Christian Prehofer†

*Correspondence:
tanmaya.mahapatra@tum.de
†Tanmaya Mahapatra and Christian Prehofer contributed equally to this work
Lehrstuhl für Software und Systems Engineering, Fakultät für Informatik, Technische Universität München, Boltzmannstraße 03, 85748 Garching b. München, Germany

Abstract

Increased sensing data in the context of the Internet of Things (IoT) necessitates data analytics. It is challenging to write applications for Big Data systems due to complex, highly parallel software frameworks and systems. The inherent complexity in programming Big Data applications is also due to the presence of a wide range of target frameworks, with different data abstractions and APIs. The paper aims to reduce this complexity and its ensued learning curve by enabling domain experts, that are not necessarily skilled Big Data programmers, to develop data analytics applications via domain-specific graphical tools. The approach follows the flow-based programming paradigm used in IoT mashup tools. The paper contributes to these aspects by (i) providing a thorough analysis and classification of the widely used Spark framework and selecting suitable data abstractions and APIs for use in a graphical flow-based programming paradigm and (ii) devising a novel, generic approach for programming Spark from graphical flows that comprises early-stage validation and code generation of Spark applications. Use cases for Spark have been prototyped and evaluated to demonstrate code-abstraction, automatic data abstraction interconversion and automatic generation of target Spark programs, which are the keys to lower the complexity and its ensued learning curve involved in the development of Big Data applications.

Keywords: Spark pipelines, IoT mashup tools, Graphical tools, Stream analytics, Flow-based programming

Introduction

Within the advancements in information and communication technologies in the last years, there are two significant trends. *First*, the number, usage and capabilities of the end-user devices, such as smartphones, tablets, wearables, and sensors, are continually increasing. *Second*, end-user devices are becoming more and more connected to each other and the Internet. With the advent of 5G networks, the vision of ubiquitous connected physical objects, commonly referred to as the Internet of Things (IoT), has become a reality. In the world of connected physical devices, *there is a massive influx of data*, which is valuable for both real-time as well as historical analysis. Analysis of the generated data in real-time is gaining prominence. Such analysis can lead to valuable insights regarding individual preferences, group preferences and patterns of end-users (e.g. mobility models), the state of engineering structures (e.g. as in structural health monitoring) and the future state of the physical environment (e.g. flood prediction in rivers). These insights can, in turn, allow the creation of sophisticated, high-impact applications. Traffic congestion can be avoided by using learned traffic patterns. Damages to buildings and bridges can be better detected, and repairs can be better planned by using structural health monitoring techniques. More accurate prediction of floods can enhance the ways authorities and individuals react to them.

Data insights are crucial to develop high impact applications. These insights can guide the development process to continually improve the quality of applications and continuously cater to the ever-changing needs of the users, thereby leading to a significantly higher rate of user satisfaction. In specific scenarios where the response of the system is context dependent, continuously performing data analytics to guide the business logic of an application is unavoidable. For example, consider an application responsible for alerting and routing ambulances and fire brigades to different parts of the city: such an application needs to perform data analytics continuously to provide effective responses.

Deriving insights from data collected is a separate challenge that falls primarily into the topic of data analytics and machine learning. Traditional Data Science techniques deal with deriving insights from datasets gathered using programs like R and Python. Examples of Python libraries include NumPy [1], SciPy [2], Pandas [3], SciKit-Learn [4], Keras [5] and others. The gathered data is cleaned up, subjected to a series of transformations, analysed and results are either visualised or saved in human readable formats. Mostly, these are *non-cluster based* techniques relying on the computing power of a single machine. There are two significant shortcomings with this style of data analytics:

1. It does not scale well to handle the processing of vast amounts of datasets (i.e. the volume of data) generated from millions of IoT sensors.
2. It cannot support the processing of large datasets in real-time (i.e. the velocity of data) to make a business decision, i.e. no support for stream analytics.

Therefore, in the past years, several sophisticated tools have emerged that focus on manipulation and analysis of data of high volume, velocity and variety commonly referred to as Big Data. Big Data analytics tools allow parallelised data analysis and learn via machine learning algorithms operating on datasets that reside in large clusters of commodity machines cost-effectively. Mostly, these are *cluster-based* data science techniques. These tools are typically used to cater to different business needs like targeted advertising, social network analysis etc. The term 'Big Data' in a technical sense refers to a specific set of storage and query languages like HDFS [6], Hive [7], Pig [8] etc. These tools have their semantics and lines of operation. Storing datasets and writing programs to analyse them involves working with different APIs and libraries. Even though the Big Data toolchains are designed for heavy real-time as well as historical data analytics, there are certain limitations associated concerning their usage:

- Working with Big Data frameworks requires developing programs using several libraries and APIs as well as working with different data abstraction formats. Including a Big Data analytics method in a user application is not easy. Developers need to write complicated code and include drivers for integration of the application with Big Data systems. This approach makes the process quite cumbersome and challenging to quickly prototype applications for performing exploratory data researches because of a vast number of varied solutions available in the ecosystem.

- The learning curve associated with it is steep and requires a considerable amount of technical expertise to use it. There is no support for end-user programming with Big Data, i.e. programming at a higher abstraction level.
- Deployment of Big Data programs on clusters for execution, fetching results back for insights and management of clusters require a considerable amount of DevOps training and expertise.

In the context of Spark For example, in Spark to create a machine learning (ML) model and apply it on streaming data, Spark provides Spark Streaming and Spark Structured Streaming libraries operating on DStreams and Streaming DataFrame data abstractions, respectively. Similarly, for creating a ML model, we can either use the resilient distributed dataset (RDD) based Spark MLlib or the newer DataFrame based Spark ML library. The choice of the streaming library would influence the choice of the machine learning library as well due to compatibility between the underlying data abstractions of the libraries. Even in the case of compatible data abstractions, the conversion of data from one data abstraction to another requires additional program codes and is far from being a straightforward process. Additionally, the complexity of using the different APIs within a library in a correct sequence, with appropriate data transformations between two API calls and clearly defining all the vital components of a Spark program like initiating and closing a Spark session make it non-trivial for a less skilled Big Data programmer to quickly prototype Spark applications.

A promising solution is to enable domain experts, that are not necessarily programmers, to develop the Big Data applications by providing them with domain-specific graphical tools. In the context of visual programming, two approaches are widely used, i.e. *block-based programming* [9] and *flow-based programming* [10] with both the approaches having their pros and cons, respectively [11, 12]. Nevertheless, as far as programming-in-the-large [13] is concerned, such as complex heterogeneous systems which can also accommodate the example of Big Data systems, flow-based programming approaches are well suited [14–16] compared to their block-based counterparts [9].

Results

This paper focuses on enabling graphical Spark programming (version 2.2.0) at a higher level with modular components via the flow-based programming paradigm. The main challenge in this direction is the presence of a wide range of data abstractions and APIs in the Spark ecosystem. This paper caters to the research contributions by the following:

1. Providing a thorough analysis of the Spark ecosystem and selecting suitable data abstractions for use in a graphical flow-based programming paradigm (discussed in "Data abstractions and APIs in Spark" section).
2. Bundling selected Spark APIs as modular composable components (discussed in "SparFlo: a subset of Spark APIs bundled as modular components" section).
3. Devising a novel, generic approach for programming Spark from graphical flows using the distilled components, comprising early-stage validation with feedbacks and code generation of Java Spark programs (discussed in "Conceptual approach for flow-based Spark programming" section).

The graphical programming concepts have been prototyped in **aFlux** [17–21], a JVM-based mashup tool and has been evaluated in three use cases ("Experimental" section) showcasing code-abstraction, automatic data abstraction interconversion and automatic generation of target Spark programs. The graphical programming concepts discussed in this paper is the first approach to support high-level Big Data application development by making it independent of the target Big Data frameworks (discussed in "Discussion" section). While this paper focuses solely on graphical Spark programming, preliminary work concerning extensions to the conceptual approach to support additional target frameworks like Apache Flink [22] has already been published [17].

Structure The rest of the paper is structured in the following way: "Background" section summarises the background information and "Related work" section summarises the state-of-the-art. "Towards flow-based Spark programming" section discusses the challenges and design issues for supporting graphical flow-based Spark programming. "Data abstractions and APIs in Spark" section summarises and compares the different data abstractions; and classifies the various APIs found in the Spark ecosystem based on their functionality and method signature. The selection of specific data abstractions and Spark APIs for supporting flow-based Spark programming is also done here. "Conceptual approach for flow-based Spark programming" section describes the conceptual approach and discusses '***SparFlo***', a library consisting of modular composable components. The components bundle a set of Spark APIs which are executed in a specific order to perform one data analytic operation. This uses only a subset of Spark APIs which are compatible with the flow-based programming paradigm. "Experimental" section describes the evaluation of the approach. This is followed by a discussion and comparison to existing works in "Discussion" section and concluding remarks in "Conclusion" section.

Background

Apache Spark

Apache Spark [23] is a fast, scalable, fault-tolerant general-purpose distributed computing platform. It makes efficient use of memory and is generally faster than traditional MapReduce programming model. It is yet another approach to cluster computing like MapReduce [24], Message Passing Interface (MPI) [25] and others. Nevertheless, its programming model has been designed to overcome the limitations of MapReduce [26]. It has excellent in-memory computation capabilities for scenarios which demand iterative computations, e.g. application of machine learning (ML) techniques, which involves the application of an algorithm repeatedly on the same dataset till optimum results are obtained, interactive explorations which enable users to submit SQL like queries and stream processing. In addition to this, Spark supports both batch as well as stream analytics. It has evolved from a framework to an ecosystem, with several libraries built around the core framework; Spark SQL [27] provides an SQL-like interface for data analysis, GraphX [28] can be used for graph computations, and different ML libraries [29] to learn from datasets. Spark is implemented in Scala [30] but provides APIs in Scala, Java, Python as well as R. It has become the most popular data analytics platform surpassing traditional MapReduce style of doing distributed data analytics and has been adopted by big giants in the IT sector. For instance, there is a Spark cluster set-up in production consisting of 8000 live nodes [31].

Fundamental concepts in Spark

Spark is under substantial research and development. With every release, new libraries and new programming models are added to it. Nevertheless, they are all centred around Spark Core. The different libraries cater to different aspects of Big Data analytics. The essential component in the Spark ecosystem is the Spark Core, which provides basic functionalities for running Spark Jobs. Spark allows us to use different libraries, and it is different from traditional Hadoop technologies. It overcomes one of the strongest shortcomings of MapReduce, i.e. a job's results need to be saved before another job can use them. Spark's core concept is an in-memory execution model that enables caching a job result in memory instead of fetching it every time from the hard drive. Consider an example [23] where we have stored the city map data as a graph. The vertices of this graph represent points of interest on the map, and the edges represent the possible routes between them along with the distance value. In order to locate a new spot on the map for a new building such that it is as close as possible to all points in the graph, we have to: (i) calculate the shortest path between all the vertices, (ii) find the farthest point distance, i.e. maximum distance to any other vertices for every vertex and (iii) find the vertex with minimum farthest point distance. In the case of a MapReduce solution, this would require three steps where the result of every preceding step needs to be saved first before it can be used in the following step. However, in Spark, these can all be computed in-memory using the concept of caching. Here we summarise the fundamental concepts of Spark as a distributed analytics engine that have been discussed.

Spark application program A Spark application can be programmed from a wide range of programming languages like Java, Scala, Python and R. In case of programming from Python, the Python source code itself is the Spark application program. However, in the case of Java/Scala, the source code is compiled to generate a Java ARchive (JAR) file. This JAR file is typically sent to a Spark cluster for execution and is traditionally known as the '*Spark driver program*' or the '*Spark application program*'.

Resilient distributed dataset The most fundamental concept provided by Spark core for running Spark jobs is the '*resilient distributed dataset*' (RDD) [26, 32]. RDDs are a read-only collection of objects which represent user input that has been partitioned over machines in the cluster. It is possible to have multiple partitions on the same machine. Each RDD contains the transformation(s) that will be applied to the data by worker processes. When a machine containing a worker process fails, information present in RDDs can be used by another worker machine to recompute the lost computation. RDDs are computed from other RDDs by applying coarse-grained transformations [26] or by reading user input from disk. RDDs are not required to be stored in physical memory as they can be recomputed at any time. However, when necessary, users can persist data represented by an RDD in memory. For example, if there is a 384 MB file which needs to be stored in a 3-node cluster set-up, HDFS automatically splits the file into 128 MB blocks and places each part on a separate node of the cluster. If the file is needed to be used by a Spark program, then the corresponding parts are loaded into the main memory of the respective nodes, and an RDD is created. The RDD thus created contains a reference to each of the blocks loaded into RAM of different cluster nodes. RDD abstracts things so that it becomes easy to work with a distributed collection and takes care of communication as well as node failure issues [23].

Spark program execution When a Spark application is submitted, a process called 'Spark driver' inspects the application and prepares an execution plan. Execution plan consists of RDDs along with computations to be performed on them. Once the execution plan is ready, worker processes/nodes are invoked to read data from external sources and perform computations as scheduled by the 'Job Scheduler' process within Spark. Spark driver program has full control over the resources required to orchestrate, control the execution and manage the worker processes. Reading data from external sources and actual computation occurs within the worker processes. Once the computations have been performed, data could either be pushed out of the run-time environment of worker processes into external receivers or brought into the driver process. Only computations whose output is sent out of the Spark execution environment are scheduled for actual execution. This style of evaluation strategy is known as '*lazy evaluation*'. The advantage of lazy evaluation is that computations which do not end in sinks, i.e. push-data out, are not executed thereby reducing computations.

Flow-based programming

Flow-based programming (FBP) is a programming paradigm invented by Morrison in the late 1960s which uses data processing pathways to design user applications [33]. It defines user applications as networks of black-box processes communicating via data chunks travelling across distinct pathways [10]. Conventional programming paradigm, i.e. control-flow programming, typically concentrates on operations and gives secondary preferences to data. In contrast to this, business applications generally are concerned with how data is processed and handled, i.e. have stringent requirements on how the data moves in the system. Consider a program which needs to be developed to read records, and if these records match specific criteria, they are sent for further processing/output or else they get discarded. A typical function written in control-flow programming paradigm would concern itself with taking care of needed records and ignoring other records. On the contrary, in strict flow-based programming paradigm, the program should specify handling logic for both matchings as well as non-matching records. Every function can be represented as a black box, i.e. a node while the parameters of the function and its return value can be represented as ports of the node via which the node consumes input and yields output. Connected pathways between such data processing black-boxes form the core of the flow-based programming paradigm. Some preliminary concepts fundamental to the flow-based programming paradigm include nodes, ports, pathways, data, dataflow graph and execution of a graph. Nodes are data processing 'black-boxes' which consume input and produce output only via ports. They may be functional, i.e. produce the same output repeatedly if given the same input multiple times. Ports are connection points between a node and data pathways. Pathways are connections between an input port of a node to an output port of a node. Pathways have a buffer limit and also support splitting into two different output ports or merging from different input ports. Data is the processing as well as controlling item flowing through the pathways from one node to another. Typically, a dataset is immutable. The directed graph formed by considering the connection between different nodes via valid pathways is a dataflow graph. The execution of a graph typically begins from the node which loads the data and pipes it into the pathway after processing it. These nodes are also called 'source nodes'. They do not have an input port. The nodes where the execution finishes are known as 'sink nodes'. These nodes do not have an output port. The execution can follow either a 'pull' or 'push' mechanism. In the 'push' mechanism, a node pipes its output as soon as it is available while in the 'pull' mechanism a node typically pulls its input from its preceding node when required. FBP is a particular case of dataflow programming. The dataflow programming paradigm specifies that dataflow controls the execution pathway of a program [34]. The actor model [35] is a very dynamic form of dataflow programming where the nodes can scale up when the situation demands and the buffer capacity of pathways can be configured as per needs [34]. The actor model relies on message passing to send data from one actor to another asynchronously. Every actor has a unique address of the pathway leading to the receiving actor. A mailbox associated with every actor (typically an ordered queue) stores received messages which are processed concurrently by the receiving actor. It is an asynchronous dataflow mode with nodes, i.e. individual actors, being strictly functional in design. aFlux, the flow-based programming tool is based on the actor model, and the graphical Spark programming concepts have been prototyped using it.

IoT mashups and mashup tools

A mashup application is a composite application developed through the agglomeration of reusable components. The individual components are known as 'mashup components', and they form the building blocks of the mashup application. The specification of control-flow between these mashup components forms the mashup logic. As control flows from one component to the next, it typically involves potential data mediation before the data received from the preceding component can be used, as well as the execution of business logic, defined within the component. This process, performed sequentially from the first to the last component of the flow, defines the business logic of the application. Graphical programming tools which permit the creation of such mashup applications are called mashup tools, and they follow the FBP paradigm. Current mashup tools do not support Big Data programming [36]. A detailed article on mashups and mashup tools has been published [37].

Related work

We did not find any flow-based programming tool or other research works which support generic high-level programming for Big Data systems independent of the underlying Big Data framework and execution engine. However, there are many different solutions to reduce the challenges involved in using specific Big Data frameworks. Here we summarise the current works and solutions aimed at supporting high-level graphical programming of Big Data applications.

The QualiMaster infrastructure configuration (QM-IConf) tool

The QM-IConf tool [38] supports model-based development of Big Data streaming applications. It introduces a high-level programming concept on top of Apache

Storm [39]. It features a graphical-flow based modelling of the streaming application in the form of a dataflow graph where vertices are stream operators, and edges represent valid dataflow paths. A valid dataflow path from vertex $v1$ to $v2$ ensures that $v2$ can consume the data produced by $v1$. The dataflow model consisting of data sources, sinks and operators, is translated into an executable Storm code. Nevertheless, it does not validate its claimed generic modelling approach against other streaming frameworks like Flink or Spark Streaming. Additionally, it supports only a specific subset of stream analytics operators to be used in the pipeline.

Lemonade

Lemonade (Live Exploration and Mining Of a Non-trivial Amount of Data from Everywhere) is a platform designed to support framing of graphical data analytics pipeline and to translate the graphical flow into a runnable Spark program [40]. It translates visual flows into Spark application in Python programming language and provides visualisation of the datasets produced from running application. It consists of front-end where the user can develop Spark flows using dragging and dropping graphical components and wiring them to form a flow. A JSON object is prepared from the graphical flow and is translated into a Spark application such that each graphical component is a Spark method call. The main disadvantage is that it provides graphical components for producing applications using Python APIs of Spark MLlib library only. It does not support framing applications for stream processing or other libraries of the Spark ecosystem. Additionally, the code generation method is not generic, uses hard-coded method call code snippets and appends them in a Python script. The tool can specify to ingest data already existing on the Big Data cluster only and not from live sources. To summarise, it is a visual programming tool for Spark machine learning restricted to specific APIs and use cases.

QryGraph

QryGraph [41] is a web-based tool which allows the user to create graphical Pig queries in the form of flow along with simultaneous syntax checking to support batch processing of datasets stored on HDFS. In QryGraph, a Pig query is represented as a dataflow graph with vertices representing data sources, sinks and transformers while edges represent valid dataflow pathways. It also allows the user to deploy the created job and manage its life-cycle.

Nussknacker

It is an open-source tool which supports model-based development of Flink applications. It also supports deployment and monitoring of Flink jobs [42]. First, a developer needs to define the data model of an application specific to a use case inside the 'Nussknacker engine'. The engine is responsible for transforming the graphical model created on the front-end into a Flink job. It uses the data model and its associated code-generation together with the front-end graphical model created by a user to generate the final Flink program. The code generation technique specific to a model should be defined beforehand. Finally, users with no prior knowledge and expertise in Flink can use GUI to design a Flink job as a graphical flow, generate the actual Flink program, deploy it to a cluster and monitor its output.

Apache Beam

Apache Beam [43] is a unified programming model and provides a portable API layer to develop Big Data batch as well as streaming applications. The programming model uses the concept of a pipeline to represent an application. In essence, a pipeline models an application as a dataflow graph consisting of sources, sinks and operators to do the data processing. An additional feature is that every pipeline ends with a runner configuration which can be specific to target Big Data Frameworks like Apache Spark, Apache Flink etc. The programming model translates the pipelines into a native Big Data job and executes it in the target environment, i.e. the same beam program can be deployed on Spark, Flink, Apex [44] etc. clusters. The beam programming model supports operators necessary for batch and stream operations, and if a target framework supports a corresponding feature, then a beam application can be run in that specific environment.

Other solutions

Apache Zeppelin provides an interactive environment for using Spark instead of writing a complete Spark application. Zeppelin manages a Spark session within its run-time environment and interacts with Spark in interactive mode [45] while consuming code snippets in Python/R. Nevertheless, to successfully interact with Spark, Zeppelin still requires programming skills from users since it requires a compilable code.

Another existing solution is *Azure*, a private cloud computing service offered by Microsoft, which also offers Spark as a service. Users can configure a Spark cluster without requiring any manual installation. Here, Spark can be used to run interactive queries, visualise data and run machine learning algorithms [46]. Nevertheless, it expects the user to have programming expertise.

Apache NiFi [47], a tool for creating data pipelines in the form of visual flows, supports integration with Big Data execution engines represented by a GUI component called a processor. Nevertheless, this processor needs to contain the Big Data application code. From the perspective of an end-user, NiFi does not reduce the programming challenges associated with Big Data, although automation via data pipelines is undoubtedly provided.

"Discussion" section gives a detailed comparison of our approach for high-level graphical programming concepts for Spark with existing solutions.

Towards flow-based Spark programming

The graphical Spark programming concepts, i.e. a user designs a flow by dragging and dropping graphical components and the system generates a complete Spark application for the user typically lowers the learning curve associated in using as well as adopting Spark. However, in order to support such a scenario, there are several challenges which need to be addressed by the conceptual approach: (i) diverse data-representational styles, APIs and libraries centred around Spark make it challenging to extract a common programming approach, i.e. using similar data structures and APIs to load, transform and pass datasets, with which to access the functionalities of Spark and formulate an approach to use it from a flow-based programming paradigm. Selection of compatible APIs and bundling them together as modular components is the first step which has been done and (ii) reconciling the difference in the programming paradigm of Spark and flow-based mashup tools can be a challenge. Spark relies on lazy evaluation, where computations are materialised if their output is necessary, while flow-based programming has a component triggered, then proceeds to execution, and finally passes their output to the next component upon completion. To program Spark from flow-based programming tools, this difference has been addressed by introducing additional auxiliary components in the programming tool level called *'bridge'* components ("Supporting sequenced Spark transformations in flow based programming paradigm" section) and starting the code generation process at the last component of the user flow.

In addition to the challenges, there are several design issues for graphical flow-based Spark programming (the first two design issues are the results of the first challenge while the last design issue is due to the second challenge enumerated above):

Design of a modular Spark component

A modular component is the basic unit of composition in a graphical flow. A typical Spark application consists of different Spark APIs invoked in a specific sequence to represent the business logic. Therefore, it makes perfect sense to model the graphical programming concepts on the same lines, i.e. represent Spark APIs via modular-GUI components which the user can drag and connect. The flow thus created essentially represents a sequence with which the Spark APIs are invoked to meet the business logic of the user.

Modularity

By *modularity*, we mean that the components are designed so that their functionality is independent of each other and contain the necessary code to execute only one aspect of the desired functionality, i.e. one specific data analytic operation—high-cohesion with loose coupling of components. [48, 49].

The first concern in this regard is deciding the *granularity level* of such a Spark component to make it *modular* in its design. Spark APIs essentially accomplish one small task within the Spark engine like converting data from one form to another. Hence, to perform a small operation like reading datasets from a streaming source, the developer has to invoke many different APIs. Hence, a modular Spark component should ideally make use of multiple Spark APIs invoked in a specific order to perform one fundamental data analytics operation. The second concern is with respect to *abstraction-level*. A typical Spark application has many aspects which do not directly correspond to the business logic but are vital for running of the application. For instance, the 'application context' which is a programming handle to identify a running Spark application, essentially manages the communication between Spark driver program and worker nodes during execution. While programming the application context, a developer configures it with various options like providing a name for the Spark application, deciding whether the application runs in a distributed mode or not and specifying the address of the Spark cluster manager. It is also interesting to mention that for batch processing where the computations typically finish, the application context must have an end statement while this is not true for streaming applications as they typically run indefinitely. Hence, portions of a Spark application not contributing to the business logic of the application, i.e.

loading, transformation or display of datasets, should not be represented as components but abstracted from the user.

Data transformation approach

Data transformations in Spark can be achieved via RDD based operations or the newer declarative APIs which operate on a higher level of data abstractions over RDD. The RDD based data transformations require the developer to write their own custom data transformation functions while the APIs are kind of pre-defined transformation functions which can be invoked directly on datasets. In traditional Spark application development, developers typically use a mix and match of both techniques. However, from a graphical programming perspective, use of API based data transformation is more reasonable as a graphical flow can be represented as a flow between different APIs connected by the end-user to achieve the desired result.

Design of the translator model

The translator model which takes the graphical flow and auto-generates the Spark application program should parse the auxiliary components ("Supporting sequenced Spark transformations in flow based programming paradigm" section) properly as they do not represent any Spark API but are used to bridge the difference in the computational model. Furthermore, it should ensure that the flow created by the user will yield a compilable Spark program. For this, it should provide early feedback to the user in case of improper usage of components in a flow. In this way, the errors arising out of the wrong formulation of a Spark flow are handled at the mashup tool level. Once the translator model accepts the Spark flow, it should result in a compilable Spark program. Additionally, the translator model should be generic and extensible to support inclusion of additional APIs, libraries and features of Spark in future.

Data abstractions and APIs in Spark

In this section, various transformations, data abstractions and APIs supported by Spark are discussed in detail to give an understanding of how Spark is used from a developer's perspective.

Data abstractions in Spark

Spark supports two kinds of transformations among different libraries in its ecosystem. First, it supports high-level operators which apply user-defined methods to data, e.g. the *map* operator. The user-defined operation has to be provided by the developer. Newer libraries of Spark have moved away from this paradigm and instead offer fine-grained operations where the operation logic is pre-fixed yet parametrisable by the programmer. Spark has introduced several *data abstractions* like RDD [50], DStream [51], DataFrame [23, 27, 52] and Streaming DataFrame [53] *to manage and organise user data within its run-time environment* and several libraries have introduced data abstractions customised for different cases. Table 1 summarises the libraries and the data abstractions each library requires for interaction. The term '*data abstraction*' and the term '*data interface*' convey the same concept and meaning. For the rest of the paper, we use the term '*data abstraction*' for maintaining homogeneity and clarity in discussions.

Table 1 Spark libraries and their supported data abstractions, as in [18, 57]

Data abstraction / Library	RDD	DStream	DataFrame	S. DataFrame
Spark Core	Yes	–	–	–
Spark Streaming	–	Yes	–	–
Spark SQL	–	–	Yes	–
Spark MLlib	Yes	–	–	–
Spark ML	–	–	Yes	–
Spark Structured Streaming	–	–	–	Yes

Fine-grained operations are possible through the declarative APIs of Spark, i.e. all APIs of libraries listed in Table 1, that are based on the Spark core library.

The resilient distributed dataset (RDD) abstraction It is the key data abstraction for in-memory processing and fault-tolerance of the engine, which is used heavily for batch processing. Higher level constructs are supplied with user-defined functions (UDFs) applied to the data in a parallel fashion. UDFs have to adhere to strong type checking requirements.

DStream data abstraction Spark streaming is an extension of Spark Core which provides stream processing of live data. Data can be read from streaming sources like Kafka [54], Twitter, Flume etc. and processed in real-time using high-level functions like map, reduce, join, window etc. The final processed output can be saved either to file systems, HDFS or even databases. Internally, Spark Streaming divides the live stream of input data into batches/chunks and feeds to the core Spark engine for processing, the output from the Spark engine is again in the form of batches of processed output. Spark Streaming provides a high level abstraction called '*discretised stream*' or '*DStream*' [51, 55] to represent a continuous stream of data. DStreams can be created from input data streams like Kafka, Twitter etc. or by applying high-level operations on other DStreams. DStream, the basic abstraction provided by Spark Streaming is internally a continuous series of RDDs, Spark's immutable distributed dataset abstraction. Any operation applied on a DStream is translated to operations on RDDs. It is interesting to note that all these RDD operations and transformations are abstracted from the developer while using the DStream abstraction and its associated operations; instead, the developer is provided with a high-level API for convenience. DStream collects data streamed over a user-defined interval and combines them with the rest of the data received so far to create a micro-batch. This approach hides the process of combining data. Spark Streaming operations can be performed on a DStream abstraction or an RDD abstraction, as DStream can be operated on by converting it to RDD. While this library is not a true stream processing library (it internally uses micro-batches to represent a stream), the most important aspect is its compatibility with Spark MLlib [29] library which makes it possible to apply machine learning models learned offline, on streaming data.

DataFrame data abstraction The *DataFrame API* [52, 56], introduced by the Spark SQL library, is a declarative programming paradigm for batch processing built using the DataFrame abstraction. This data abstraction treats data as a big table with named columns, similar to real-world semi-structured data (e.g. Excel file). DataFrame API provides a declarative interface, with which data and parameters required for processing can be supplied. Data is read into the environment using user-defined schema, and DataFrame is created as a handle to trace the data as the schema changes in the course of transformations. The actual implementation of the operations performed on the data to produce the desired transformation is abstracted from the user. Spark ML is accessed using the DataFrame API.

Streaming DataFrame data abstraction Spark Structured Streaming is a fault-tolerant stream processing engine built on top of the Spark SQL engine. The Spark SQL engine is responsible for running the streaming query incrementally and continuously updating the result as new streaming data arrive. Structured Streaming queries are executed in small micro-batches internally. The Spark Structured Streaming library provides real-time stream processing using *Streaming DataFrame* APIs, an extension of DataFrame APIs. The Streaming DataFrame API can be used to express all sorts of streaming aggregations, event-time windows etc. The critical idea in Structured Streaming abstraction is that live data stream is treated as a table which continuously grows, and newly arriving data is appended to it. The processing model is very similar to the batch processing model as the streaming logic is applied as a batch-query and Spark applies it as an incremental query on top of the unbounded table. In this data abstraction, for the input data stream, an unbounded table called 'input table' is created and new data is appended to it continuously as new rows of data. A 'trigger time' is defined at which the new rows of data is appended to the input table, i.e. the input table grows at the re-occurrence of the trigger time. A query run on the input table generates a 'result table'. Hence, at every trigger, not only the input table grows but also the result table, thereby changing the output result set continuously. The mode of updating output supported are of three types: (i) complete mode: the entire updated result table is written as output, (ii) append mode: only the newly added rows to the result table since the last point of trigger are written as output and (iii) update mode: only the rows that were updated in the result table since the last point of trigger are written as output. The update mode differs from the complete mode as this outputs only the rows that were changed since the last point of trigger.

It is interesting to note that this data abstraction is incompatible with the Spark ML library because the incremental processing programming model of Spark Structured Streaming programming is not compatible with the Spark ML processing model, where repeated iterations are carried out on entire datasets.

Comparing different Spark data abstractions

Spark libraries have been built on different abstractions and support various data abstractions for interaction, as listed in Table 1. The core abstraction is RDD; the other libraries have added layers of abstraction on top of this core abstraction. Interoperability between libraries is supported in several cases, as illustrated in Table 2. There are cases where interoperability between different Spark libraries is not possible. For example, Spark Structured Streaming library cannot be used with the DataFrame APIs of Spark ML library. This poses a severe limitation to apply machine learning models on streaming data. On the other hand, Spark Streaming and Spark ML are naturally inter-operable because both are built on RDD abstraction. However, to apply a machine learning model on streaming data created using the Spark ML, we have to do several internal conversions. First, the streaming data represented in DStream data abstraction needs to

Table 2 Interoperability between different Spark data abstractions, as in [18, 57]

Target abstraction / Source abstraction	RDD	DStream	DataFrame	S. DataFrame
RDD	–	No	Yes	No
DStream	Yes	–	No	No
DataFrame	Yes	No	–	No
Streaming DataFrame	No	No	No	–

be converted to RDD, which is supported as indicated in Table 2. Then, the RDD data abstraction is converted into DataFrame so that the machine learning model can be applied as Spark ML uses DataFrame data abstraction. Spark ML introduces the concept of Pipelines. A Pipeline is a model to pack the stages of the machine learning process and produce a reusable machine learning model. Reusable models can be persisted and deployed on demand. The inherent RDDs in the streaming data represented via DStream data abstraction can be accessed either through the 'Transform' or 'ForEachRDD' method. Out of the two available methods, the 'Transform' method applies the user-defined transformation on RDDs and produces new DStream which is not our desired intention as we want a DataFrame to apply the machine learning model created from the Spark ML library. The 'ForEachRDD' is an action method that applies user-defined transformation and does not return anything as its purpose is to push data out of the run-time environment and can be used for this purpose. Abstracting the process of creating DataFrames from DStream using 'ForEachRDD' function can be abstracted from the user to automate such interconversion of data abstractions wherever necessary.

Spark APIs

Developers typically use a user-facing method invocation to achieve data transformations. These developer-facing methods are called as APIs. Spark has easy-to-use APIs which are intuitive and expressive for operating on large datasets, available in a wide range of programming languages like Scala, Java, Python and R. There are three sets of APIs in Spark which have been discussed below.

APIs using RDD-based data transformations As defined earlier, an RDD is an immutable distributed dataset partitioned across various nodes in the cluster which can be operated in parallel. The operations can be controlled with low-level APIs offering either *transformations* or *actions*. In the earlier days, Spark had low-level APIs solely making use of RDD-based data transformations. These low-level RDD-based APIs are typically used in scenarios where [58]: (i) low-level transformations, actions and control of dataset are necessary, (ii) the dataset to be analysed is unstructured, i.e. cannot be represented in relational format and (iii) data manipulation via functional programming constructs is preferred. It is interesting to mention that even now in Spark with the coming of new data abstractions and APIs based on them, the low-level RDD-based APIs have not lost their importance. *RDDs stand as a pillar of interoperability between other data abstractions since they are built on top of RDDs.* The user can easily move data between DataFrames/Datasets and RDDs via simple method calls.

DataFrame-based APIs DataFrame is also an immutable distributed dataset collection just like RDDs. However, here data is organised into columns just like in relational databases; which makes large scale processing of data easier by providing higher levels of abstractions and domain-specific language APIs across a wide range of programming languages.

Dataset-based APIs In newer versions of Spark, DataFrame APIs have been merged with Dataset APIs to provide unifying data processing capabilities across various libraries. As a result of this unification, developers need to learn a few concepts and work with a single high-level API. From Spark 2.0 onwards, Dataset has two distinct API characteristics: (i) a strongly-typed API and (ii) an untyped API.

From a programming perspective, DataFrame is a collection of generic objects, i.e. Dataset [row], where 'row' is a generic untyped Java virtual machine (JVM) object. In contrast to this, Dataset is a collection of strongly-typed JVM objects, i.e. Dataset [T], where 'T' is a defined class in Java/Scala. The unified API has several benefits [58]:

1. Static-typing and run-time safety: If we consider 'static-typing and runtime safety' to be a spectrum then string SQL query is the least restrictive while Dataset is the most restrictive in SparkSQL. To explain the previous statement, we consider an example. For instance, we cannot detect any syntax errors in a SparkSQL string query until runtime whereas in DataFrame and Dataset they can be detected during compile time. If a function in DataFrame is invoked which is not part of the API, the compiler detects this. Nevertheless, accessing and using a non-existent column name does not get detected until run time. On the other extreme side of the spectrum, is the Dataset, the most restrictive because all Dataset APIs are expressed as lambda functions and JVM typed objects. Any mismatch of the typed parameters gets detected at compile time. Here, even analysis errors can be detected at compile time, thereby saving development time.
2. Ease of use: Although Dataset/DataFrames render a structure which may limit or restrict what can be done with the data, it introduces a rich semantics and an easy set of domain-specific operations that can be expressed as high-level constructs. For example, it's much simpler to perform aggregations, selections, summations etc. operations by accessing a Dataset typed object's attributes than using RDD. Expressing computation in a domain specific API is more natural than with relation algebra type expressions offered in RDDs.

When to use DataFrame/Dataset If we require to have a rich semantics, high-level abstractions over datasets and easy to use domain-specific APIs, then DataFrame or Dataset APIs form a right candidature. Additionally, a higher degree of type-safety at compile time, high-level expressions, columnar access of data, unification and simplification of APIs across Spark libraries then DataFrame or Dataset APIs is a natural choice. Nevertheless, we can always use DataFrames and change back to RDDs whenever we need fine-grained control.

Selection of APIs for flow-based Spark programming

For supporting flow-based Spark Programming, the DataFrame APIs of Spark have been selected. The reasons for this are:

1. The RDD based APIs require user-defined functions to bring about data transformation, which is impractical to be used from an end-user tool. Additionally, it introduces substantial type checking requirements to be enforced manually for correct Spark programming.
2. Dataset APIs though offer the best in terms of detecting syntax errors and analytical errors at compile time; nevertheless, they depend on Dataset [T], where T is a predefined Java class. It is challenging to predefine classes for all possible kinds of datasets. Restricting this to specific use-cases would render the approach inflexible, non-generic and inextensible.
3. The DataFrame API provides columnar access to data, renders custom view on a dataset, introduces easy to use domain specific APIs, offers a vibrant abstraction and at the same time detects syntax errors during compile time. This fits the use-case of the flow-based programming model, i.e. mashup tools. The only missing feature is the detection of analytical errors during compile time, i.e. accessing a non-existent column in a DataFrame. This is reasonably easy to implement generically and has been described in "Conceptual approach for flow-based Spark programming" section.

Classification of Spark APIs

The various APIs found in different libraries of Spark have different method signatures and perform either data loading, data transformation or data writing out of Spark runtime environment. In this section, we classify the Spark APIs based on their functionality, i.e. input, transformation and action. Further, we sub-classify the transformation APIs based on their method signatures. This is useful to *create generic invocation statements to invoke the standalone method implementation of the APIs belonging to the same API category and operating on the same data abstraction.* This is pivotal to make the code generation process generic and make it independent from the Spark APIs as all APIs belonging to the same category and used in a graphical flow can be invoked in a similar generic way.

RDDs support only two kinds of operations [50] i.e. (i) *transformations* which help in data transformation and create a new dataset from an existing one, (ii) *actions* which return the result of data transformation after running computations to the Spark driver program. It is interesting to note that there is no notion to classify read operations, i.e. operations which read data into the Spark runtime environment. This is primarily because of Spark's lazy evaluation strategy. All transformation functions are evaluated in a lazy fashion, i.e. they remember the transformations applied on a file and are computed when the results are needed starting from reading the file. Since all APIs invariably work with RDDs internally irrespective of the higher data abstraction used. Therefore, all Spark APIs fall broadly into two categories, i.e. either 'transformation' or 'action.'

A typical Spark follows the model of reading data, analysing it and finally giving out the result. Here, we have classified various APIs of different Spark libraries using the notion of this logical data flow model. Accordingly, we have three categories, as discussed below.

Input A Spark application begins with identifying data sources and ingesting them into its run-time environment. APIs to read data from sources such as file systems and streaming sources are available out of the box. *'SparkSession'* class is the entry point to programming Spark with the Dataset or DataFrame APIs. This class provides two methods:

1. read () : *returns a 'DataFrameReader'* It creates a DataFrame.
2. readStream() : *returns a 'DataStreamReader'* It creates a Streaming DataFrame.

The *'DataFrameReader'* class provides methods to read datasets from external environments and represent them in DataFrame format. The various read methods (APIs) offered by this class fall into the *'input'* category as per the classification done in this work. The *'DataStreamReader'* class provides methods to read streaming datasets from external environments.

Similarly, the *'StreamingContext'* is the main entry point for using the streaming functionality of Spark, i.e. working with DStreams. It provides methods (APIs) to create streams and work with them.

Transformation 'Transformation' APIs transform user data within the run-time environment. Some data transformation APIs act on one data source while there are transformation APIs that act on two data sources. Additionally, there are other transformation APIs which prepare objects required by other data transformation APIs. The transformation APIs available in Spark have been classified into different sub-categories based on the number of data-sources they require as input to operate.

Type A: *These APIs operate on one data abstraction and produce a new data abstraction* They may or may not take additional parameters. The different types of transformations supported by Spark SQL, Spark ML and Spark Structured Streaming libraries include: (i) Static DataFrame Operations APIs of Spark SQL to produce a new DataFrame as per user-specified criteria, (ii) ML Transformers of Spark ML transform one dataset to another, (iii) Streaming Aggregations of Spark Structured Streaming add column to produce new Streaming DataFrame.

Type B: *These APIs operate on two data abstractions and produce a new data abstraction* If we need to operate on more than two data abstractions, then the APIs need to be invoked iteratively. Examples include: (i) DataFrame Join APIs of Spark SQL which produce a new DataFrame by joining two static DataFrames, (ii) DStream Join APIs of Spark Streaming which produce a new DStream by joining two DStreams, (iii) Streaming DataFrame Join APIs of Spark Structured Streaming which produce a new streaming DataFrame by joining two streaming DataFrames, (iv) Streaming and Static DataFrame Join APIs of Spark Structured Streaming which produce a new streaming DataFrame by joining streaming DataFrame and static DataFrame.

Type C: Spark libraries support APIs that *take one data abstraction in addition to other user-defined object parameters and produce an object such as a machine learning model or a pipeline model or maybe even a new data abstraction.* Such a reusable object can be applied on other data frames that match the schema of the dataset using which the reusable object was prepared. Example APIs include ML Estimators, ML Pipeline Model, ML Train-Validation Split Model and ML Cross-Validation Model provided by Spark ML. The general method signature of different types of transformation APIs is given below:

data abstraction df1.API() // Type A
data abstraction df1.API(user params) // Type A
data abstraction df1.API(data abstraction df2) //Type B
data abstraction df1.API(data abstraction df2, object userParams) // Type C

Type D: Other Spark APIs which do not follow the above method signature patterns have been classified as *Type D*.

Action These are APIs which trigger input APIs and transformation APIs associated with the data source and cause it to materialise. The lazy evaluation strategy of Spark requires action APIs for data transformations to materialise in the run time environment. Action APIs typically write to file systems or streaming sinks, i.e. push result out of the Spark run-time environment. Examples include (i) Spark Session Writer API of Spark SQL writes a static DataFrame to file system, (ii) Spark Session Stream Writer of Spark SQL writes a streaming DataFrame to a streaming sink like Kafka, (iii) ML Writer of Spark ML persists a ML model to the file system.

Which type of APIs are supported in the conceptual approach? The APIs have been categorised into either input, transformation—Type A, transformation—Type B, transformation—Type C and action. The underlying method signatures are similar for all APIs which fall in the same category except for transformation—Type D. Hence, the standalone method implementation of these APIs can be invoked by generic statements which correspond to the API category and the data abstraction used by the API. Every modular Spark component uses a generic method invocation statement for its standalone method invocation as listed in Table 3 and explained in "Generation of Spark driver program" section.

APIs not having common method signatures and which cannot be classified have been categorised as Type D. The standalone method implementation of such APIs cannot be invoked by a generic invocation statement and are not supported in this work.

Conceptual approach for flow-based Spark programming

The conceptual approach for programming Spark via graphical flows is presented in two parts:

Modelling of graphical Spark flows One of the primary assumptions is that end-users would typically follow the logical data flow model of reading, transforming and writing data out while specifying a Spark application. This idea is used to guide the user and enforce the sequence of connecting components such that every flow results into a compilable Spark driver program. The flow is captured and represented as a directed acyclic graph (DAG) [59], where *the start vertex represents data read operations, branches are pathways for data transformations, and end vertices represent data write operations.* Any vertex in a branch must be compatible with the schema produced by its immediate predecessor. Modelling of loops in the graphical flow is not supported because the data abstraction output from each component is immutable. Since Spark data abstractions like DataFrame, DStreams etc. are abstractions over RDD, so they are immutable like RDDs. The DAG can have multiple start vertices, since users can read datasets from two different IoT data sources, merge them and, then, run analytics on them. In short, the DAG stores the type of graphical components used by the user and also their positional information, both of which are necessary to generate a Spark application.

Method implementations of Spark operations that take a data abstraction schema as well as user parameters as input are maintained which make use of one or more Spark APIs to do the data transformation and return a modified data abstraction schema as the output. Every vertex in the DAG typically corresponds to one Spark operation and, thus, to one stand-alone method implementation.

It is interesting to mention here that, since the programming style, as well as the execution model of a Spark application, is different from those of actor-based dataflow models, which follow the flow-based programming paradigm and asynchronous message passing, additional auxiliary graphical components have been introduced to express a Spark program from mashup tools. These are typically used for enforcing a strict sequence of operations, e.g. when defining the order in pipeline operations of machine learning APIs or for bridging datasets, like in join or merge operations. These need to be preserved in the DAG as well. These vertices enforce a strict pathway of data transformation by overriding the asynchronous dataflow model and do not correspond to any Spark operation. Additionally, the code generation begins lazily, i.e. when the end vertex of the DAG is traversed.

The captured DAG is passed to a code generator, which first generates the necessary code skeleton for initialising a Spark session and then closes the session at the end of the application, to create the runnable Spark application. For the actual business logic of the flow, it wires the method implementations of Spark operations by providing the data abstraction schema and user parameters as inputs. The only requirements for this wiring process are that the data abstraction provided as an input is the same as the data abstraction of the output of the previous method, and the data abstraction schema must be compatible. By compatibility, it is meant that the data abstraction schema from one operation, for instance, a DataFrame, has the necessary columns which the receiving operation would make use of and the receiver expects the schema using the correct data abstraction, i.e. DataFrame.

Suitable data abstraction For expressing a Spark program, the most suitable *data abstractions*, i.e. DataFrame (including Streaming DataFrame) and DStream were selected. Users would typically drag different graphical components and wire them together in the form of a flow. The chosen declarative APIs require some input and produce predictable outputs. Hence, these are an ideal choice as wiring components. From the tool's perspective, the input is a compatible schema, and the output is a corresponding altered schema. In contrast to this, supporting data-transformations based on RDDs making use of user-defined functions (UDFs) would impose challenges that are difficult to solve generically. As every UDF has tight type requirements, this would introduce type validation problems from a tool's perspective.

Then, the scope of the graphical components has been defined. The graphical components do not represent a single Spark API but rather a set of APIs invoked in specific order to perform a specific data-analytic operation. Since representing every Spark API via a graphical component would involve wiring a large number of components for a Spark program, the aforementioned design attempts to balance a programming tool's usefulness and usability.

Composing a graphical Spark flow

This section discusses how the *conceptual approach* works in details, i.e. how to compose a graphical Spark flow, its validation to ensure such a flow generates a compilable Spark program. However, an important question arises here: *What kind of graphical components would support such an application model and how to develop such components?*. Figure 1 shows the high-level application model from a developer's perspective. It is evident from the figure that while the user focuses on joining various graphical components to specify the data transformation logic, the developer sees the same components as a bundled set of Spark APIs. These graphical modular components internally correspond to the fundamental execution component used in a mashup tool, for instance, these correspond to actors in aFlux which when instantiated invoke the set of Spark APIs and result in the generation of a complete Spark application program (*aka Spark driver program*).

The approach can be broadly classified into three essential parts dealing specifically with: (i) components, (ii) flow validation and (iii) application generation.

Components

A '*component*' is a basic unit of Spark flow composition. As discussed earlier, a component encapsulates a set of Spark APIs invoked in a specific order to perform a specific data analytics operation, e.g. reading data from a file or a streaming source like Kafka etc. A component is a critical element in the design since each component is a handle for the end-user to communicate the business logic as well as configuration information to the back-end. At the same time, a component serves to enforce the flow compositional

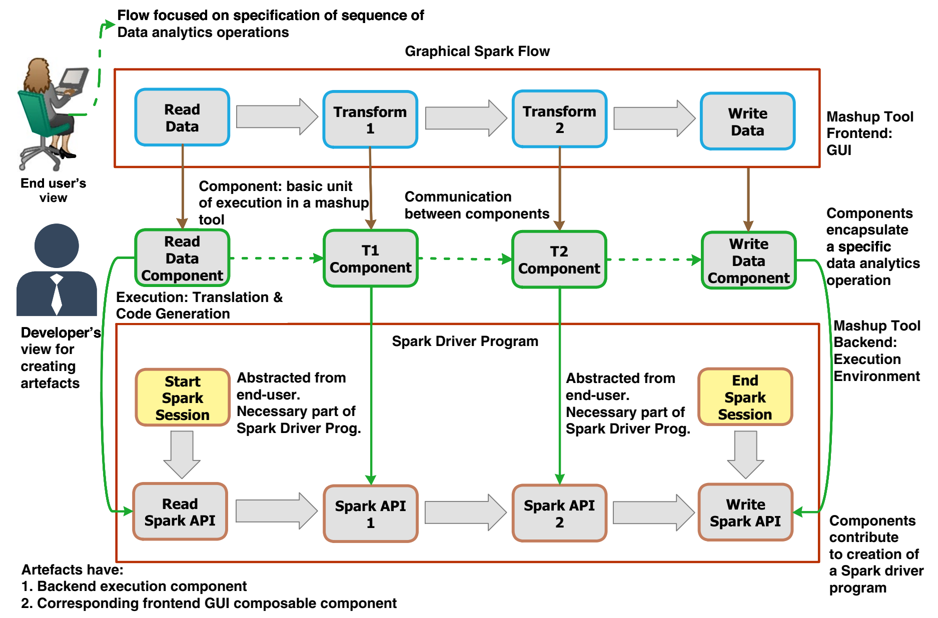

Flow focused on specification of sequence of Data analytics operations
Graphical Spark Flow
Read Data
Transform 1
Transform 2
Write Data
Mashup Tool Frontend: GUI
End user's view
Component: basic unit of execution in a mashup tool
Communication between components
Components encapsulate a specific data analytics operation
Read Data Component
T1 Component
T2 Component
Write Data Component
Execution: Translation & Code Generation
Developer's view for creating artefacts
Spark Driver Program
Mashup Tool Backend: Execution Environment
Start Spark Session
Abstracted from end-user. Necessary part of Spark Driver Prog.
Abstracted from end-user. Necessary part of Spark Driver Prog.
End Spark Session
Read Spark API
Spark API 1
Spark API 2
Write Spark API
Components contribute to creation of a Spark driver program
Artefacts have:
1. Backend execution component
2. Corresponding frontend GUI composable component

Fig. 1 Developer's perspective for building Spark components for mashup tools, as in [57]

rules so that the final resultant flow always produces a compilable and runnable Spark application. For supporting graphical Spark programming, these components have been classified into different categories, as explained below. These components are developed by the developer of the graphical flow-based programming tool and used by the user of such tools to program Spark graphically.

Categorisation of components: To support generic modelling of graphical Spark flows in mashup tools, we identified the different classes of APIs that exist within Spark in "Classification of Spark APIs" section. Accordingly, different categories of components are supported for graphical flow-based Spark programming. One specific challenge is dealing with multiple incoming connections to a component in a mashup tool. Since in flow-based programming paradigm [10] of mashup tools, every component executes on receiving a message from its preceding component and the order of messages is not preserved. However, in a Spark flow, some components which may receive more than one incoming connections require preservation of the order of messages received. For such scenarios, special components called *bridge* components have been designed to express the Spark sequenced operations in a flow-based programming paradigm.

There are basically *five component types* namely *input*, *transformation*, *bridge*, *action* and *executor*. Components of these types form the vertices in the DAG. Input components have no incoming connections, and they read data from external sources into the run-time environment. From the user's perspective, these start a Big-Data processing flow, i.e. they are the first components in flow, and other components consume their output.

From the mashup tool's perspective, they introduce the schema to be used by succeeding components in the flow. Transformation components are intermediate components and represent operations on ingested data; they consume as well as produce an output schema. Transformation APIs in Spark are designed to accept, at most, two compatible schema variants, which means they can accept, at the most, two incoming connections; they must also have at least one outgoing connection. *Bridge components* do not consume or produce schema. Accordingly, they do not correspond to any method implementation of Spark operation but are used to express the Spark sequenced operations ("Supporting sequenced Spark transformations in flow based programming paradigm" section) in a flow-based programming paradigm. For example, they can impose order in processing data coming from preceding transformations. There are two classes of Bridge components, namely '*Class A*' and '*Class B*'. Class A components are used to distinguish two incoming connections by annotating the connections with meta-data before being passed on to the next component. Class B components impose order from among any number of independent incoming transformation connections and assemble them into a sequence by preserving order. Action components allow the user to save the transformed data to external file systems or to stream it out to message distribution systems. Finally, an executor component collects all the incoming connections from multiple action components and adds abstractions related to the Spark driver program. This has all the data required to generate a Spark application after the executor component has been triggered.

Key attributes of a component: A component, which is the basic unit of a Spark flow has the following attributes:

(i) (Data abstraction) colour
denotes the data abstraction the component uses internally. It is used to identify the data abstractions of the encapsulated APIs within the components. Since different data abstractions are supported, a flow should consist of only components of the same colour to produce a compilable and runnable Spark application ("Components" section).

(ii) Component category
contains the unique category to which a component belongs, i.e. either input, action, transformation or bridge etc.

(iii) Unique name
is used to identify the component when captured from a front-end user flow and converted into an internal model for Spark application generation.

(iv) Configuration panel
allows users to supply the necessary parameter for its optimal functioning.

(v) Internal logic
refers to the core functionality of the component, i.e. how it takes the user-supplied parameters, generates an invocation statement to invoke the standalone method implementation of the underlying Spark APIs, prepares and sends a message to the next connected component in the flow.

Every Spark component can be viewed from two different perspectives, i.e. one from the end-user perspective of using it in a graphical flow representing a Big Data analytics operation along with a configuration panel and other as a component instantiated as an *executable-unit* in the run-time with pre-programmed attributes and embedded logic. Figure 2 shows the composition of a Spark flow using the modular components. From the Figure, it can be seen that the user specifies configuration via a configuration panel and specifies its incoming as well as outgoing connections. With that, the component receives messages from its predecessors and parses the messages. It adds its positional hierarchy data and sends the complete information to its successors and its specific information to be stored in the internal state. The positional hierarchy is also known as sequences in flow-based programming paradigm [10] which are to enforce the correct

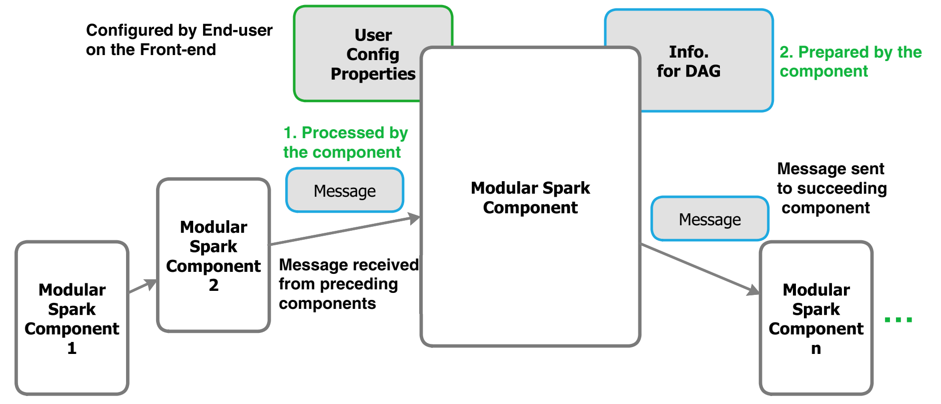

Configured by End-user on the Front-end
User Config Properties
Info. for DAG
2. Prepared by the component
1. Processed by the component
Message
Modular Spark Component
Message sent to succeeding component
Message
Modular Spark Component 2
Message received from preceding components
Modular Spark Component 1
Modular Spark Component n
...

Fig. 2 Composition of a Spark flow using the modular components, as in [57]

sequence of execution of components in a flow. In this context, the positional hierarchy determines if a specific component can be used in a specific position in a flow.

Hence, every component interacts with the end-user designing the flow and also contributes to creating an internal model of the user flow. All the components in a Spark flow are validated based on their position in the flow, i.e. *if they are allowed in a specific position in the flow, compatibility of the data abstractions and if the input schema is compatible with them.* For validation, the meta-data storing the category of every Spark component and its list of permissible predecessors is maintained. If the validation is passed, then, an internal model of the user flow is created from which the generation of a runnable Spark application proceeds.

Compositional rules: Based on the classification of components, a Spark flow needs to adhere to the following rules to generate a compilable and runnable Spark application:

1. Flow is unidirectional. Every branch in a flow begins with an input component, followed by one or more transformation components which must lead to one and only one action component.
2. Every flow must end with only one executor component, and each action component in the flow must be connected to the executor component.
3. Transformation components, which require incoming connections in a specific order, must be preceded directly by bridge component(s).
4. A component accepts an incoming connection(s) if and only if the schema derived from the incoming connection(s) is valid against schema checks. This means that a named column operated upon in a component is part of its incoming schema.
5. Each component internally uses one Spark data abstraction, which is represented by a uniform colour code. A flow composed of different coloured components, except for the executor component, is not accepted. This is done to support easy validation of a flow. For instance, in a streaming use case using DStream data abstraction, when we require to use DataFrame APIs for specific functionality like applying a ML model on streaming data which require interconversion of data abstraction, then the component houses the interconversion algorithms internally but presents the use case context-specific data abstraction for validation, i.e. DStream, since the flow starts with a data loader component for loading datasets from a streaming source using DStream abstraction.

Composability of components: *Composability* is a system design principle which deals with the interrelation between different components of a system [60]. A system is said to be composable when its different components can be assembled in various configurations to satisfy a user requirement [61]. The decisive criteria for components of a system to be composable are that they should be independent or self-contained and stateless [62]. A composable system is more natural to validate because of its consistency to achieve a certain goal [62].

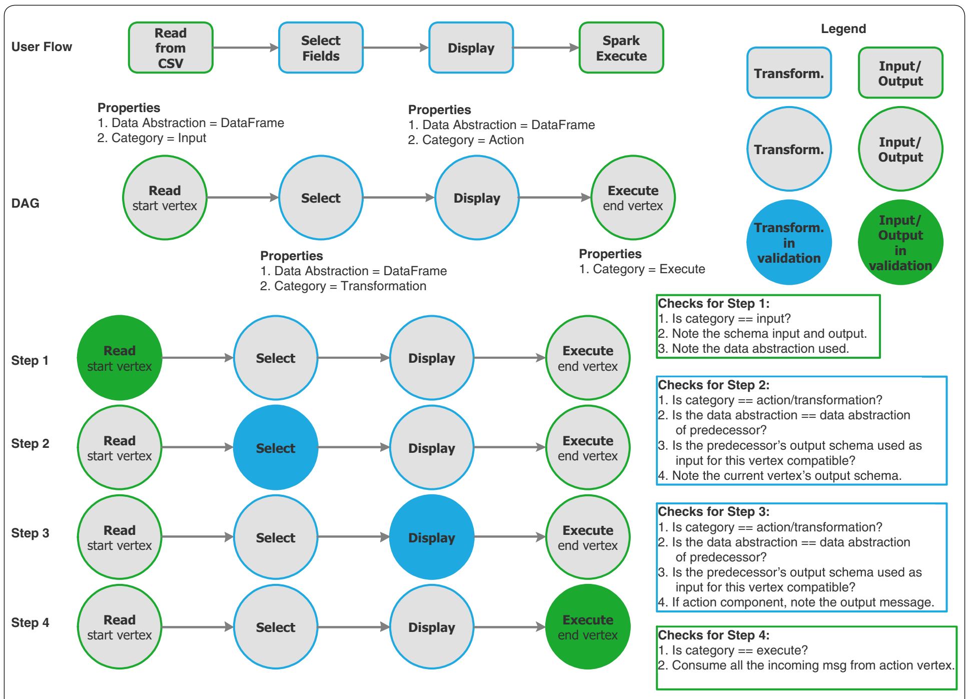
In the context of this work, we have outlined that the components designed are modular, i.e. self-contained and work on taking a data abstraction schema as input and produce a modified schema of data abstraction as output, i.e. stateless. *The graphical components when composed following the compositional rules enumerated above, such that they pass the flow validation to yield a compilable and runnable Spark program, are said to be composable.*

Every Spark component has a unique internal name and an associated standalone method implementation for an analytic operation. Meta-data of this information are maintained at the tool level. The code-generator receives the DAG as input and generates the required static code, e.g. starting a Spark session, inclusion of Java packages. Then, it checks the vertex in the DAG and determines its category. If it is an input or transformation vertex, then it uses the vertex's internal name to determine its associated method implementation. It calls the method via Java Reflection, passing the data abstraction schema and the vertex's user-supplied configuration values as parameters. This process is done iteratively for all vertices in the DAG until the code generator reaches the end vertices, which indicate actions and terminate the flow. Here, it calls the appropriate method to publish the data and closes the Spark session. The resulting Java file is compiled into a runnable Spark application and deployed on a cluster.

Validation of a Spark flow

Validation of a user created graphical Spark flow is the first step before generating the Spark application program. It is *to ensure that the components used in the flow use the compatible data abstraction and their positional hierarchy in the flow, i.e. a component can consume the output produced by its predecessor, ensures the generation of a compilable and runnable Spark application.* By validation of a flow, it is meant:

- The flow first loads the datasets then has transformation components, and the last components are the action components to return the output.
- The transformation components used after loading of datasets do not make use of any named column name which is missing in the schema of the loaded dataset, i.e.



Properties
1. Data Abstraction = DataFrame
2. Category = Input

Properties
1. Data Abstraction = DataFrame
2. Category = Action

Properties
1. Data Abstraction = DataFrame
2. Category = Transformation

Properties
1. Category = Execute

Checks for Step 1:
1. Is category == input?
2. Note the schema input and output.
3. Note the data abstraction used.

Checks for Step 2:
1. Is category == action/transformation?
2. Is the data abstraction == data abstraction of predecessor?
3. Is the predecessor's output schema used as input for this vertex compatible?
4. Note the current vertex's output schema.

Checks for Step 3:
1. Is category == action/transformation?
2. Is the data abstraction == data abstraction of predecessor?
3. Is the predecessor's output schema used as input for this vertex compatible?
4. If action component, note the output message.

Checks for Step 4:
1. Is category == execute?
2. Consume all the incoming msg from action vertex.

Fig. 3 Validation of a Spark flow

the first transformation component can consume the schema produced by its data loader component.

- The schema of the dataset modified by a transformation component can indeed be processed by the succeeding transformation component/action component.
- The components use a uniform data abstraction of Spark.

If the validation fails, then the user must correct the flow and re-deploy it. If a flow successfully passes the validation phase, it is sent to the next stage: i.e. representing the flow in an internal model to be used for Spark driver program generation. Figure 3 shows the validation for a flow which reads a CSV file, selects some fields and displays the results. Algorithm 1 shows the flow validation steps.

Algorithm 1: validation steps of a Spark flow

```
1 The user flow is checked for no cycles and represented as a topologically sorted DAG. If cycles are present,
  then the validation fails.
2 For the start vertex $v_1$, the following operations are performed:
    – Note the data abstraction: $v_1^{da}$.
    – Check the category: If $v_1^{cat} \neq input$, where $cat \in \{input, action, transformation, executor, bridge\}$, then the
      validation fails.
    – Note the input and output schema: $v_1^{s_{in}}$ and $v_1^{s_{out}}$, where
      $s_{in} = s_{out} \in \{DataFrame, DStream, StreamingDataFrame\}$.
    – Mark current vertex as 'visited'.
3 Traverse the next unvisited vertex except the end vertex. Perform the following checks:
    – Check the compatibility of data abstraction with its immediate predecessor's: if $v_i^{da} \neq v_{i-1}^{da}$, then the
      validation fails.
    – Check the category: If $v_i^{cat} \neq transformation$ or $v_i^{cat} \neq action$, then the validation fails.
    – If $v_i^{cat} = transformation$ and it specifies ordering of data-flow then a counter is started, i.e. $count = 1$
      – Process the next connected vertex together with current vertex, increase the counter, i.e.
        $count = count + 1$.
      – If $v_{i+1}^{cat} \neq bridge$, then the validation fails.
      – Mark current vertex as 'visited'.
    – Check compatibility with predecessor's schema: If $v_{i-1}^{s_{out}} \neq v_i^{s_{in}}$, then the validation fails.
    – Note the output schema: $v_i^{s_{out}}$.
    – Mark current vertex as 'visited'.
4 For the end vertex $v_n$:
    – Check the category: If $v_n^{cat} \neq executor$, then the validation fails.
    – Check category of its connected predecessor: If $v_{n-1}^{cat} \neq action$, then the validation fails.
```

Generation of the internal model

Once the validation process has been successful, the flow is deployed in the runtime environment; therefore an execution unit of the implementing tool is instantiated for every graphical component used in the flow, and the execution follows starts from the first component of the flow. In the runtime environment, as the components execute, a DAG is regenerated. However, this DAG not only captures the user flow, but additional information is attached to every vertex during its creation, similar to the intermediate code generation step in case of compilers [59].

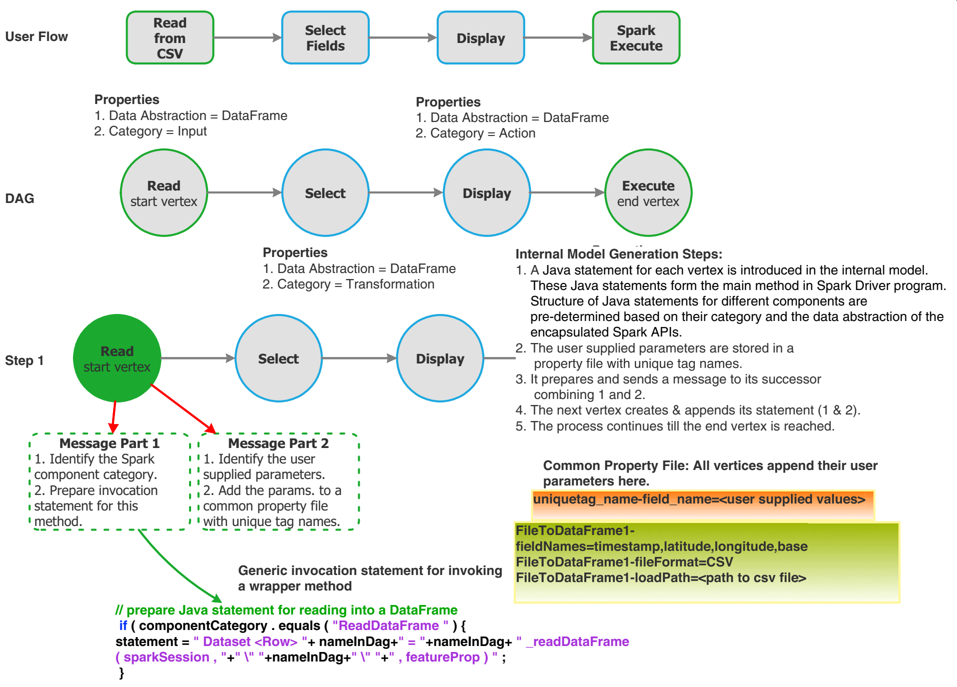
Message passing The components on execution rely on message passing. This message contains how to invoke the corresponding component's standalone method implementation of Spark APIs and the list of user parameters that should be passed while invoking it. The message also helps all successors to know the output of their predecessor and in turn create their invocation statements, append to the message and pass it on. Therefore, every vertex during its execution creates a message for all its connected successors, which has two distinct parts:

> Message part 1: (contains the generic method invocation statement): The component in its '*internal logic*' has the necessary code to check the *category* of the component. Accordingly, Java statements to invoke the standalone method implementation of the encapsulated Spark APIs via reflection [63, 64] are added for every vertex. These Java statements form the main methods in Spark Driver program. The structure of Java statements for different components is pre-determined based on the category of the component, i.e. whether they represent input, transformation, or action components ("Classification of Spark APIs" section) and the data abstraction used, i.e. DataFrame, DStream or Streaming DataFrame. These Java statements form the first part of the message.
>
> Message part 2: (contains the list of user-supplied parameters for the component): The component checks the user-supplied parameters and creates a property file [65] to store them. It uses the unique name of the component, as every component has one as its essential attributes, and the property name to create a tag for every user parameter. This assumes the following form: <*uniqueName-fieldName=user-supplied values*>. The first component creates the property file, and other components in the flow use the same file to append their values. The second message contains the path of the property file and its unique tag to access its user-supplied parameters.

The component combines both message parts and stores them in the vertex of the DAG, and it also sends as a message to all its successors. The successor components use this information to create their message, store it, as well as append their message to the original message and pass it on. The last component which is the '*executor*' component adds Java statements to create a Spark session within which other Spark operations can be invoked and also create a generic statement to access the user-supplied parameters from the property file. Lastly, it assembles all the Java statements of all the components of the flow to create a combined list of Java statements to invoke the generic method implementation of the Spark operations represented by the individual components in the flow. Figures 4, 5, 6 illustrate the steps of the internal model generation of a Spark flow. The example uses a flow which reads data from a CSV file, selects some fields of data and displays them. In Fig. 6, the final assembled list of Java statements is shown in the form of a green-coloured box which is the *internal model representation of the user created Spark flow.*

Generation of Spark driver program

During the creation of *internal model* representation of a user flow, each component used in the flow creates an invocation statement to invoke its corresponding generic method implementation by passing the user supplied configuration as parameters. After



Properties
1. Data Abstraction = DataFrame
2. Category = Input

Properties
1. Data Abstraction = DataFrame
2. Category = Action

Properties
1. Data Abstraction = DataFrame
2. Category = Transformation

Internal Model Generation Steps:
1. A Java statement for each vertex is introduced in the internal model. These Java statements form the main method in Spark Driver program. Structure of Java statements for different components are pre-determined based on their category and the data abstraction of the encapsulated Spark APIs.
2. The user supplied parameters are stored in a property file with unique tag names.
3. It prepares and sends a message to its successor combining 1 and 2.
4. The next vertex creates & appends its statement (1 & 2).
5. The process continues till the end vertex is reached.

Message Part 1
1. Identify the Spark component category.
2. Prepare invocation statement for this method.

Message Part 2
1. Identify the user supplied parameters.
2. Add the params. to a common property file with unique tag names.

Common Property File: All vertices append their user parameters here.

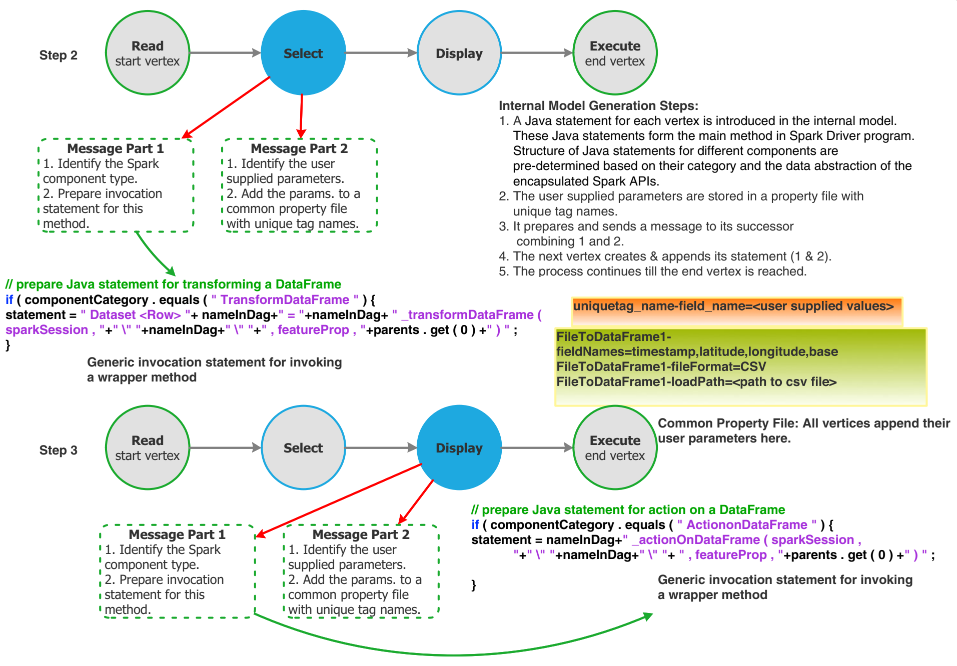
```
uniquetag_name-field_name=<user supplied values>
```

```
FileToDataFrame1-
fieldNames=timestamp,latitude,longitude,base
FileToDataFrame1-fileFormat=CSV
FileToDataFrame1-loadPath=<path to csv file>
```

Generic invocation statement for invoking a wrapper method

```
// prepare Java statement for reading into a DataFrame
 if ( componentCategory . equals ( "ReadDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _readDataFrame
( sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp ) " ;
 }
```

Fig. 4 Internal model representation of a Spark flow: part 1



Internal Model Generation Steps:
1. A Java statement for each vertex is introduced in the internal model. These Java statements form the main method in Spark Driver program. Structure of Java statements for different components are pre-determined based on their category and the data abstraction of the encapsulated Spark APIs.
2. The user supplied parameters are stored in a property file with unique tag names.
3. It prepares and sends a message to its successor combining 1 and 2.
4. The next vertex creates & appends its statement (1 & 2).
5. The process continues till the end vertex is reached.

Message Part 1
1. Identify the Spark component type.
2. Prepare invocation statement for this method.

Message Part 2
1. Identify the user supplied parameters.
2. Add the params. to a common property file with unique tag names.

```
// prepare Java statement for transforming a DataFrame
if ( componentCategory . equals ( " TransformDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrame (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
```

Generic invocation statement for invoking a wrapper method

```
uniquetag_name-field_name=<user supplied values>
```

```
FileToDataFrame1-
fieldNames=timestamp,latitude,longitude,base
FileToDataFrame1-fileFormat=CSV
FileToDataFrame1-loadPath=<path to csv file>
```

Common Property File: All vertices append their user parameters here.

Message Part 1
1. Identify the Spark component type.
2. Prepare invocation statement for this method.

Message Part 2
1. Identify the user supplied parameters.
2. Add the params. to a common property file with unique tag names.

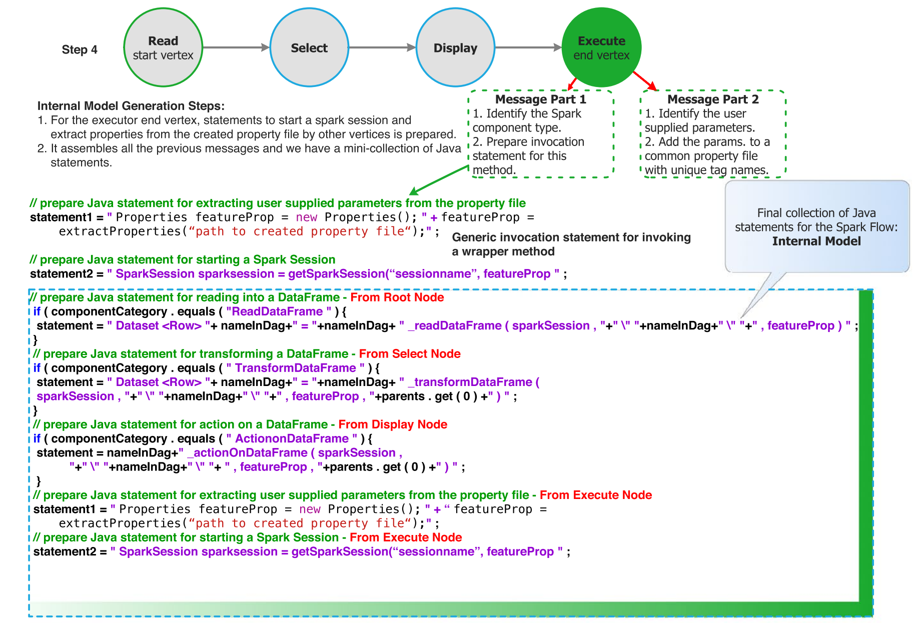
```
// prepare Java statement for action on a DataFrame
if ( componentCategory . equals ( " ActiononDataFrame " ) {
statement = nameInDag+" _actionOnDataFrame ( sparkSession ,
        "+" \" "+nameInDag+" \" "+ " , featureProp , "+parents . get ( 0 ) +" ) " ;

}
```

Generic invocation statement for invoking a wrapper method

Fig. 5 Internal model representation of a Spark flow: part 2

the preparation of the *internal model* of a Spark flow, it can be used to generate a complete Spark driver program. In the internal model, each vertex has contributed a Java statement, and all these have been assembled by the last vertex of the DAG, i.e. which


Step 4
Read start vertex
Select
Display
Execute end vertex
Message Part 1
1. Identify the Spark component type.
2. Prepare invocation statement for this method.
Message Part 2
1. Identify the user supplied parameters.
2. Add the params. to a common property file with unique tag names.
Generic invocation statement for invoking a wrapper method
Final collection of Java statements for the Spark Flow: Internal Model

Internal Model Generation Steps:
1. For the executor end vertex, statements to start a spark session and extract properties from the created property file by other vertices is prepared.
2. It assembles all the previous messages and we have a mini-collection of Java statements.

```
// prepare Java statement for extracting user supplied parameters from the property file
statement1 = " Properties featureProp = new Properties(); " + featureProp =
    extractProperties("path to created property file");" ;

// prepare Java statement for starting a Spark Session
statement2 = " SparkSession sparksession = getSparkSession("sessionname", featureProp " ;

// prepare Java statement for reading into a DataFrame - From Root Node
if ( componentCategory . equals ( "ReadDataFrame " ) {
 statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _readDataFrame ( sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp ) " ;
}
// prepare Java statement for transforming a DataFrame - From Select Node
if ( componentCategory . equals ( " TransformDataFrame " ) {
 statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrame (
 sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
// prepare Java statement for action on a DataFrame - From Display Node
if ( componentCategory . equals ( " ActiononDataFrame " ) {
 statement = nameInDag+" _actionOnDataFrame ( sparkSession ,
       "+" \" "+nameInDag+" \" "+ " , featureProp , "+parents . get ( 0 ) +" ) " ;
 }
// prepare Java statement for extracting user supplied parameters from the property file - From Execute Node
statement1 = " Properties featureProp = new Properties(); " + " featureProp =
    extractProperties("path to created property file");" ;
// prepare Java statement for starting a Spark Session - From Execute Node
statement2 = " SparkSession sparksession = getSparkSession("sessionname", featureProp " ;
```

Fig. 6 Internal model representation of a Spark flow: part 3

corresponds to the execute component of the user flow. This collection of statements become the statements within the main method of the Spark driver program. *API based code generator* is supplied with these prepared statements for producing a Spark application in Java programming language [66].

To make the code generation process as generic as possible, generic method implementations making use of one or more Spark APIs are maintained. These are called as '*wrapper methods*'. All such '*wrapper methods*' are bundled into a library called '*Splux*'. Each component used by the user in a flow on instantiation parses the user supplied configuration values and appends them to a property file and prepares an invocation statement to invoke the corresponding wrapper method in Splux. The collection of all such invocation statements contributed by various components of the flow are used in the main method of the Spark driver program which essentially invokes the wrapper method during execution passing it the user supplied configuration values as parameters. The invocation statement for every component is pre-determined based on the category of the component, i.e. whether they represent input, transformation, or action components ("Classification of Spark APIs" section) and the data abstraction used, i.e. DataFrame,

Table 3 Example: mapping of component category → Spark API type, as in [57]

SN	Component class	Component category	Spark API classification
1	ReadDataFrame	Input	Input
2	ReadDStream	Input	Input
3	TransformDataFrame	Transformation	Transformation Type A
4	TransformDStream	Transformation	Transformation Type A
5	TransformDataFrames	Transformation	Transformation Type B
6	ActionOnDataFrame	Action	Action

DStream or Streaming DataFrame, thereby forming the '*component class*' (Table 3). The *wrapper methods* contained in *Splux* are *invoked via reflection.* From an implementation perspective, JavaPoet [67], an API based code generator, is used to assemble the reflection statements to generate a Spark driver program.

SparFlo: a subset of Spark APIs bundled as modular components

Spark offers many different libraries in its ecosystem accessible either via the RDD-based programming approach or invocation via APIs working on different data abstractions like DataFrame, DStreams etc. The manual development of Spark application involves interaction with these elements. To support the development of Spark applications via graphical flow-based programming, more abstraction is necessary. With the approach described in the paper, a subset of Spark APIs operating on DataFrame and DStream abstractions have been selected which are compatible with the flow-based programming paradigm.

'SparFlo' is a library consisting of modular ("Towards flow-based Spark programming" section) composable ("Components" section) components. The components in *SparFlo* bundle a set of Spark APIs from the selected subset of Spark APIs which are executed in a specific order to perform one data analytic operation. By modularity, we mean that every component representing a set of Spark APIs has everything necessary within it to achieve the desired functionality and is independent of other components, i.e. in this context perform one data analytics operation by taking only some user-supplied parameter as input for customisation of the data analytics operation. These modular components are composable when composed confirming to the flow compositional rules discussed in "Components" section. The components of SparFlo can then be expressed as compositional units in a mashup tool, for instance, they have been expressed as actors in aFlux.

Additional *compositional units* are sometimes necessary to develop graphical flow-based Spark programs using the SparFlo components. This happens when the execution semantics of the implementing graphical flow-based tool follows a non-sequential execution model, for instance, *bridge components* have been developed to express pipelined operations during a machine learning model creation in aFlux ("Supporting sequenced Spark transformations in flow based programming paradigm" section). Such additional compositional units do not form part of SparFlo but are implementation specific. This is because if the implementing tool follows a sequential flow-based programming paradigm, then additional compositional units may not be necessary. In short, *SparFlo is a component library using a subset of Spark APIs which can be easily integrated into graphical tools based on the flow-based programming paradigm.*. Figure 7 illustrates the elements and characteristics of SparFlo.

Extensibility: Inclusion of new/forthcoming Spark APIs A new/forthcoming Spark API can be modelled as a SparFlo component to be used in a flow-based tool like aFlux and auto-generate a runnable Spark program using the approach described in "Conceptual approach for flow-based Spark programming" section, *if and only if*:

- The new Spark transformation is *accessible via an untyped API.*
- It *must use either the DataFrame, DStream or Streaming DataFrame data abstraction.*

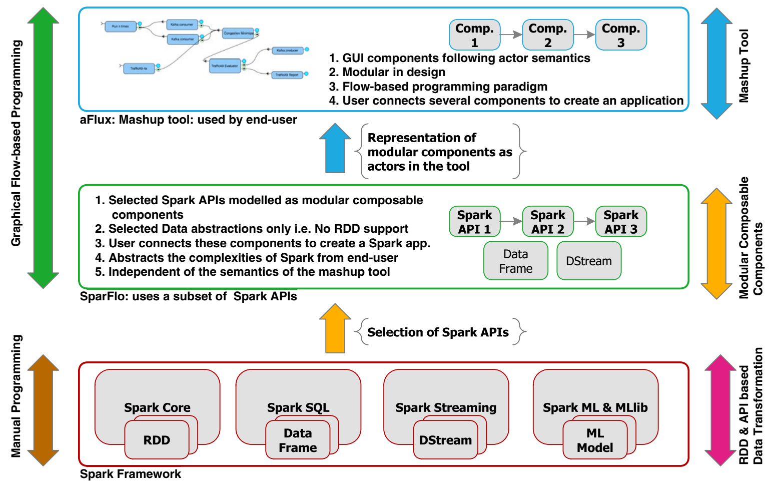

Graphical Flow-based Programming
Manual Programming
Comp. 1
Comp. 2
Comp. 3
1. GUI components following actor semantics
2. Modular in design
3. Flow-based programming paradigm
4. User connects several components to create an application
aFlux: Mashup tool: used by end-user
Mashup Tool
Representation of modular components as actors in the tool
1. Selected Spark APIs modelled as modular composable components
2. Selected Data abstractions only i.e. No RDD support
3. User connects these components to create a Spark app.
4. Abstracts the complexities of Spark from end-user
5. Independent of the semantics of the mashup tool
Spark API 1
Spark API 2
Spark API 3
Data Frame
DStream
SparFlo: uses a subset of Spark APIs
Modular Composable Components
Selection of Spark APIs
Spark Core
RDD
Spark SQL
Data Frame
Spark Streaming
DStream
Spark ML & MLlib
ML Model
Spark Framework
RDD & API based Data Transformation

Fig. 7 SparFlo: A library of modular composable components using a subset of Spark APIs, as in [57]

For example, to model a new component called *'StreamDFRunningAverage'* in SparFlo and implement it in aFlux which supports running average computations in Spark Structured Streaming, the following analysis, divided into two broad categories, are to be undertaken from a developer's perspective:

API analysis: Analysis of the APIs to support such a transformation component, i.e. if it has an untyped API and its internal data-abstraction is either DataFrame, DStream or Streaming DataFrame. In this case, this transformation uses Streaming DataFrame and has an untyped API.

Component analysis: To support a new component, its category needs to be determined, i.e. whether its an input, transformation or action component. In this case, it is a transformation component. A list of valid predecessors and successors for the transformation component needs to be determined for supporting validation when the component is used in a flow.

After the analysis, the development activities for supporting the component within aFlux are again divided into two categories as below:

Development of a wrapper method: Once the Spark APIs for transformation have been identified, the Splux library containing generic method implementation of all Spark APIs needs to be extended. A new generic method implementation for the new Spark API to execute running average operation on streaming data in Spark run-time environment should be added which can be invoked by reflection technique during auto-generation of Spark driver program in aFlux.

Component development: In the case of aFlux, a new actor representing the transformation component needs to be implemented with the following structural definitions:

- Component category: This component belongs to *transformation* category of aFlux components as defined in "Composing a graphical Spark flow" section. The category helps decide the positional hierarchy of the component in a flow which is required for validation of the flow. Since this is a transformation component hence, this can neither be the first component nor the last component in a flow.
- Colour: The component should have the colour code representing the data-abstraction it uses internally. Here, it's colour code should permit the user to know that it uses Streaming DataFrame internally.
- Mapping to API type: This component uses Spark APIs which belong to '*Transformation Type A*' class of APIs. Hence, its mapping to API type is *'TransformDataFrame'* as indicated in Table 3 which would determine its generic invocation statement used in the internal model to invoke its corresponding wrapper method.
- Cardinality of incoming connections: The number of incoming connections for this component is one. In case, it is more than one; then an internal timer needs to be implemented to wait for all messages from the component's predecessors to arrive before starting the execution.
- Message/schema checks: The component which is an executable-unit must include checks to ascertain the schema received as a message from its predecessor is valid, and it can work upon it to bring about a successful transformation.
- Representation in the internal model: Since '*StreamDFRunningAverage*' is a transformation component, hence it must contribute to the internal model by producing a unique name and adding the generic Java statement to invoke its wrapper method via reflection.
- User configurable properties: All configurable parameters of the API are made available to the user for customisation.

With this approach, any new API of Spark fulfilling the conditions listed can be added to SparFlo and programmed in mashup tools via the approach described to auto-generate a compilable and runnable Spark driver program.

Implementation

Based on this conceptual approach, Spark SQL, Spark ML, Spark Structured Streaming components have been prototyped in aFlux based on the DataFrame data abstraction and Spark Streaming components based on the DStream data abstraction. Special components called 'bridge' components have been built to support SparFlo modular, composable components. Additionally, from an implementation perspective, the component's have specific key properties which are vital for their functioning.

Supporting sequenced Spark transformations in flow based programming paradigm

In a flow-based programming paradigm, the control flows from one component to another from start to end as connected in the flow. In an actor-model based flow programming paradigm, each component reacts on receipt of a message in its mailbox and passing its output to the next connected component. The messages received by a component in its mailbox are processed one at a time. This form of execution-style is not conducive to support specific Spark operations which must be executed in a sequence. Each actor has one only mailbox, and all messages from other connected components arrive in the same mailbox, and all messages are processed with equal priority in the order of their arrival. Special handling is required in case of multiple incoming connections from components belonging to the same component category which require ordering. Special components called *'bridge components'* are used to order the arrival of messages in an actor's mailbox so that they are processed in a sequence amicable to the way Spark APIs are invoked when used via manual programming. The bridge components are of two kinds:

Class A: These components are used to annotate the messages coming from two different pathways to indicate the order of processing. A typical example is joining data from two branches, which introduce two incoming connections with an equal or different priority. Spark join operations are similar to SQL joins where only in the case of an inner join, the two data sources have equal priority, i.e. applying join on dataset 1 with dataset 2 or vice-versa always yields the same result. However, in case of an outer join, the position in the join is necessary for the operation, i.e. the order of processing is vital to perform either join on dataset 1 with dataset 2 or vice-versa. The *'PrepareJoin'* component collects messages from two incoming connections and annotates them with the order of processing before passing it on to the component where the actual join operation is performed i.e. the *'JoinDF'* component.

Class B: These components impose order in the execution of many transformation connections and assemble them into a sequenced operation. A typical scenario would be the creation of a machine learning model which follows a specific sequence of operations. Accordingly, the *'PreparePipeline'* component accepts incoming connections and arranges them in a specific order as specified by the end-user.

Figure 8 illustrates the working of the two different kinds of bridge components.

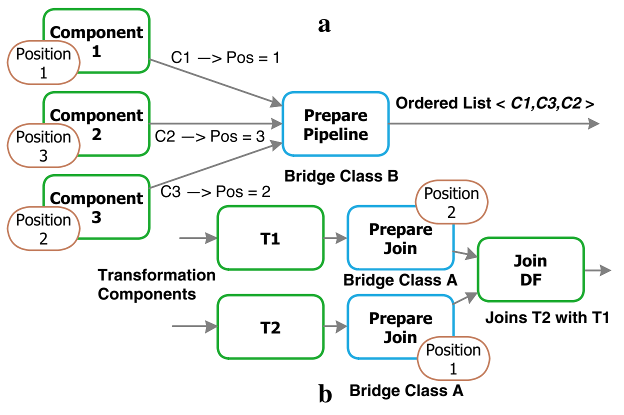


Fig. 8 Bridge components: working details, as in [57]

Components: key properties

A component which is internally instantiated into an actor has some properties defined by the developer in addition to encapsulating a data analytics operation in the form of a set of Spark APIs. These properties are necessary for correct working of components. These are not visible to the user of the components while designing a graphical Spark flow. These properties include:

Category: Every component belongs to a *category* as defined in "Composing a graphical Spark flow" section which defines its positional hierarchy in a Spark flow. For example, incoming connections to a component belonging to *action* category can only come from components belonging to either *transformation* or *input* category.

Mapping to API type: Every component maintains a mapping between its category and the type of Spark API it represents as listed in Table 3. This is also known as the class of the component. This is essential in the creation of the *internal model* representation of a user flow i.e. creation of invocation statements which follow a generic style specific to every Spark API type. Spark APIs belonging to input, action, transformation—Type A, transformation—Type B and transformation—Type C have a corresponding class of implementing components. Since Spark APIs belonging to transformation—Type D do not have common method signature and therefore difficult to frame generic invocation statements without compromising the generalisability of the approach are not supported. Therefore, no component class maps to transformation—Type D of Spark APIs.

Cardinality of incoming connections: The cardinality of incoming connections for every component is a feature inherited from the component category. Those components which accept more than one incoming connections must implement a timer to wait for all the incoming messages to arrive before starting their execution since actor model relies on asynchronous message handling. Some of the components require proper ordering of messages before processing them. For instance, the '*PreparePipeline*' component of bridge class B category which accepts more than one incoming connection during its execution implements a timer to wait for all incoming messages to arrive before processing them.

List of valid predecessors: Every component has a list of predecessors making use of the same data abstraction of Spark. This is vital to enforce the flow compositional rules while using a component i.e. deciding the correctness of its positional hierarchy in a flow. The *'list of valid predecessors'* ensure that the component can accept the schema of its preceding component and its output produces a schema acceptable for its successor components in the flow.

Message content: The message received by every component from its predecessor must be complete and sufficient for its processing needs. For instance, the '*PreparePipeline*' component needs the positional hierarchy of all its predecessors before starting its execution. Hence, all valid predecessors of '*PreparePipeline*' component must include this vital information in its output message.

Message/schema checks: Since, the message received by every component is a modified schema therefore a component must perform some schema checks to ensure it is indeed correct and processing it will not lead to generation of a message incompatible with its successors or lead to execution failures. Essential conditions for the incoming schema/message must be defined within each component in accordance to the specifications of the Spark API which the component is representing. For instance, the '*FeatureAssembler*' component makes use of the '*VectorAssembler*' API from Spark ML library. VectorAssembler combines two or more features of numeric SQL data-types i.e. either an integer, a double or a float to produce a new field of vector data type. This attribute does not apply to *bridge* components as they typically impose ordering of connections and do not alter and produce schema.

Contribution to internal model: Contribution to internal model from a component occurs in the form of adding an invocation statement to invoke the standalone method implementation of the data analytic operation it represents. Apart from bridge components, every other component contributes to the internal model.

Experimental

The evaluation scenario has been designed to capture the *modularity of the approach*, *code-abstraction from end-user*, *automatic handling of Spark session initialization code*, *interconversion of data between different data abstractions of Spark* as well as ease of creating quick Spark jobs by *providing high-level abstraction via graphical flow-based programming*. Here, an example of taxi fleet management with three use cases has been considered: (i) producing a machine-learning model to learn traffic conditions, (ii) applying the model to streaming data to make decisions, (iii) performing aggregations on streaming data. In all three use cases, the case of manually programming them in Java has been compared with the specification of graphical flows using Spark components of aFlux.

Evaluation scenario The dataset in a traffic scenario consists of information that was published by a vehicle at the beginning of a new trip. The dataset contains three elements: time-stamp, latitude and longitude. The time-stamp records the time at which a new trip commenced; latitude and longitude identify the geographic coordinates from where the new trip commenced. The goal is to devise a machine-learning based rush-hour fleet management solution to reduce waiting time for customers using Spark. Machine learning is employed to partition the city into sectors using historical data. Thus, the model prepared remains on the disk, which is then applied to real-time streaming data. Finally, stream aggregations, such as window count and running count based on event time, are applied to streaming data to receive real-time updates. The idea is to demonstrate how users can develop Spark applications for three different Spark libraries via aFlux vis-a-vis programming the same solution manually. Development of a Spark application consists of identifying relevant Spark libraries and using relevant APIs to build the solution. For example, a KMeans Algorithm is a good choice for identifying the trip start hotspots.

In the dataset, <*latitude,longitude*> can be used as features for training a KMeans algorithm. The model trained on historical data should be applied to real-time data. The Pipeline

API from the Spark ML library is a good choice for building a re-usable model which can persist on external file systems. Since Spark ML is built on the Spark SQL engine, using DataFrame API is the natural choice for this application. Spark libraries, which handle streaming data, support applying persisting models on real-time data and support event-time based window aggregations on streaming data. Spark Structured Streaming is a good choice for performing aggregations as it supports event-time based windowed processing. However, the programming model of Spark Structured Streaming is not compatible with Spark machine learning libraries. Hence, Spark Streaming must be used to apply the created model to real-time data and Spark Structured Streaming must be used to perform aggregations.

Therefore, we selected: (i) Spark ML for developing re-usable K-Means Model, (ii) Spark Streaming for applying the model on real-time data and (iii) Spark Structured Streaming: for applying aggregations on real-time data.

Use Case 1: Batch processing: producing a machine learning model

To understand the approach and how it provides advantages, we begin by devising a Spark application via a manual programming approach first. The relevant Spark libraries for building an application which produces a machine learning model from datasets have been identified and has been described in the following section.

Approach: Manual programming via Java

The Spark application built using Spark ML chiefly consists of the following stages:

UC1-S1 creating application context: A batch processing Spark application begins by first initialising a Spark session. The Spark session is the handle for Spark driver to orchestrate different tasks of the application like reading datasets, analysing them and returning results of analytics etc. Once the Spark session has been created, data is read into the Spark run-time environment using the session handle.

UC1-S2 reading datasets: Data is read using Reader function of the SparkSession. DataFrame API views data as a table. Hence, the schema must be supplied to it. Data in CSV format is read in this particular example.

UC1-S3 data transformations: Next, ML algorithms are applied as data transformations to produce a KMeans model fitted on the dataset. First, the data is prepared for processing. '*latitude*' and '*longitude*' are selected as features for the training of the KMeans algorithm. VectorAssembler API creates a field of Vector Data type out of one or more fields present in the DataFrame. Pipeline API is used to prepare a data analytics sequence: VectorAssembler for data preparation followed by KMeans for data analysis. The name *'features'* has been chosen as the name for feature Vector and *'prediction'* as the name for the field produced by the KMeans algorithm. PipelineModel writer API has been used to persist the model on the file system for later application.

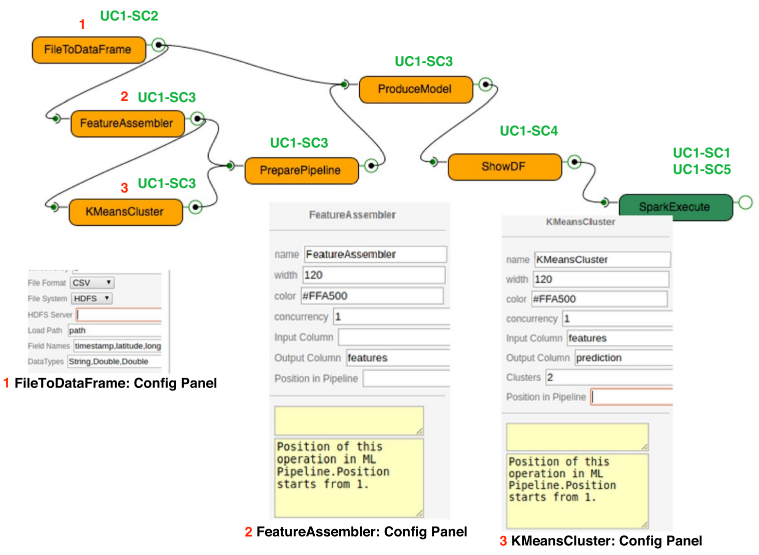
UC1-S4 invocation of actions: The next thing is to display the transformed data. Transformation on the dataset has been invoked using transform API. This transformation is executed only when an action API is invoked on the transformed DataFrame. Show API has been used to display the transformed data on the console.

UC1-S5 terminating application context: Lastly, the Spark driver is terminated by invoking stop functionality of Spark session.

Approach: Graphical Programming via aFlux

The approach via aFlux enables the user to create the same Spark program by connecting graphical components, leading to auto-generation of the Spark driver program. Figure 9 illustrates an aFlux flow for producing a machine learning model using Spark ML library of Spark. The components used are: (i) the 'FiletoDataFrame' component, which has only one output port and no input port. This component belongs to *input* category and encapsulates input category of Spark APIs as classified in "Classification of Spark APIs" section, (ii) the 'FeatureAssembler' component which has one input as well as one output port. This belongs to *transformation* category and encapsulates transformation—Type B category of Spark APIs, (iii) the 'KMeansCluster' component, which has one input as well as one output port. This belongs to *transformation* category and encapsulates transformation—Type A category of Spark APIs, (iv) the 'PreparePipeline' component which has two input ports and one output port. This belongs to *bridge* category and does not encapsulate any Spark APIs, (v) the 'Produce Model' component, which has two input and one output port. This belongs to *transformation* category and encapsulates transformation—Type C category of Spark APIs, (vi) the 'ShowDF' component, which has only one input port and no output port. This component belongs to *action* category and encapsulates action category of Spark APIs and (vii) the 'Spark Execute' component to mark the end of the flow to begin flow validation and code generation. The wiring of components is in correspondence to the stages described in manual programming:

> UC1-S1 creating application context: The last aFlux component 'Spark Execute' marks the end of the graphical flow. Its encounter in the flow conveys special meaning to the translator, i.e. to generate the Spark session initialisation, as well as the termination codes necessary for the Spark application.
>
> UC1-S2 reading datasets: The first aFlux component in the flow, i.e. 'FiletoDataFrame' has a configuration panel where the user can specify the file type, its location,



File Format CSV
File System HDFS
HDFS Server
Load Path path
Field Names timestamp,latitude,long
DataTypes String,Double,Double

FeatureAssembler
name FeatureAssembler
width 120
color #FFA500
concurrency 1
Input Column
Output Column features
Position in Pipeline

Position of this operation in ML Pipeline.Position starts from 1.

KMeansCluster
name KMeansCluster
width 120
color #FFA500
concurrency 1
Input Column features
Output Column prediction
Clusters 2
Position in Pipeline

Position of this operation in ML Pipeline.Position starts from 1.

Fig. 9 Spark flow in aFlux for producing machine learning model, as in [57]

and what fields to read. This component abstracts away the code necessary to read the file and to convert it to a DataFrame.

UC1-S3 data transformations: The 'FeatureAssembler' and 'KMeansCluster' components can be used to prepare data and apply a machine learning algorithm on the prepared data. The component 'FiletoDataFrame' has been wired up with the 'FeatureAssembler' component which in turn has been wired up with 'KMeansCluster'. Since a pipeline needs to be prepared with these components, they both have been wired to 'PreparePipeline' component. In the configuration panel of both 'FeatureAssembler' and 'KMeansCluster', their respective position in the pipeline must be specified. Figure 9 illustrates the configuration panels of 'FeatureAssembler' and 'KMeansCluster'. Finally, the model is prepared with the 'Produce Model' component, and the result is saved.

UC1-S4 invocation of actions: To display the dataset that was transformed by the 'ProduceModel', it is connected to the 'ShowDF', which is an action component.

UC1-S5 terminating application context: The 'Spark Execute' component takes care of both the creation and the termination of the application context.

What has been evaluated?

Abstraction: The graphical flow-based Spark programming offers a high-level of abstraction and shields the user from the underlying code of Spark, i.e. users do not have to write any Spark code to prototype a Spark application. Additionally, the auto-generation of Spark session initialisation code which is vital for the execution of a Spark program but is unnecessary from a program's problem-solving perspectives thereby allowing the user to concentrate on the concrete problem and achieves a high-level graphical programming paradigm.

Easy parametrisation: Every component used in the flow provides easy customisation via user-supplied parameters from the component's configuration panel, as shown in Fig. 9.

Modularity: Representation of different Spark APIs as modular components help the user to compartmentalise a problem and select specific components to meet those goals since every component caters an independent functionality. The only requirement is that it should be compatible with the predecessor's output.

Flow validation: When the flow is composed by observing the compositional rules discussed in "Components" section, the flow is validated for correctness, i.e. if such a flow would generate a compilable and runnable Spark program and the user is notified of errors if any. Moreover, all components using a particular data abstraction use the same colour code which helps the user in composing a flow. The only visible check from the user side is to ensure the correct positional hierarchy of components in the flow, i.e. the flow begins by an input component, followed by transformation and action/output components.

Use Case 2: Stream processing: applying a machine learning model

For the second use case, i.e. stream processing, using Spark Streaming is not straightforward since the Spark Streaming library is built on the Spark Core and data abstraction provided is *DStreams*. Since Spark core, treats all data as unstructured, working with this involves many steps of data abstraction format conversions.

Approach: Manual programming via Java

The Spark application built using Spark Streaming chiefly consists of the following stages:

UC2-S1 creating application context: The first step with any Spark application is to create an application context. A Spark Streaming context is created, which requires a duration of micro-batch as one of its input.

UC2-S2 reading datasets: After the creating of a streaming application context, data is read from a compatible input source like Kafka. Kafka records published with topic 'trip-data' are read in the form of $< key, value >$ pairs into JavaPairDstream data abstraction.

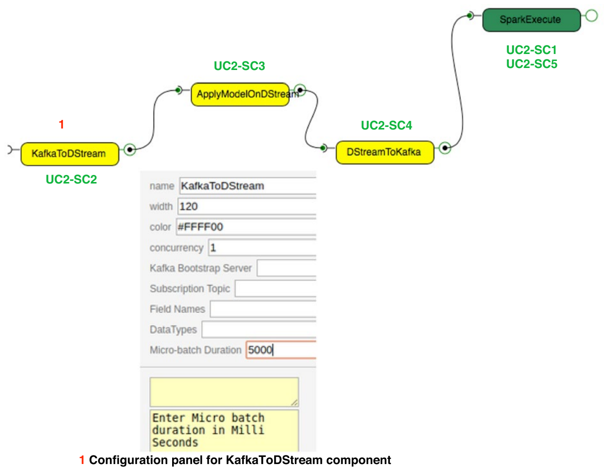
UC2-S3 data transformations: For performing data transformation, i.e. applying a model produced using Pipeline API on DStream data abstraction is not possible without suitable data abstraction interconversion. Hence, we convert DStreams collected over the micro-batch duration into RDD. Each RDD is transformed into DataFrame, and the created machine-learning model from the first use case is applied.

UC2-S4 invocation of actions: Action step involves pushing data out of Spark runtime environment, i.e. push back results to Kafka. Spark Streaming does not support in-built Kafka listener. *'ForEachPartition'* method can be used to push data out of each partition.

UC2-S5 terminating application context: Streaming applications run indefinitely, and the Spark Driver is programmed to keep the context alive for an indefinite period by invoking *'awaitTermination()'* method on the stream context created at the start of the application.

Approach: Graphical programming via aFlux

Figure 10 illustrates an aFlux flow for applying a machine learning model on real-time datasets. The components used are: (i) the 'KafkatoDStream' component, which has only one output port and no input port. This component belongs to *input* category and



name KafkaToDStream
width 120
color #FFFF00
concurrency 1
Kafka Bootstrap Server
Subscription Topic
Field Names
DataTypes
Micro-batch Duration 5000

```
Enter Micro batch
duration in Milli
Seconds
```

Fig. 10 Spark flow in aFlux for applying model to Streaming data, as in [57]

encapsulates input category of Spark APIs as classified in "Classification of Spark APIs" section, (ii) the 'Apply Model on DStream' component which has one input as well as one output port. This belongs to *transformation* category and encapsulates transformation—Type A category of Spark APIs, (iii) the 'DStreamToKafka' component which has only one input port and no output port. This component belongs to *action* category and encapsulates action category of Spark APIs and (iv) the 'Spark Execute' component to mark the end of the flow to begin flow validation and code generation. The wiring of components corresponds with the stages described in manual programming:

UC2-S1 creating application context: The 'Spark Execute' component creates a spark session, and this is abstracted from the user.
UC2-S2 reading datasets: The first component, 'KafkatoDStream' reads data from Kafka and has the necessary conversion code to transform the data into DStream.
UC2-S3 data transformations: The second component, 'Apply Model on DStream', takes the path of a previously created machine learning model and applies it to the input DStream data-set. This component abstracts the process of converting DStream to RDD and then to DataFrame to apply the saved model.
UC2-S4 invocation of actions: The transformed data must be pushed back to Kafka. Producing Kafka records from modified DataFrame is accomplished by the 'DStreamToKafka' component. It does the automatic conversion of data from DStreams to $< key, value >$ format for Kafka.
UC2-S5 terminating application context: The 'Spark Execute' component takes care of both the creation and termination of the application context.

What has been evaluated?

Auto-conversion between different data abstractions: The second component, 'Apply Model on DStream' in the flow does auto-conversion of data abstractions transparently from the user. Spark ML is built on the Spark SQL engine and hence uses DataFrame APIs. The machine learning model created is using DataFrame data abstraction. In stream processing, the data abstraction used is DStreams. Hence, these components transparently read the DStream data from the previous component, convert it into RDD and then invoke the saved ML model on it. After applying the ML model, the resultant-set is again converted back from RDD to DStream and sent as the output to the next component, i.e. 'DStreamToKafka' which takes the DStream input and outputs to Kafka. The auto-conversion process is *not trivial* as the data from DStream has to be converted to RDD after which user-defined functions are necessary to ensure that the dataset has all required columns on which a particular ML model can be applied or to select exact columns to apply the ML model successfully. After application of the ML model, user-defined functions are necessary to convert it back to DStream data abstraction. This process cannot be automated as the fields of a dataset to be selected to apply the ML model have to be checked and selected manually. Furthermore, using RDD necessitates the usage of custom transformation functions. This is done on a case-to-case basis for specific components requiring interoperability with other data abstractions and is left to the developer of the component to provide.

Use Case 3: Performing streaming aggregations

In the third use case, stream aggregations are performed based on event-time and windowed over a given duration. Data is read from Kafka and results are pushed back again to Kafka.

Approach: Manual programming via Java

The Spark application built using Spark Structured Streaming chiefly consists of the following stages:

UC3-S1 creating application context: A Spark session for streaming application is created in the same way as discussed in earlier use cases.

UC3-S2 reading datasets: Data is read from Kafka. Spark Structured Streaming library comes with a built-in Kafka reader which reads Kafka messages in JSON format, maps them to the schema supplied and creates a DataFrame abstraction where entire value part of the record is treated as a string. After this, an unbounded table is created for performing aggregations on the data.

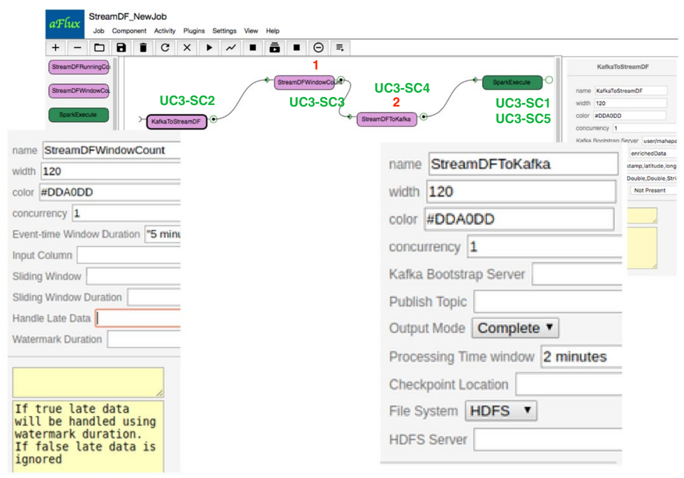
UC3-S3 data transformations: Windowed stream aggregations based on event time is performed on the streaming data with watermark for handling late arrival of data.

UC3-S4 invocation of actions: Spark Structured Streaming associates a streaming query with DataFrames. Only those transformation paths that have an associated streaming query directly or in their path will be included in the execution plan, i.e. data transformations applied onto, and hence results need to be pushed back to Kafka.

UC3-S5 terminating application context: In the last step, the *'awaitTermination()'* method is invoked on the streaming query so that it runs indefinitely.

Approach: Graphical programming via aFlux

Figure 11 illustrates an aFlux flow for performing streaming aggregations on real-time datasets. The components used are: (i) the 'KafkaToStreamDF' component, which has



KafkaToStreamDF
name KafkaToStreamDF
width 120
color #DDA0DD
concurrency 1
Kafka Bootstrap Server user/mahapa
enrichedData
stamp,latitude,long
Double,Double,Stri
Not Present

name StreamDFWindowCount
width 120
color #DDA0DD
concurrency 1
Event-time Window Duration "5 minu
Input Column
Sliding Window
Sliding Window Duration
Handle Late Data
Watermark Duration

```
If true late data
will be handled using
watermark duration.
If false late data is
ignored
```

name StreamDFToKafka
width 120
color #DDA0DD
concurrency 1
Kafka Bootstrap Server
Publish Topic
Output Mode Complete
Processing Time window 2 minutes
Checkpoint Location
File System HDFS
HDFS Server

1 Configuration panel for StreamDFWindowCount component 2 Configuration panel for StreamDFToKafka component

Fig. 11 Spark flow in aFlux for streaming aggregations, as in [57]

only one output port and no input port. This component belongs to *input* category and encapsulates input category of Spark APIs as classified in "Classification of Spark APIs" section, (ii) the 'StreamDFWindowCount' component which has one input as well as one output port. This belongs to *transformation* category and encapsulates transformation—Type A category of Spark APIs, (iii) the 'StreamDFtoKafka' component which has only one input port and no output port. This component belongs to *action* category and encapsulates action category of Spark APIs and (iv) the 'Spark Execute' component to mark the end of the flow to begin flow validation and code generation. The wiring of components corresponds with the stages described in manual programming:

UC3-S1 creating application context: The 'Spark Execute' marks the end of the graphical flow. Its encounter in the flow conveys special meaning to the translator, i.e. to generate the Spark session initialisation, as well as the termination codes necessary for the Spark application.
UC3-S2 reading datasets: The 'KafkaToStreamDF' component is used to read streaming messages from Kafka. It reads $< key, value >$ from Kafka and converts it into streaming DataFrame automatically.
UC3-S3 data transformations: Spark Structured Streaming library offers different aggregations on streaming data. Graphical components to perform window based count and running count operations via the 'StreamDFWindowCount' and 'StreamDFRunningCount' respectively have been implemented. The second component, i.e. the 'StreamDFWindowCount', takes the kind of window-aggregation to be performed as user-input and applies it to the incoming data-set.
UC3-S4 invocation of actions: The result is automatically converted to $< key, value >$ format and pushed to Kafka by the 'StreamDFtoKafka' component.
UC3-S5 terminating application context: The 'Spark Execute' component takes care of both the creation and the termination of the application context.

The action component used in all three use cases of graphical Spark programming has no output port functionality but is connected to the 'Spark Execute' component to mark the end of the flow.

Discussion

Table 4 summarises the comparison of the conceptual approach discussed in this paper with existing solutions offering similar abstraction over Big Data application development. In particular, we compare and contrast the following criteria:

Interaction endpoint For high-level Big Data programming, the tool or approach should offer an endpoint to interact, which can accommodate even less skilled Big Data programmers. A graphical programming interface following the flow-based programming paradigm or the block-based programming paradigm, support for customisation and auto-generation of native Big Data code would be ideal. *The second column in* Table 4 *lists this criterion.* All tools except Apache Beam and Apache Zeppelin provide a graphical interface to design an application. The beam is a high-level unified programming model which has its APIs to write a Big Data application. The application written using Beam's APIs can be executed in a wide range of target frameworks like Spark,

Table 4 Comparison of high-level Spark programming with existing solutions, as in [57]

Tools	Interaction endpoint	Target framework	High-level programming	Code-snippet input not required	Generate Big Data program
Lemonade	Flow-based GUI tool	Spark ML (via Python APIs)	✓	✓	✓
Apache Zeppelin	Interactive shell	Multi-language back-end including Spark and Flink	✗	✗	✗
Apache NiFi	Flow-based GUI tool	Interfaces with Spark and Flink	✓	✗	✗
Apache Beam	Flow-based programming API	Unified Programming model for Big Data systems including Spark and Flink	✗	✗	✗
Microsoft Azure	Flow-based GUI tool	Includes Spark	✓	✗	✗
QryGraph	Flow-based GUI tool	Pig	✓	✓	✓
Nussknacker	Flow-based GUI tool	Flink	✓	✓	✓
QM-IConf	Flow-based GUI tool	Storm	✓	✓	✓
Our approach prototyped in aFlux	Flow-based GUI tool	Spark, Flink [17]. *Extensible*	✓	✓	✓

Flink, Apex, MapReduce, IBM Streams etc. [68]. It is not a graphical tool rather a set of unified APIs which is difficult for less skilled Big Data programmers to use. Similarly, Apache Zeppelin has an interpreter which can take SQL queries or python code snippets and run them against many target environments, including Spark and Flink. In contrast to this, the conceptual approach discussed here, prototyped in aFlux, is a graphical flow-based programming tool which supports customisation of components used in a graphical flow, operates at a high-level over the target Big Data frameworks, abstracts code usage in terms of input during flow design and automates code generation for the user.

Supported target frameworks The second criterion for comparison is the target Big Data framework over which the tool provides a high-level abstraction and if the solution is tied to this particular framework or can be extended. *The third column in Table 4 lists this criterion.* Tools like Lemonade, QryGraph, Nussknacker, QM-IConf are tied to one specific Big data framework and the approach used is not extensible. Zeppelin's interpreter is extensible to frameworks which support SQL based querying or Scala APIs. Apache NiFi is not a Big Data programming tool. It is used to design dataflow pipelines via flow-based programming paradigm. Nevertheless, it has operators/nodes to be used in a flow which can interface it with Big Data applications like Spark and Flink to read data from or send data to. However, to use such interfacing, the developer should write the Spark/Flink application separately and connect it via the data interface operator. Apache Beam's unified API abstracts a large number of Big Data execution engines, i.e. a program developed using Beam's unified API can be executed in several different execution environments with minimal changes [68]. Microsoft Azure is a graphical flow-based platform used heavily for designing Big Data and machine learning applications. The manner in which the graphical flow is translated and run in native Big Data environments is unknown as it is a proprietary tool. However, when a user needs explicitly to use a Big Data target framework, for example, Spark, then the user needs to provide Spark code-snippets, and the platform supports graphical connection to a Spark cluster to send the code snippet for execution and fetch the output. In contrast to this, the conceptual approach developed in this paper, prototyped in aFlux, currently supports Spark. It has been extended to other frameworks like Flink [17].

Level of abstraction A third criterion to compare is to understand the abstraction level a tool offers over a Big Data framework for reducing its complexity, i.e. is the programming done at a high-level over the target APIs. *The fourth column in Table 4 lists this criterion.* All of the existing tools, including the conceptual approach discussed here, provide an abstraction over their supported Big Data frameworks. Apache Beam requires to manually program using its provided APIs and therefore cannot be considered as a high-level programming tool. Nevertheless, it abstracts the complexity of several Big Data frameworks while simultaneously introducing its usage complexity. Similarly, Zeppelin does not provide real high-level programming, but it offers interaction with underlying systems via small code-snippets and not complete programs.

Usage of code-snippets input during application development Another interesting feature to compare is if the tool explicitly requires the user to input code-snippets while developing a Big Data application via graphical flows. The code-snippets are either used for connecting different components used in a flow or for customisation of a component's functionality. This introduces additional complexity and makes it difficult for less skilled Big Data programmers to use the tool. *The fifth column in Table 4 lists this criterion.* Tools like Lemonade, QryGraph, Nussknacker, QM-IConf and the conceptual approach discussed here provide a graphical flow-based application development environment without any code-snippet usage. On the other hand, Zeppelin requires code snippets provided, and they are run in an interactive mode. Apache NiFi does not need code snippets in a flow, but when interfacing with Spark or Flink program is needed, the program must be developed by the user. Apache Beam mandates manual programming using its own set of high-level APIs. Microsoft Azure, in general, does not require code-snippets, but when working with Spark, explicit code-snippets are required as input from the user. *Code Generation* It is also interesting to compare and contrast if the tools generate the complete native Big Data program from the graphical flow created by the user. *The sixth column in Table 4 lists this criterion.* All tools except Apache Zeppelin, Apache NiFi, Apache Beam and Microsoft Azure generate a native Big Data program. Apache Zeppelin makes use of code snippets and is an interactive shell. Apache NiFi runs the flow in its execution environment without generating any final code for the user to inspect. Apache Beam requires the user to program manually using its own set of APIs. Microsoft Azure is a proprietary platform and runs the user flow in its execution environment without any code generation. Tools like QryGraph, Lemonade, QM-IConf, Nussknacker and the conceptual approach discussed in this paper generate target Big Data program from the user flow.

Conclusion

The Big Data ecosystem has a steep learning curve owing to complexity in the programming models of the underlying frameworks, different data abstractions supported in them coupled with the presence of redundancy-abundant APIs. There is no support for end-user programming in the ecosystem, which restricts its widespread application among domain-experts who are not programmers. In this context, the main idea of the paper was to provide domain-experts with flow-based graphical tools for high-level programming of Spark applications. The work involved (i) an analysis of the Spark framework to select the most suitable data abstractions for use in a graphical flow-based programming paradigm. From the available data abstractions of Spark, DStream and DataFrame (including Streaming DataFrame), were selected. The conceptual approach described here supports only the transformations accessible via untyped, i.e. DataFrame APIs and using the aforementioned data abstractions. Any transformation making use of RDD based approach involving user-defined data transformation is not supported. In this context, a classification of APIs present in different Spark libraries was done to create generic invocation statements to invoke the standalone method of the selected APIs, (ii) SparFlo, a library consisting of modular, composable components was created. The components in SparFlo bundle a set of Spark APIs from the selected subset of Spark APIs which are executed in a specific order to perform one data analytic operation. These modular components are composable, following the flow compositional rules and (iii) a generic approach for Spark programming via graphical flows involving validation of flows and auto-generation of Spark programs created using SparFlo components has been proposed, prototyped and evaluated via three use cases. The graphical programming concepts described in this paper have been successfully extended to support other Big Data frameworks like Flink [17]. We are currently working to support graphical flow-based machine learning programming based on frameworks like TensorFlow [69], Torch [70], Caffe [71], Mahout [72] and Singa [73].

Abbreviations
IoT: Internet of Things; MPI: Message Passing Interface; ML: machine learning; JAR: Java ARchive; RDD: resilient distributed dataset; FBP: flow-based programming; UDF: user-defined functions; JVM: Java Virtual Machine; DAG: directed acyclic graph; QM-IConf: The QualiMaster Infrastructure Configuration tool; Lemonade: Live Exploration and Mining Of a Non-trivial Amount of Data from Everywhere.

Acknowledgements
The authors are grateful to Shilpa Ghanashyam Gore who actively contributed to the development of Spark plug-ins for aFlux.

Authors' contributions
Both authors read and approved the final manuscript.

Funding
This work was supported by the German Research Foundation (DFG) and the Technical University of Munich within the Open Access Publishing Funding Programme.

Availability of data and materials
Data sharing is not applicable to this article as no datasets were generated or analysed during the current study.

Ethics approval and consent to participate
This paper uses figures, tables and text from the PhD dissertation titled "High-level Graphical Programming for Big Data Applications" of Tanmaya Mahapatra to be published in Technical University of Munich, 2019. All relevant inclusions from the dissertation have been cited appropriately.

Competing interests
The authors declare that they have no competing interests.

Received: 18 July 2019 Accepted: 18 November 2019
Published online: 08 January 2020

References

1. SciPy.org: NumPy. https://www.numpy.org. Accessed 27 May 2019.
2. SciPy.org: SciPy library. https://www.scipy.org. Accessed 27 May 2019.
3. pandas: Python Data Analysis Library. https://pandas.pydata.org. Accessed 27 May 2019.
4. scikit-learn: Machine learning in Python. https://scikit-learn.org/stable/. Accessed 27 May 2019.
5. Keras: Keras: The Python deep learning library. https://keras.io. Accessed 27 May 2019.
6. Apache: HDFS Architecture Guide. 2018. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed 26 Nov 2018.
7. Apache: The Apache Hive data warehouse software. 2008. https://hive.apache.org. Accessed 27 May 2019.
8. Apache: Pig. 2019. https://pig.apache.org/. Accessed 27 May 2019.
9. Holwerda R, Hermans F. A usability analysis of blocks-based programming editors using cognitive dimensions. In: 2018 IEEE symposium on visual languages and human-centric computing (VL/HCC); 2018, pp. 217–25. https://doi.org/10.1109/VLHCC.2018.8506483.
10. Morrison JP. Flow-based programming, 2nd Edition: a new approach to application development. Paramount: CreateSpace; 2010.
11. Bau D, Gray J, Kelleher C, Sheldon J, Turbak F. Learnable programming: blocks and beyond. Commun ACM. 2017;60(6):72–80. https://doi.org/10.1145/3015455.
12. Mason D, Dave K. Block-based versus flow-based programming for naive programmers. In: 2017 IEEE blocks and beyond workshop (BB). 2017, pp 25–8. https://doi.org/10.1109/BLOCKS.2017.8120405.
13. DeRemer F, Kron HH. Programming-in-the-large versus programming-in-the-small. IEEE Trans Softw Eng. 1976;SE–2(2):80–6.
14. Cunniff N, Taylor RP. Empirical studies of programmers: second workshop. chap. Graphical vs. textual representation: an empirical study of novices' program comprehension. Norwood: Ablex Publishing Corp.; 1987, pp. 114–31. http://dl.acm.org/citation.cfm?id=54968.54976.
15. Gorlick M, Quilici A. Visual programming-in-the-large versus visual programming-in-the-small. In: Proceedings of 1994 IEEE symposium on visual languages; 1994, pp. 137–44. https://doi.org/10.1109/VL.1994.363631.
16. Whitley KN, Novick LR, Fisher D. Evidence in favor of visual representation for the dataflow paradigm: an experiment testing labview's comprehensibility. Int J Hum Comput Stud. 2006;64(4):281–303. https://doi.org/10.1016/j.ijhcs.2005.06.005.
17. Mahapatra T, Gerostathopoulos I, Fernández FA. Prehofer, C.: Designing Flink Pipelines in IoT Mashup Tools 2018;2316(03), 41–53. http://ceur-ws.org/Vol-2316/paper3.pdf.
18. Mahapatra T, Gerostathopoulos I, Prehofer C, Gore SG. Graphical Spark Programming in IoT Mashup Tools. In: 2018 fifth international conference on Internet of Things: systems, management and security; 2018, pp. 163–70. https://doi.org/10.1109/IoTSMS.2018.8554665.
19. Mahapatra T, Prehofer C. aFlux: Flow-based programming for Big Data. 2019. https://aflux.org. Accessed 20 June 2019.
20. Mahapatra T, Prehofer C, Gerostathopoulos I, Varsamidakis I. Stream analytics in IoT mashup tools. In: 2018 IEEE symposium on visual languages and human-centric computing (VL/HCC). 2018, pp 227–31. https://doi.org/10.1109/VLHCC.2018.8506548.
21. Mahapatra T, Prehofer C. aFlux: Graphical flow-based data analytics. Softw Impacts. 2019;2:100007. https://doi.org/10.1016/j.simpa.2019.100007
22. Friedman E, Tzoumas K. Introduction to Apache Flink. Newton: O'Reilly; 2016.
23. Zecevic P, Bonaci M. Spark in action. 1st ed. Greenwich: Manning Publications Co.; 2016.
24. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, Volume 6, Berkeley: OSDI'04. USENIX Association; 2004, pp. 10. http://dl.acm.org/citation.cfm?id=1251254.1251264.
25. Forum MP. Mpi: A message-passing interface standard. Knoxville: Tech. rep.; 1994. http://dl.acm.org/citation.cfm?id=1863103.1863113
26. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. In: E.M. Nahum, D. Xu (eds.) HotCloud. USENIX Association. 2010. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.
27. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M. Spark sql: relational data processing in spark. In: Sellis TK, Davidson SB, Ives ZG, eds. SIGMOD conference. ACM; 2015, pp. 1383–94. https://doi.org/10.1145/2723372.2742797.
28. Apache: GraphX: GraphX is Apache Spark's API for graphs and graph-parallel computation. https://spark.apache.org/graphx/. Accessed 27 May 2019.
29. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A. Mllib: Machine learning in apache spark. J Mach Learn Res. 2016;17(34):1–7. http://jmlr.org/papers/v17/15-237.html.
30. Scala: The scala programming language. https://www.scala-lang.org. Accessed 05 June 2019.
31. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I. Apache Spark: a Unified engine for Big Data processing. Commun ACM. 2016;59(11):56–65. https://doi.org/10.1145/2934664.
32. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. 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, NSDI'12. USENIX Association, Berkeley; 2012, pp. 2. http://dl.acm.org/citation.cfm?id=2228298.2228301.
33. Morrison JP. Flow-based programming. In: Proceedings 1st international workshop on software engineering for parallel and distributed systems. 1994, pp 25–9.
34. Carkci M. Dataflow and reactive programming systems: a practical guide. 1st ed. Scotts Valley: CreateSpace Independent Publishing Platform; 2014.
35. Agha G. Actors: a model of concurrent computation in distributed systems. Cambridge: MIT Press; 1986.
36. Mahapatra T, Gerostathopoulos I, Prehofer C. Towards integration of Big Data analytics in Internet of Things Mashup tools. In: Proceedings of the seventh international workshop on the Web of Things, WoT '16. New York: ACM; 2016, pp. 11–6. https://doi.org/10.1145/3017995.3017998.
37. Mahapatra T, Prehofer C. Service mashups and developer support. Dig Mobil Platf Ecosyst. 2016;. https://doi.org/10.14459/2016md1324021.
38. Eichelberger H, Qin C, Schmid K. Experiences with the model-based generation of big data pipelines. In: Mitschang B, Nicklas D, Leymann F, Schöning H, Herschel M, Teubner J, Härder T, Kopp O, Wieland M, editors. Datenbanksysteme für business, technologie und web (BTW 2017)—workshopband. Bonn: Gesellschaft für Informatik e.V; 2017. p. 49–56.
39. Apache: Apache Storm. https://storm.apache.org. Accessed 27 May 2019.
40. Santos Wd, Avelar GP, Ribeiro MH, Guedes D, Meira W Jr. Scalable and efficient data analytics and mining with lemonade. Proc VLDB Endow. 2018;11(12):2070–3. https://doi.org/10.14778/3229863.3236262.
41. Schmid S, Gerostathopoulos I, Prehofer C. Qrygraph: a graphical tool for big data analytics. In: SMC'16. 2016.
42. Touk: Nussknacker. streaming processes diagrams. https://touk.github.io/nussknacker/. Accessed 27 May 2019.
43. Apache: Apache Beam: An advanced unified programming model. https://beam.apache.org. Accessed 27 May 2019.
44. Apache: Apache Apex: Enterprise-grade unified stream and batch processing engine. https://apex.apache.org. Accessed 27 May 2019.
45. Zeppelin Apache. https://zeppelin.apache.org/docs/0.7.0/. Accessed 22 June 2018.
46. Microsoft: Microsoft Azure. https://docs.microsoft.com/en-us/azure/hdinsight/. Accessed 22 June 2018.
47. Apache: NiFi. https://nifi.apache.org/docs.html. Accessed 05 Sept 2018.
48. Baldwin CY, Clark KB. Modularity in the design of complex engineering systems. Berlin: Springer; 2006. p. 175–205. https://doi.org/10.1007/3-540-32834-3_9.
49. Johansson P, Holmberg H. On the modularity of a system. 2010.
50. Apache Spark: RDD Programming Guide. 2019. https://spark.apache.org/docs/latest/rdd-programming-guide.html. Accessed 16 May 2019.
51. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: Fault-tolerant streaming computation at scale. In: M. Kaminsky, M. Dahlin, eds. SOSP. ACM; 2013, pp. 423–38. http://dblp.uni-trier.de/db/conf/sosp/sosp2013.html#ZahariaDLHSS13.
52. Apache Spark: Spark SQL, DataFrames and datasets guide. https://spark.apache.org/docs/latest/sql-programming-guide.html. Accessed 24 Apr 2018.
53. Apache: Spark Structured Streaming Programming Guide. 2018. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. Accessed 12 Dec 2018.
54. Apache Kafka: A distributed streaming platform. 2018. https://kafka.apache.org. Accessed 24 Apr 2019.
55. Zaharia M, Das T, Li H, Shenker S, Stoica I. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing, HotCloud'12. USENIX Association, Berkeley; 2012, pp. 10. http://dl.acm.org/citation.cfm?id=2342763.2342773.
56. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M. Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data, SIGMOD '15. New York: ACM; 2015, pp. 1383–94. https://doi.org/10.1145/2723372.2742797.
57. Mahapatra T. High-level graphical programming for Big Data applications. Dissertation, Technische Universität München, München; 2019. http://mediatum.ub.tum.de/?id=1524977
58. Damji J. A Tale of three Spark APIs. https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html. Accessed 24 Dec 2018.
59. Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. Addison-Wesley series in computer science / World student series edition. Addison-Wesley; 1986. http://www.worldcat.org/oclc/12285707.
60. Wikipedia: Composability. https://en.wikipedia.org/wiki/Composability. Accessed 27 May 2019.
61. Attiogbé C, André P, Ardourel G. Checking component composability. In: Löwe W, Südholt M, editors. Software composition. Berlin: Springer; 2006. p. 18–33.
62. Neumann P. Principled assuredly trustworthy compusable architectures. DARPA final report, SRI Project P11459, December 2004.
63. Li Y, Tan T, Xue J. Understanding and analyzing java reflection. ACM Trans Softw Eng Methodol. 2019;28(2):7:1–50. https://doi.org/10.1145/3295739.
64. Livshits B, Whaley J, Lam MS. Reflection analysis for java. In: Proceedings of the third Asian conference on programming languages and systems, APLAS'05. Berlin: Springer; 2005, pp. 139–60. https://doi.org/10.1007/11575467_11.
65. Wikipedia: .properties article. https://en.wikipedia.org/wiki/.properties. Accessed 27 May 2019.
66. Stahl T, Voelter M, Czarnecki K. Model-driven software development: technology, engineering, management. New York: Wiley; 2006.
67. JavaPoet. https://github.com/square/javapoet/blob/master/README.md. Accessed 24 Apr 2018.
68. Apache: Beam capability matrix. https://beam.apache.org/documentation/runners/capability-matrix/. Accessed 05 June 2019.
69. TensorFlow: An open source machine learning library for research and production. 2015. https://www.tensorflow.org. Accessed 20 June 2019.
70. Torch: A scientific computing framework for LUAJIT. 2002. http://torch.ch. Accessed 20 June 2019.

71. Caffe: Deep learning framework. 2017. https://caffe.berkeleyvision.org. Accessed 20 June 2019.
72. Apache: Mahout: For Creating Scalable performant machine learning applications. 2009. https://mahout.apache.org. Accessed 20 June 2019.
73. Apache: Singa. 2015. https://singa.incubator.apache.org. Accessed 20 June 2019.

Publisher's Note