High-level Graphical Programming for Big Data Applications

Tanmaya Mahapatra

Fakultät für Informatik

Informatik 4 - Software und Systems Engineering

der Technische Universität München

High-level Graphical Programming for Big Data Applications

Tanmaya Mahapatra, M.Sc.

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:	1. Priv.-Doz. Dr. Christian Prehofer
	2. Prof. Dr. Florian Matthes

Die Dissertation wurde am 21.06.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.10.2019 angenommen.

Acknowledgement

First and foremost, I would like to offer my sincere prayers at the feet of my mother *Bagalamukhi*, one of the extraordinary manifestations of *Divine Mother* without whose grace and blessings nothing is permissible in this Universe. She has been my beacon of hope, the scintillating sun to ponder over as well as a shoulder to cry on while sailing across the turbulent waters of life.

गेहं नाकति, गर्वितः प्रणमति, स्त्री-संगमो मोक्षति,
द्वेषी मित्रति, पातकं सु-कृतति, क्ष्मा-वल्लभो दासति ।
मृत्युर्वैद्यति, दूषणम् गुणति, त्वत्-पाद-संसेवनात्,
वन्दे त्वां भव-भीति-भञ्जन-करीं गौरीं गिरीश-प्रियाम् ।।

"O Mother! By adoring your lotus feet, the adorer's house turns into heaven, pride turns into humility, a desire-ridden person attains a pure mind free from desires of all sorts & hence fit for self-realisation, an enemy turns into a great friend, an emperor turns into a commoner, a sinner turns into a saint, death itself serves the adorer as a great friend-cum-healer, and all bad habits go away giving rise to an impeccable & exemplary character. By a mere glance, you burn away all worldly relations of the worshipper which are the products of delusion & ignorance of the human mind, purifying him slowly to make him realise his own deathless, pristine nature which is non-different from you. Hence, throwing away all my worldly worries & concerns O Mother! I incessantly meditate on you"

I want to express my sincere gratitude and reverence from the very inner core of my heart to Dr. Christian Prehofer for providing me with a unique and memorable opportunity to complete my dissertation under his supervision. I am appreciative of all his constructive criticisms, feedbacks, discussions, patience in understanding my mental attitude and moral support throughout the work, which played a pivotal role in the successful completion of this dissertation.

I am also thankful to Prof. Dr. Manfred Broy for providing all the facilities needed to carry out my research and a productive environment to work. My special thanks to Prof. Dr. Florian Matthes for providing me with an opportunity to work in a very conducive research project which laid the foundations of this work. I am also grateful to Prof. Dr. Klaus A. Kuhn for supporting me during the last phase of the dissertation.

I would like to express my unstinting thanks to Dr. Ilias Gerostathopoulos for providing me with moral support, guidance and helping me to steer through the tumultuous phases of a PhD. I am incredibly indebted to him for instilling in me the level of confidence and showing everything is achievable with perseverance. His constant boosting and words of encouragement became one of my sources of inspiration to complete the work in time.

My thanks to Georgios Pipelidis for his constant encouragement and words filled with optimism. That indeed helped me a lot to progress smoothly.

My special thanks to my friends and office colleagues, especially Margret Heinze, Ingrid Gatz, Hannelore Plagge, Dr. Diego Marmsoler, Dr. Daniel Méndez, Nikolaos Tsiamitros, Dominik Ascher, Habtom Gidey and Rishabh Kumar. They made me feel that I was not alone at any stage of my work.

My tons of love and regards to Dr. Biswa Mohan Mahapatra, my father and Dr. Puspanjali Rath, my mother, for their continuous support throughout the journey. Her incessant advise against dropping-out has finally fructified! Thank you for being a pillar of support for all these years.

Lastly, I would like to conclude while thanking all remaining members of my research group and all those friends whose name I might have missed for their wholehearted support, guidance and suggestions which ultimately culminated in the form of this dissertation.

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 and highly parallel software frameworks as well as systems. The inherent complexity in programming Big Data applications is also due to the presence of a wide range of target frameworks, each with their varied data abstractions and APIs. The thesis 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 thesis contributes to these aspects by (i) providing a thorough analysis and classification of the widely used Spark and Flink frameworks 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 Big Data systems from graphical flows that comprises early-stage validation and code generation of Big Data applications.

The thesis also demonstrates that a flow-based programming model with concurrent execution semantics is suitable for modelling a wide range of Big Data applications. The graphical programming approach developed in this thesis is the first approach to support high-level Big Data application development by making it independent of the target Big Data frameworks. Use cases for Spark and Flink have been prototyped and evaluated to demonstrate code-abstraction, automatic data abstraction conversion and automatic generation of target Big Data programs, which are the keys to lower the complexity and its ensued learning curve involved in the development of Big Data applications.

Zusammenfassung

Die enorme Menge an Sensordaten im Rahmen des Internet der Dinge (Englisch: Internet of Things oder IoT) erfordert komplexe Datenanalysen. Anwendungen für solche Big Data Systeme zu erstellen ist aufwendig, da sehr komplexe, hochparallele Softwaresysteme und Werkzeuge verwendet werden. Die inhärente Komplexität bei der Programmierung von Big Data Anwendungen aufgrund der Vielzahl von unterschiedlichen Datenabstraktionen und APIs welche dabei zum Einsatz kommen, erschwert die Entwicklung solcher Anwendungen zusätzlich. Diese Arbeit zielt darauf ab, diese Komplexität und die daraus resultierende Lernkurve zu reduzieren, indem sie es Domänenexperten, die nicht unbedingt erfahrene Big Data Programmierer sind, ermöglicht, Datenanalyse-Anwendungen mit Hilfe von domänenspezifischen, grafischen Werkzeugen zu entwickeln. Hierbei wird das Konzept der datenflussorientierten Programmierung verwendet, das in sogenannten IoT-Mashup-Tools üblich ist.

Die Arbeit adressiert diese Probleme durch (i) Analyse und Klassifikation der weitverbreiteten Spark und Flink Frameworks sowie Auswahl geeigneter Datenabstraktionen und APIs für den Einsatz in einem grafischen, datenflussorientierten Programmierparadigma und (ii) Entwicklung eines neuartigen, generischen Ansatzes zur Programmierung von Big-Data-Systemen aus grafischen Abläufen, welcher die frühzeitige Validierung und Codegenerierung von Big Data Anwendungen umfasst.

Die Arbeit zeigt auch, dass ein datenflussorientiertes Programmiermodell mit paralleler Ausführungssemantik für die Modellierung einer breiten Palette von Big Data Anwendungen geeignet ist. Der in dieser Arbeit entwickelte grafische Programmieransatz ist ein erster Ansatz zur Unterstützung der Entwicklung von Big Data Anwendungen auf hohem Abstraktionsniveau, indem er sie unabhängig von den angestrebten Big Data-Frameworks macht. Anwendungsfälle für Spark und Flink werden prototypisch erstellt und ausgewertet, um Code-Abstraktion, automatische Datenabstraktionskonvertierung und automatische Generierung von Ziel-Big-Data-Programmen zu demonstrieren. Diese sind der Schlüssel zur Senkung der Komplexität und der daraus resultierenden Lernkurve.

Contents

List of Figures

List of Tables

1 Introduction

"The unexamined life is not worth living."

— Socrates

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.

Nevertheless, *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 [141], SciPy [142], Pandas [125], SciKit-Learn [140], Keras [97] and others. The data gathered 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. 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 [24], Hive [20], Pig [27] 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 tool-chains 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.

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* [90] and *flow-based programming* [121] with both the approaches having their pros and cons, respectively [114, 40]. Nevertheless, as far as programming-in-the-large [65] is concerned, such as complex heterogeneous systems which can also accommodate the example of Big Data systems, flow-based programming approaches are well suited [81, 59, 162] compared to their block-based counterparts [90].

Block-based Programming In block-based programming, the programming constructs are represented via graphical blocks. Popular examples include Google Blockly [79] and MIT Scratch [118]. Example of how such a program looks like is shown in Figure 1.1. The block-based programming approach suffers mainly in the dimensions of:

Viscosity The viscosity of a notational system describes its resistance to change [90]. A system is more viscous when a single change (in the mind of the user) "requires an undue number of individual actions" [82].

Role-Expressiveness It is defined as *"intended to describe how easy it is to answer the question: 'What is this bit for'?"* [82]. The 'bit' can be an individual entity in the notation or a substructure in the program, consisting of multiple entities [90].

Block-based programming approach uses visual notation for programming constructs like for loop, while loop, methods containing logic etc. This affects *understandability* when the logic of the program grows, affecting the dimensions of *'role-expressiveness'* and *'viscosity'* severely. Additionally, using graphical constructs for underlying programming constructs defeats the purpose of high-level programming where the critical requirement is a *higher-level of abstraction* and not mere substitution of programming constructs with visual notations. Nevertheless, attempts have been made to support high-level programming for data analytics via the block-based programming approach as in the tool named 'milo' [132].

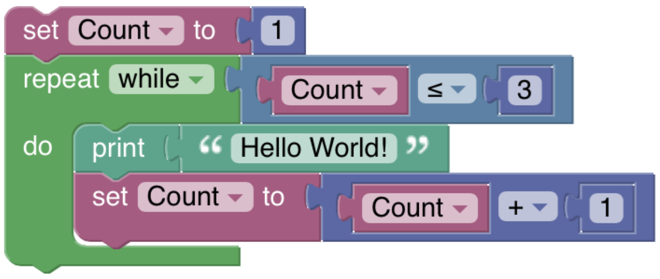


Figure 1.1: Block-based programming: print inside a while loop [1]

Flow-based Programming In flow-based programming, the program is constructed by specifying the data-flow between various components in the form of a graph where the vertices represent data sources, data transformers and data sinks while the edges represent data-flow pathways. Example of how such a program looks like is shown in Figure 1.2. Such graphical approach restricts expressiveness in specific scenarios because some information that can be easily expressed via code is challenging to represent in visual notation and must be implemented internally [143, 89].

Despite the enormous potential of combining IoT sensing and actuating with data analytics, developing applications that control the operation of IoT sensors and actuators is challenging. Developers have to write complex codes to access the datasets from the sensors of different devices. Also, they need to perform data mediation before using the data for real insights. To deal with some of these challenges, dedicated IoT development

[1] source: `https://developers.google.com/blockly/`

tools called IoT mashup tools can be used. Such tools expedite the process of creating and deploying simple IoT applications that consume data generated from IoT sensors, publish data to external services or other devices. They typically offer graphical interfaces for specifying the data-flow between sensors, actuators, and services which lowers the barrier of creating IoT applications for end-users. Dataflow based programming is a case of flow-based programming paradigm [48]. Hence, the resulting application follows the flow-based programming model, where outputs of a node in the flow become inputs of the next node. With decreased learning curve, end-users, i.e. users with no prior experience and training can quickly prototype an IoT application. An exemplary tool is Node-RED [92], a prominent visual programming environment developed by IBM for visual programming of IoT applications. Currently, *the mashup tools* including Node-RED *do not support the specification and execution of flows that include Big Data analytics computations*, i.e. they do not support in-flow data analytics with Big Data technologies.

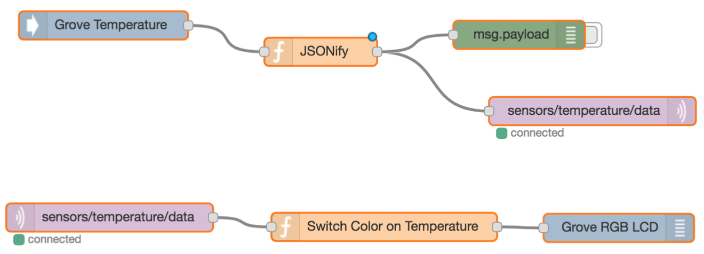


Figure 1.2: Flow-based programming in Node-RED [92]: print inside a while loop [2]

A flow-based programming approach has been used in the context of IoT application development via graphical mashup tools. The graphical flow-based programming paradigm of the mashup tools provides an optimal construct for supporting high-level programming for Big Data applications which would qualify as programming-in-the-large. Big Data analytics via flow-based graphical tools would abstract away all the technical complexities involved in setting up and writing Big Data applications from the end-user and also permit to leverage the opportunities offered by the increasing number and sensing capabilities of connected devices for developing sophisticated applications that employ data analytics as part of their business logic to make informed decisions based on sensed data. This would lead to the usage of Big Data analytics in the context of IoT.

1.1 Need to go Beyond

Until recently, a lot of research has been done on how to collect and store such data in Big Data infrastructure. Analytics is performed on these datasets separately to gain insights [51]. However, as far as the application development scenario is concerned using graphical flow-based programming tools, no significant amount of research has been done on how to make use of Big Data analytics during application development. Traditionally, the worlds of IoT and Big Data have stood apart from each other. IoT is used for

[2]source: `https://software.intel.com/en-us/node/721391/`

collecting data into storage and writing business logic while Big Data for analysis. However, in many scenarios, it is important to interlink both the worlds in an integrated way. There are certain scenarios in which business logic of flow-based applications may need input from Big Data jobs, i.e. applications may require to trigger data analysis. Similarly, after the execution of Big Data jobs there may arise need to perform some additional task, i.e. may need to trigger a business logic. An example scenario has been illustrated where the integration of Big Data analytics and business logic in application development is really useful to generate value for end-users.

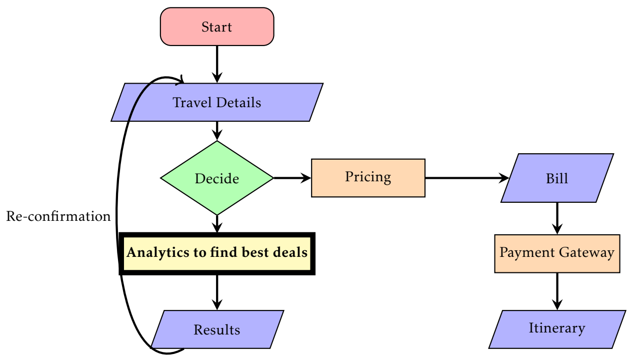


Figure 1.3: Mashup involving Big Data analytics for travel route optimisation.

Public transportation is becoming increasingly tough in most modern cities of the world today. It is desirous to know real-time traffic situations for smooth transit within different areas of a city. Therefore the idea of connected mobility is highly sought for. Connected mobility, an application of IoT, takes into account all available transit options, payment services along with real-time traffic information and map services to facilitate optimal route planning for hassle free transportation. The traffic conditions, payment services, parking spot availability, public transit options with their rates and historical data are offered as REST services in the context of connected mobility. Applications can be created via flow-based programming tools by third party application developers by consuming the offered REST services appropriately which assist the user to travel from one point to another with in the city limits. The application suggests the user to use a combination of transit options and handles the entire trip cost in an integrated manner since different services may have different providers. It also guides the user during his travel with a map. The flow of such an application where real-time analytics is unavoidable is depicted in Figure 1.3. The application takes user input and during first iteration it performs analytics to get real-time traffic information and then appropriately suggest optimal routes with a combination of transportation options which can be followed for those paths. This cycle is iterated till the user is satisfied with the results after which the flow in the application moves on to calculate the trip cost, display it to the user, handle the payment through a payment gateway and present the final itinerary to the user.

1.2 Problem Statement

Although high-level graphical programming of Big Data systems via flow-based programming tools would enable end-users, i.e. users with little to no prior programming expertise, to develop sophisticated, high-impact applications, it is far from straightforward. Such an approach should go beyond from merely making use of graphical flows that only act as data providers for Big Data clusters. Developers should also be able to specify Big Data analytics jobs and consume their results within a single application model. This can effectively enhance the development of applications that continuously harness the value out of sensed data in their operation. The main rationale is to develop a uniform application model and associated model of computation, that will facilitate specification of big data analytics as a part of the regular business logic of an application.

High-level graphical programming of Big Data applications via flow-based programming tools entails the following major challenges:

C1 Inherent complexities in programming of Big Data applications. The Big Data ecosystem is too large in the number of frameworks, solutions and platforms available to work with. Each framework is complex to use. The available solutions can be categorised into distributions, execution engines/frameworks, analytics platforms etc. The focus here is on the execution engines/frameworks which started with the MapReduce [23] programming model on top of Hadoop Distributed File Systems (HDFS) [24]. Hadoop MapReduce is an open-source framework to write applications to process data stored on HDFS in batch mode. But the paradigm shifted rapidly towards Apache Spark [168] which is way faster when compared to MapReduce as it does most of its operations in memory, thereby reducing the number of disk access operations. It also offers high-level operators on top of custom data abstractions like the Resilient Distributed Dataset (RDD) [31] whereas every functionality needed has to be manually coded while working with MapReduce. Additionally, Spark offers functionality to process streaming datasets. Similarly, Apache Flink [150] with its own data abstractions and APIs is also gaining popularity. Its a strong competitor of Spark. The important point to highlight here is that the most popular Big Data Frameworks, i.e. Apache Spark and Apache Flink which are used to write Big Data applications typically offer the following differences which gives rise to a learning curve:

1. Both of the frameworks offer different programming models and data abstractions to represent and process datasets, like RDDs, DataFrame, DStreams in case of Spark while DataSet and DataStream in case of Flink.

2. They offer different APIs on top of data abstractions to work with datasets.

3. Different libraries are present in the framework for machine learning, graph processing, stream processing etc. Additionally, often the APIs with their functionalities overlap.

It is necessary to distil the programming concepts, data abstractions and APIs from the different libraries of both the frameworks which are well-suited to be represented and used from a flow-based programming paradigm. In addition to this, the inherent data processing architecture of both the systems are different as Flink is heavily based on Kappa architecture [71] while Spark tries to be Lambda [113] compliant as it grew out from the ecosystem of Hadoop.

C2 Integrating with Big Data systems. IoT applications heavily rely on REST architectural style for communication and integration between components because it is simple and the communication is uniform. Uniform communication is especially important in IoT due the seer presence of a large number of heterogeneous devices. This is an opportunity for Big Data analytics tools: if they also offered their APIs in REST, they could be invoked as regular services within IoT applications. Certain Big Data tools already offer RESTful APIs. For instance, in Spark one can invoke Spark jobs, monitor and control them via REST calls. Unfortunately, most other Big Data analytics tools (including Hive and Pig) lack REST interfaces. Additionally, the problem takes a second dimension, i.e. Big Data tools like Spark and Flink make use of different programming APIs, evaluation models and data representation formats to create runnable programs within those environments and its not possible to achieve this via simple RESTful approach. Generic translation of flows into native Big Data programs coupled with RESTful operations to load, monitor and return results from Big Data systems to flow-based programming tool environment is needed to create applications which perform data analytics and also consume their results in a single application model.

C3 Usage of flow-based programming for Big Data analytics. It is important to state that the flow-based programming tools used for IoT application development generally involved collection and processing of datasets. They have also been used to feed data into Big Data systems for processing but were not designed to support flow-based Big Data programming. Therefore, an improvement over the current architectural style of flow-based programming tools is necessary to support high-level programming of Big Data applications, particularly in the following dimensions:

1. The single-threaded model and blocking semantics used in flow-based programming tools prevent specification of Big Data application via flows as Big Data programs are typically repetitive and time-consuming. Without the support for multi-threaded and concurrent execution semantics, the applications developed would cause bottleneck for the components connected after time-consuming data analytics components in a flow. Additionally, with a single thread of execution it would be difficult to scale the application to meet increasing data processing loads.

2. Flow-based programming tools have so far focused on enabling end-users, even non-programmers, create and deploy relatively simple applications. As a result, a number of common features for development environments and platforms, such as built-in security mechanisms and code generation capabilities, are not included in current popular flow-based programming tools. To allow the integration with Big Data analytics tools, flow-based programming tools have to provide these missing features, along with features included in data scientists tool-kits (e.g. pre-fetching of example data from a dataset, graphical inspection of datasets, etc.). In the end, the challenge is to shift the focus from end-users to developers and data scientists.

C4 Visual programming not language independent. Flow-based programming tools provide a graphical language for modelling the data-flow between the various components of an application. This notation has its limitations when modelling complex behaviours that involve loops or generic operation [129, 131]. At the same time, Big Data analytics tools have their own (non-graphical) languages, such as Pig Latin, SQL/SparkSQL [34] and different APIs for writing driver programs to be executed in Spark and Flink run-time environments. There are currently only a few attempts to specify Big Data analytics jobs graphically (e.g. the QryGraph tool for specifying Pig Latin queries [138]). A seamless development experience integrating flow-based applications and Big Data analytics would ideally provide a single visual notation for specifying both the application logic and the data analysis logic in graphical editor (e.g. allowing for graphically specifying the contents of 'Analytics to find deals' box in Figure 1.3). As this would allow developers to use a single consolidated tool chain, it would greatly simplify the application development.

1.3 Research Goals

Responding to the challenges presented in Section 1.2, this thesis focuses primarily to support high-level graphical programming for Big Data applications via flow-based programming paradigm. The thesis thus targets the following research goals (Figure 1.4 illustrates the overall ideas of the research goals):

G1 The first goal is to *analyse Big Data frameworks*. Big Data systems typically come with a wide range of frameworks like Spark, Flink, Pig etc. which have very diverse programming style. For instance, Pig programs follow a scripting style while Hive supports SQL like queries to query datasets. Spark and Flink involve complex data abstraction formats, different APIs and have different libraries offering different functionalities. All these are used in conjunction by a developer manually to create a driver program which then runs in the run-time environment to do the actual data-analytics. In order to support high-level programming of such systems, it is of paramount importance to:

1. *Analyse the target Big Data frameworks, extract data abstractions and APIs* which are compatible with the flow-based programming paradigm. Two Big Data frameworks namely Spark and Flink have been chosen for the thesis work.

2. Representation of the selected APIs operating on the compatible data abstractions as modular components (for definition of modularity in the context of this work, please refer to Section 5.2).

G2 The second goal is to *support graphical programming of existing Big Data systems*, i.e. represent the modular components in a graphical tool based on the flow-based programming paradigm and enable high-level programming by:

1. Developing a *generic approach to parse graphical flows making use of such modular components to generate native Big Data programs* and with support for early-stage validation so that a flow always yields a compilable and runnable Big Data program.

2. Additionally, *design new improved concepts for flow-based programming tools* to support designing applications with *concurrent execution semantics*, thereby overcoming the prevalent architectural limitations in the state-of-the-art IoT mashup tools. A flow-based programming model with concurrent execution semantics is suitable for modelling a wide range of Big Data applications currently used in Data Science. Without the aforementioned semantics, designing a flow involving Big Data analytics would lead to components waiting to execute for a long time as Big Data jobs usually take long to finish their execution. This would lead to inefficient design of applications.

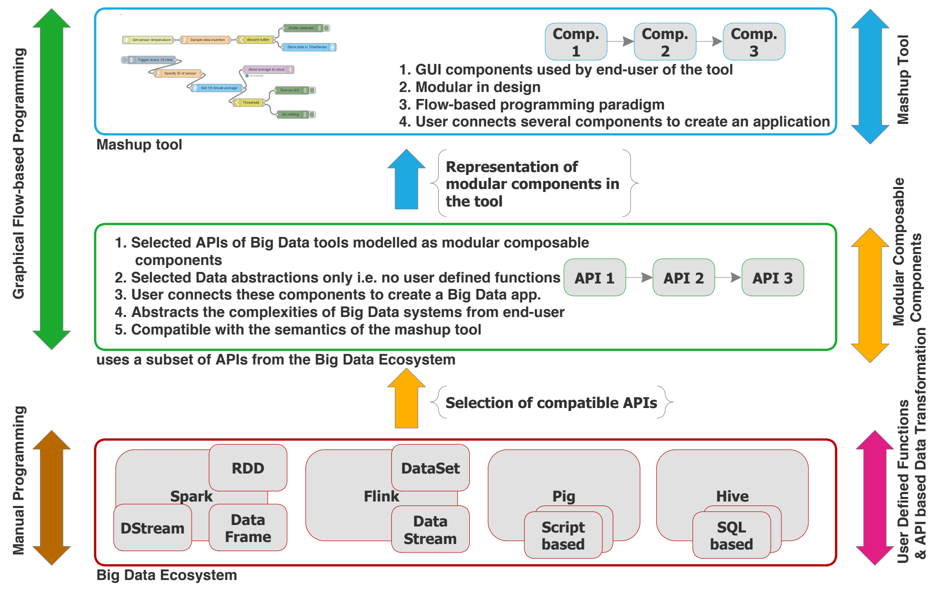

Graphical Flow-based Programming
Comp. 1
Comp. 2
Comp. 3
1. GUI components used by end-user of the tool
2. Modular in design
3. Flow-based programming paradigm
4. User connects several components to create an application
Mashup tool
Mashup Tool
Representation of modular components in the tool
1. Selected APIs of Big Data tools modelled as modular composable components
2. Selected Data abstractions only i.e. no user defined functions
3. User connects these components to create a Big Data app.
4. Abstracts the complexities of Big Data systems from end-user
5. Compatible with the semantics of the mashup tool
API 1
API 2
API 3
uses a subset of APIs from the Big Data Ecosystem
Modular Composable Components
Selection of compatible APIs
Manual Programming
Spark
RDD
DStream
Data Frame
Flink
DataSet
Data Stream
Pig
Script based
Hive
SQL based
Big Data Ecosystem
User Defined Functions & API based Data Transformation

Figure 1.4: Modular subset of Big Data systems compatible with flow-based programming paradigm (tool-agnostic)

The two goals cut across all the identified challenges, while focusing explicitly on challenges **C1-C3**.

1.4 Contributions & Publications

The main contributions presented in this thesis comes from a number of publications stemming from the research work and collaboration done within the TUM-Living Lab Connected Mobility project, in which the author participated as an early stage researcher. The following peer-reviewed papers and technical reports form the core contribution presented in this thesis.

Publications

1. Mahapatra, T., Gerostathopoulos, I. and 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, ser. WoT'16. New York, NY, USA: ACM, 2016, pp. 11-16. `http://doi.acm.org/10.1145/3017995.3017998`

2. Mahapatra, T., Prehofer, C., Gerostathopoulos, I. and Varsamidakis, I.: Stream Analytics in IoT Mashup Tools. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 227-231 (Oct 2018). `https://doi.org/10.1109/VLHCC.2018.8506548`

3. Mahapatra, T., Gerostathopoulos, I., Prehofer, C. and Gore, S. G. , "Graphical Spark Programming in IoT Mashup Tools," 2018 Fifth International Conference on Internet of Things: Systems, Management and Security, Valencia, 2018, pp. 163-170. `https://doi.org/10.1109/IoTSMS.2018.8554665`

4. Mahapatra, T., Gerostathopoulos, I., Fernández, F. and Prehofer, C., "Designing Flink Pipelines in a IoT Mashup Tools," Proceedings of the 4th Norwegian Big Data Symposium (NOBIDS 2018), vol. 2316, pp. 41-53, Trondheim, Norway, November 14, 2018. `http://ceur-ws.org/Vol-2316/paper3.pdf`

5. Mahapatra, T. and Prehofer, C., "Service Mashups and Developer Support," Project Consortium TUM Living Lab Connected Mobility, Digital Mobility Platforms and Ecosystems, Software Engineering for Business Information Systems (sebis) TUM, July 2016. `https://mediatum.ub.tum.de/node?id=1324021`

An overview of the contributions is presented in Chapter 3.

1.5 Structure

The thesis is structured in the following way. Chapter 2 presents the state-of-the art in Big Data analytics and IoT application development using mashup tools. It also builds the storyline of the importance and relevance of Big Data analytics in IoT applications and pinpoints the limitations to overcome in order to support high-level graphical programming of Big Data applications. Chapter 3 breaks down the research goals **G1** and **G2** into three concrete objectives **O1 — O3**, provides a detailed discussion of the contributions made and approach followed in the thesis. The concrete objectives points to the chapters where they have been addressed by specific contributions discussed in those chapters. Chapter 4 discusses a new graphical flow-based programming tool concept to support development of applications with built-in stream processing capabilities. Chapter 5 and Chapter 6 discuss graphical programming of Spark and Flink respectively. Finally, Chapter 7 concludes the thesis and gives the author's subjective opinions and insights from the derivatives of the investigation done as part of the thesis work.

2 Background

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."

— Grace Hopper

This chapter summarises the associated libraries, infrastructures and technologies available for Big Data analytics. It gives an overview of the emerging field of IoT and why Big Data analytics in the context of IoT is essential. In the same time, end-user development in the domain of IoT application plays a crucial role where graphical flow-based programming tools called *'mashup tools'* have been used to reduce application development efforts. A similar approach can be used to support high-level programming of Big Data applications. The chapter presents an overview of the state-of-the-art IoT mashup tools currently in use and also highlights their current architectural limitations, supporting their inadequacy to support high-level programming of Big Data applications.

Data Science has become a much hyped term in the last years. What makes this term even more interesting is that academicians and industry lab researchers do not agree on a common definition. In addition to this, there is a distinct lack of respect for the decades of work done by researchers, whose work is based on work done by mathematicians as well as statisticians for decades before the term was popularised by technology industries. On a very superficial level, it appears to be reselling of simple statistics and mathematical algorithms and not a broad field by itself [139]. In addition, simultaneous usage of the term 'Big Data' along with Data Science often raises questions on their actual definition and application. This chapter attempts to bring a clear perspective of the terms and help build the foundations required to understand the research contributions made in the thesis.

2.1 Data Science

The current age has seen an increased production in the volume of data from many sources, covering almost all aspects of our lives like internet shopping, browsing, online searches, tweets etc. In the context of IoT, devices are equipped with a wide range of sensors which collect data almost round the clock and transmit this, i.e. we kind of have

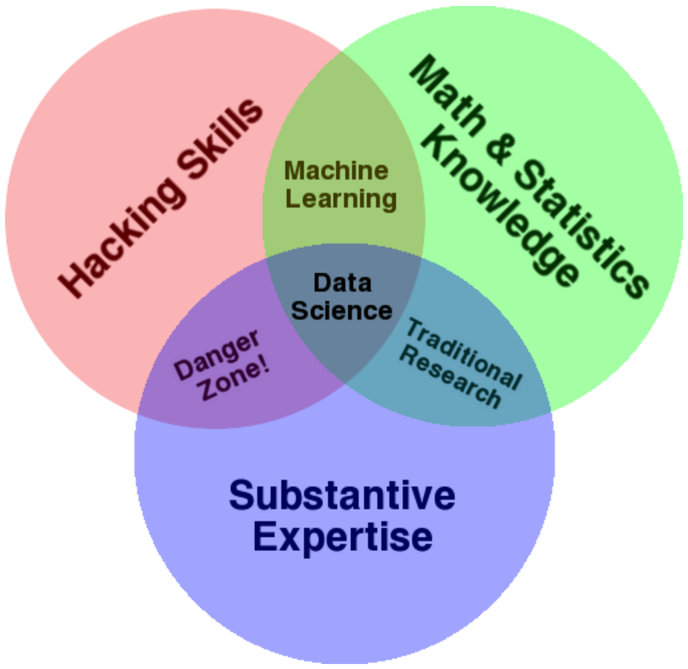


Figure 2.1: Drew Conway's Venn diagram of data science, as in [151]

infinite streams of data pouring into our systems continuously. The data thus generated is continuously monitored and this process is not knew. But the current age has given rise to a phenomenon popularly known as *'Datafication'*, i.e. the data is subjected to learning techniques to gain insight of the individual as a species or learning in-depth about the external environment. Alternatively, it can be defined as a process of "taking all aspects of life and turning them into data [58]." For instance, LinkedIn [105] datafies professional networks while Twitter [154] datafies random thoughts. Datafication is an interesting concept which essentially transforms the purpose of datafied things and churns out new forms of value [58]. Drew Conway's Venn diagram nicely summarises Data Science as shown in Figure 2.1. His explanation for various components involved in Data Science is as follows:

Hacking Skills *Data is a commodity traded electronically, therefore, in order to be in this market you need to speak hacker. Far from 'black hat' activities, data hackers must be able to manipulate text files at the command-line, thinking algorithmically, and be interested in learning new tools.*

Machine Learning *Data plus math is machine learning, which is fantastic if that is what you - if that is what you are interested in, but not if you are doing data science.*

Math & Statistics Knowledge *Once you have acquired and cleaned the data, the next step is to actually extract insight from it. You need to apply appropriate math and statistics methods, which requires at least a baseline familiarity with these tools.*

Traditional Research *Substantive expertise plus math and statistics knowledge is where most traditional researcher falls. Doctoral level researchers spend most of their time acquiring expertise in these areas, but very little time learning about technology.*

Substantive Expertise *Science is about discovery and building knowledge, which requires some motivating questions about the world and hypotheses that can be brought to data and tested with statistical methods. Questions first, then data.*

Danger Zone *This is where I place people who, 'know enough to be dangerous', and is the most problematic area of the diagram. It is from this part of the diagram that the phrase 'lies, damned lies, and statistics' emanates, because either through ignorance or malice this overlap of skills gives people the ability to create what appears to be a legitimate analysis without any understanding of how they got there or what they have created.*

2.2 Doing Data Science

A typical Data Science program model looks as shown in Figure 2.2.

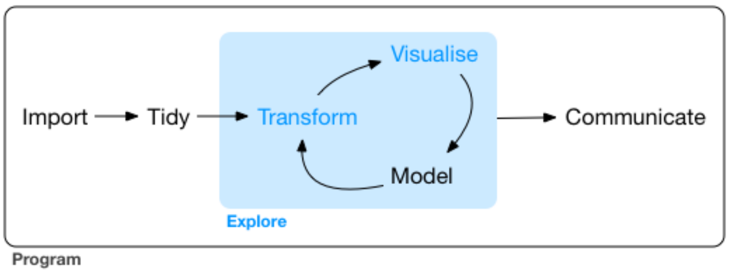


Figure 2.2: Data science program approach, as in [163]

It essentially involves:

1. Importing data from files, databases, web APIs into data structures supported in programs like R or Python.

2. After the data has been imported, it is subjected to clean-up, i.e. elimination of duplicates and storing it in appropriate data structural formats such that it is compatible with the transformation functions.

3. Cleaned-up data is passed to transformation functions which help to narrow-down the dataset, i.e. zoom into an area of observational interest. These functions can create toned-down versions of datasets which can be stored in new variables which in turn can be passed as arguments to new transformation functions. These functions can also be used to create a set of summary statistics.

4. The transformed data is used to generate knowledge/insights about the original dataset. Broadly, there are two approaches:

    - Graphical visualisation of data via plots of different kinds which is both intuitive and raises questions easily in case of any discrepancy in the transformed dataset. The downside of this approach is that it does not scale well and is often prone to displaying unanticipated results, i.e. result which the user was not expecting.

    - Models are mathematical tools which can answer specific questions and they scale well. A model works on the basis of certain assumptions and therefore, cannot produce unanticipated results.

5. The last part is effective communication of the insights gained from the visualisation and modelling techniques to others.

2.3 (Big) Data Analytics

Big Data is an umbrella term to refer to the new approaches that are needed to properly manage and analyse these ever-growing amounts of information, which differ from traditional data in the so-called 4V's: volume, variety, velocity and veracity [55]. Each of these properties entails a different challenge that Big Data platforms must address: they must be capable of handling great amounts of data that come from different sources (hence with different structures) and do it quickly enough while, at the same time provide results that really matter. In many use cases like stocks transactions, user interactions with social networks, exchange of messages between end-user terminals and mobile base stations, or events in smart cities infrastructures (e.g. pollution monitoring, traffic control, lamp posts, etc.) data occur as a series of events that follow a flow-like structure. In the last decade, a number of Big Data analytics tools have emerged to satisfy different business needs such as targeted advertising, social network analysis, sentiment analysis, malware analysis and others. From a technical perspective, Big Data analytics can however be divided into two modes:

Batch Mode It essentially deals with manipulating and querying in parallel large amounts of data residing in clusters of commodity machines.

Streaming Mode It involves accommodating and analysing large amounts of incoming data as they come.

These two modes co-exist in the lambda architecture [113], where the outputs from the two processing modes are combined in a serving layer before delivering the final result. We describe the different components of a Big Data Analytics system in the following sub-sections.

Big Data Processing Systems Big Data processing systems typically rely on a multitude of high-level languages, programming models, execution engines and storage en-

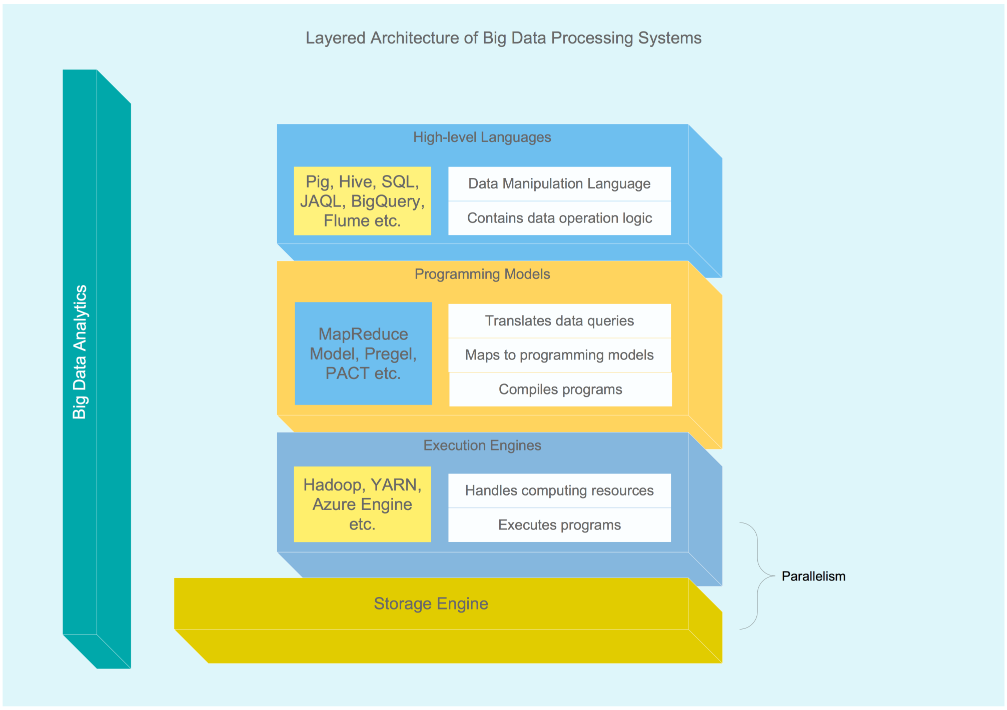

Layered Architecture of Big Data Processing Systems
Big Data Analytics
High-level Languages
Pig, Hive, SQL, JAQL, BigQuery, Flume etc.
Data Manipulation Language
Contains data operation logic
Programming Models
MapReduce Model, Pregel, PACT etc.
Translates data queries
Maps to programming models
Compiles programs
Execution Engines
Hadoop, YARN, Azure Engine etc.
Handles computing resources
Executes programs
Parallelism
Storage Engine

Figure 2.3: Four layered architecture of Big Data processing systems

gines. This typically forms a four-layered architecture as shown in Figure 2.3. The *high-level* languages allow a non-technical person to specify a data query or processing algorithm in a data manipulation program. Examples of typical high-level languages are Pig [27], Hive [20], SQL, JAQL [42, 38], Flume [21], BigQuery [80] etc. The *programming model* is the heart of the big data processing system. MapReduce [23] is one of the most widely used programming model. The queries in high-level language are translated and represented in a suitable programming model and passed onto the execution engine for execution. The *execution engine* is responsible for handling computing resources and executing the programs expressed in a particular programming model. Typical execution engines are ‘Yet Another Resource Negotiator’ (YARN) [13], Flume Engine [21], Tera Data Engine [50, 9, 49], Azure Engine [44, 46, 45] etc. The programs during execution make use of the *storage engine* to actually retrieve datasets, perform operation on them and return result sets. Google File System (GFS) [77], HDFS, Voldemort [158], Tera Data Store [50, 9, 49] are some of the most widely used storage engines.

2.4 Batch Analytics

Batch analytics systems rely on collecting data before they are analysed, so they simplify the whole process notably. However, they require huge storage facilities, as they are not capable of handling data as they arrive. Most of the currently used Big Data paradigms, like *MapReduce* [63], are based on the batch processing. Batch processing behaves reasonably well when assessing large volumes of data is the priority, especially regarding bounded datasets. However, they fail in providing good results with low latency, thus being inappropriate for real-time systems. Some popular solutions built on top of MapReduce to provide batch analytic capabilities include Pig, Hive, Spark etc. It is possible to somehow adapt MapReduce to data streams, by breaking the data into a set of finite batches. This is known as *micro-batching* and it is what Spark Streaming does [68].

2.4.1 Hadoop

Hadoop is an open source framework [22] to write scalable applications involving a large number of networked systems primarily meant for handling massive datasets and their associated computation. It is written in Java and runs inside the Java Virtual Machine (JVM). It makes sense to leverage the power of Hadoop infrastructures when we have to deal with petabytes of data [145]. It was created by Yahoo employee Doug Cutting and the inspiration came from the concepts of Google File System (GFS) [77]. In the recent versions, Hadoop supports a resource management layer like the YARN [134]. The core of Hadoop is the open-source implementation of the MapReduce programming model, which works with datasets spread across multiple networked systems managed by the Hadoop Distributed File System. It is the de-facto standard for all Big Data processing systems.

2.4.1.1 The Hadoop Distributed File System

When a dataset outgrows the storage capacity of a single physical machine then it becomes necessary to partition it across a number of separate machines. The filesystems which manage the storage across a network of separate machines are called *distributed filesystems.* The Hadoop Distributed File System (HDFS) is a distributed file system written in Java that can be run on low-cost commodity hardware preferably running GNU/Linux. It is designed for storing very large files with streaming data access patterns, i.e. it is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern [161]. It offers a high degree of fault-tolerance. Most Hadoop projects use HDFS as the file handling system, making it the foundation of all Hadoop based infrastructures [23]. Figure 2.4 shows the HDFS architecture, how it splits dataset into blocks and spreads it across a large number of networked machines. HDFS stores data in blocks which are saved in multiple nodes based on a user-defined replication factor (by default 3). During retrieval, HDFS tries to minimise the distance to a data block by selecting a node that is close to the client. This leads to a higher throughput. Every data chunk has a checksum and this can be requested by the client to ensure the integrity of the entire dataset. HDFS typically has the following components:

NameNode It is the single master node of a HDFS set-up. It manages the namespace of the filesystem and controls file access of clients. Every file is divided into blocks (64 MB by default) which are then stored across different DataNodes as managed by the NameNode. When a client requests for a specific file, the NameNode knows where its associated blocks are stored. It informs this information to the client and the client requests to the specific DataNodes to send the blocks of data. It is also responsible for monitoring the status of DataNodes and in case of any failure, it

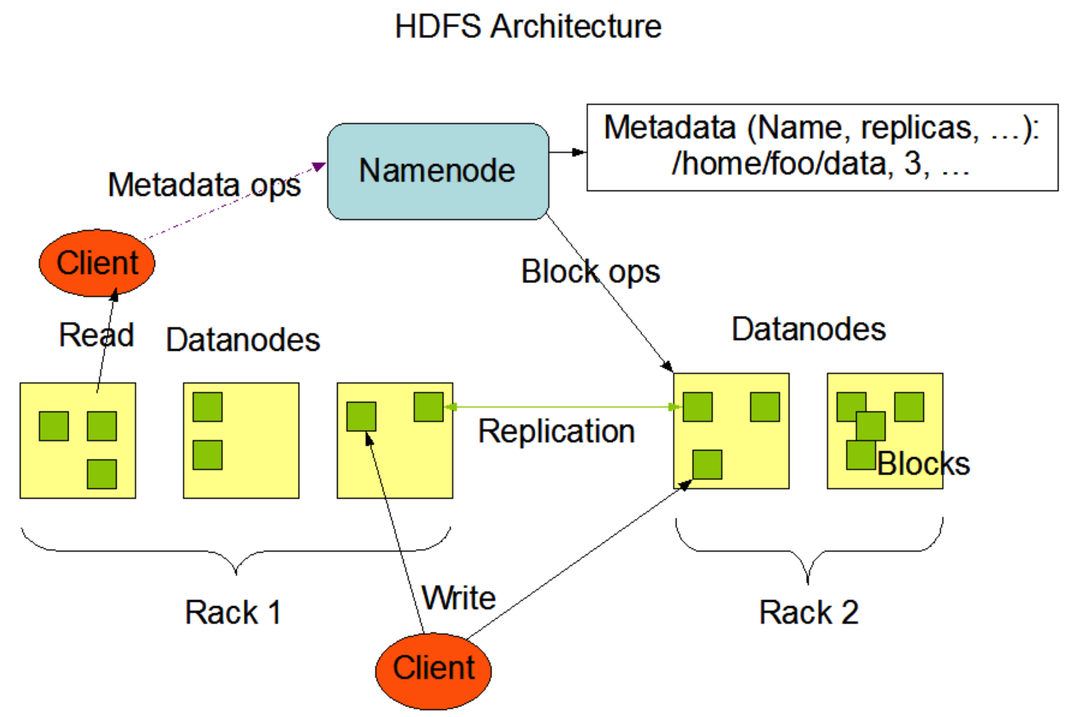


Figure 2.4: HDFS architecture, as in [24]

redoes the replication of lost data blocks to ensure that the replication factor of data is always maintained.

DataNodes These are the slave nodes and they can be more than one in number. These are responsible for managing the local storage and catering to client data requests.

The problem of ensuring that the data replicates are coherent when spread across many DataNodes is resolved by restricting the user to write to a file only once or append to it. In case of any changes, the user is forced to delete the file and write it again from scratch.

2.4.1.2 MapReduce

MapReduce is a programming model for data processing [161], originally developed by Google [63]. Hadoop can run MapReduce programs written in a wide range of programming languages like Java, Ruby, Python and C++. It provides an interface to write fault-tolerant highly parallel applications that can be run on a huge amount of nodes and work on petabytes of data [23]. It offers scheduling, monitoring and fault-safety to the developer who creates MapReduce applications [145].

The main idea is to use the map and reduce/fold from functional programming. MapReduce breaks the processing into two distinct phases: *the map phase* and *the reduce phase*. Each phase has key-value pairs as input and output, the types of which are defined by the programmer. The programmer also specifies two functions: the *map function* and the *reduce function*. The map function is applied to all elements of a dataset which can be done in parallel as HDFS supports chunking/dividing data into blocks, i.e. each node in the HDFS can run a map function on the block of data available on that node. The results of all the map functions are passed to the reduce function, which produces the final result of the computation. In case of MapReduce, it is perfectly valid if the map functions produce zero, one or more outputs and similarly the reduce function can produce multiple results for a single input. This is in stark contrast to traditional functional programming paradigm which also makes MapReduce more flexible. However, it is not suitable for smaller programs as the task has to be divided and run across multiple nodes which increases overhead considerably.

2.4.2 Hive

Hive is a data warehousing solution built on top of Hadoop. Its main goal is to simplify the querying and analysis tasks in Hadoop by providing a familiar SQL-like syntax for performing these tasks. Hive alleviates the problem of writing custom MapReduce (MR) programs that are hard to maintain and reuse and allows non-programmers to interact with Hadoop for reporting and ad-hoc data analysis.

Hive provides an SQL-like declarative language called HiveQL for specifying queries. Queries are internally compiled into MR programs and executed on a Hadoop cluster. In particular, Hive supports data definition statements for creating tables, data manipulation (DML) statements such as load, and typical SQL statements such as select, join, union, group by, order by, etc. Database schemas are kept in a system catalogue called metastore, which is physically stored in a relational database. As HDFS is not optimised for the use cases of a relational database, Hive combines HDFS with the fast random access from well known databases like MySQL or a local file system in a component called metastore. When working with Hive, a user can create tables schemas and load data to them from files in the HDFS. Hive supports reading and writing in a number of serialisation formats including CSV and JSON.

Once a query is issued, it gets translated into an execution plan. In case of data definition language (DDL) statements, the plan consists only of metadata operations, while LOAD statements are translated to HDFS operations. In case of INSERT statements and regular queries, the plan consists of a directed-acyclic graph of MR jobs, which gets executed in the Hadoop cluster.

2.4.3 Pig

Pig is a scripting layer on top of Hadoop MR. It can be used as alternative to Hive for simplifying the querying and analysis tasks. However, whereas Hive targets data analysts with SQL expertise, Pig targets mainly developers with procedural programming expertise.

Pig provides a procedural query language called Pig Latin. A Pig Latin program is a sequence of statements, each of which specifies only a single data transformation. Statements are constructed with the use of SQL-style high-level data manipulation constructs, e.g. JOIN, GROUP, ORDER, DISTINCT, FILTER, FOREACH and others. As an important difference to SQL, where only flat tables are allowed, Pig Latin has a nested data model that allows non-atomic data types such as tuple, set and map to occur as fields of a table. This provides more intuitive and flexible programming abstractions. Apart from using its built-in constructs, Pig allows users to provide User-Defined Functions (UDF), typically written in Java, that extend the functionality of Pig.

A Pig Latin program essentially can be represented by a directed acyclic graph where nodes represent data transformations and links represent data-flow. This is called logical plan. Logical plans get translated to physical plans, which in turn get translated to MR jobs by the Pig compiler.

2.5 Stream Analytics

The idea of processing data as streams, i.e. as they come in, is different from batch processing. The latter approach was followed in the first Big Data-processing systems, such as in Hadoop's MapReduce and in Apache Spark, which mainly dealt with reliable parallel processing of Big Data residing in distributed file systems, such as Hadoop's HDFS. Stream processing of Big Data has been recently sought as a solution to reduce the latency in data processing and provide real-time insights (e.g. on the scale of seconds or milliseconds).

In particular, an ideal stream-processing platform should meet the following requirements [147]:

- **Low latency**. Streaming platforms usually make use of *in-memory* processing, in order to avoid the time required to read/write data in a storage facility and thus decrease the overall data-processing latency.

- **High throughput**. Scalability and parallelism enable high performance in terms of data-processing capability. The real-time performance of stream-processing systems is frequently demanded even with spikes in incoming data [74].

- **Data querying**. Streaming platforms should make it possible to find events in the entire data stream. Typically, SQL-like language is employed [147]. However, since data streams never end, there needs to be a mechanism to define the limits of a query; otherwise it would be impossible to query streaming data. This is where the *window* concept takes part. Windows define the data in which an operation may be applied, so they become key elements in stream-processing.

- **Out-of-order data**. Since a streaming platform does not wait for all the data to become available, it must have a mechanism to handle data coming late or never arriving. A concept of *time* needs to be introduced, to process data in chunks regardless of order of arrival.

- **High availability and scalability**. Stream processors will most likely handle ever-growing amounts of data and in most cases, other systems could rely on them, e.g. in IoT scenarios. For this reason, the stream-processing platform must be reliable, fault-tolerant and capable of handling any amount of data events.

The first approaches to stream processing, notably Storm and Spark Streaming, used to focus on requirements such as low latency and high throughput [93]. *Lambda architecture*, a well-known approach [74, 113, 99] combines batch and stream-like approaches to achieve shorter response times (on the order of seconds). This approach has some advantages, but one critical downside: the business logic needs to be duplicated into the stream and the batch processors. In contrast to this, stream-first solutions, such as Apache Flink, meet all the outlined requirements [74].

2.6 Spark

Spark 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. There have been many approaches to cluster computing like MapReduce [62], Message Passing Interface (MPI) [73] etc. Spark Programming model has been designed to overcome the limitations of MapReduce [168]. It's excellent in-memory computation capabilities are good for scenarios which demand iterative computations, e.g. application of machine learning techniques, which involves application of an algorithm repeatedly on a 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 provides a SQL-like interface for data analysis, GraphX can be used for graph computations and different Machine Learning libraries to learn from datasets.

Spark in implemented in Scala [137] but provides APIs in Scala, Java as well as Python. 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 [170].

2.6.1 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. But 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'*. The terms *'Spark Application program'* and *'Spark Driver program'* essentially convey the same message. In this thesis, the term *'Spark Driver program'* has been used extensively.

2.6.2 Spark Fundamentals

Spark is a distributed computing platform engine which is under heavy 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 most important component in the Spark ecosystem is the Spark Core, which provides basic functionalities for running Spark jobs. Since, Spark allows us to use different libraries it is very different from traditional Hadoop technologies. Spark can be studied from two different perspectives: (i) as a distributed analytics engine, (ii) as a library providing different libraries and APIs for Big Data analytics. One of the strongest shortcomings of MapReduce is that job results need to be saved before they can be used by another job. Spark's core concept is an in-memory execution model that enables caching job result in memory instead of fetching it every time from the hard disk. Consider an example [171], where we have stored the city map data as a graph. The vertices of this graph represents 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:

1. Calculate the shortest path between all the vertices.
2. Find the *farthest point distance*, i.e. maximum distance to any other vertices for every vertex.
3. 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 succeeding step. But in Spark, all these can be computed in-memory using the concept of caching. Here in this section, the fundamental concepts of Spark as a distributed analytics engine has been discussed.

Resilient Distributed Dataset The most important functionality provided by Spark core for running Spark jobs is the *'resilient distributed dataset'* (RDD) [168]. RDDs are 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 RDD can be used by another worker machine to recompute the lost computation. RDDs are computed from other RDDs by applying coarse-grained transformations [168] or by reading user input from disk. RDDs are not required to be stored in physical memory as they can be recomputed any time. However, when necessary, users can persist data represented by an RDD in memory. Figure 2.5 shows a typical RDD operation in Spark. For example [171], there is a 300 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 a 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 distributed collection and takes care of communication as well as node failure issues [171].

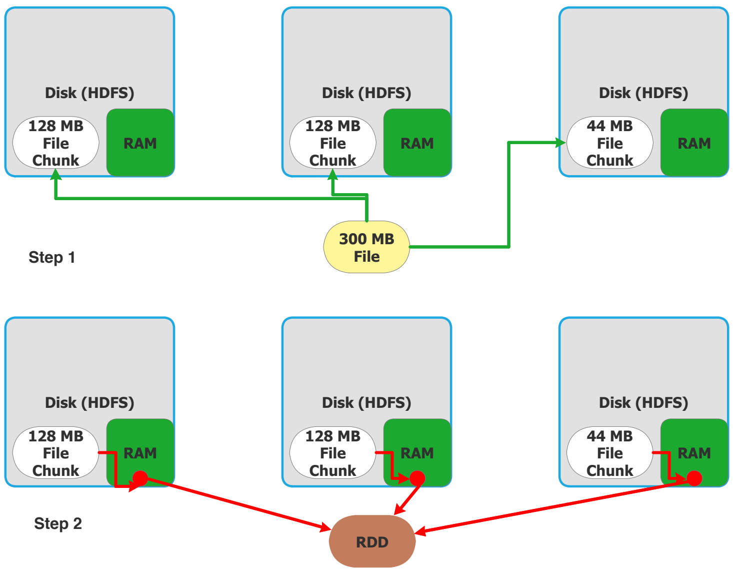


Figure 2.5: RDD operations in Spark, following [171]

Spark driver process 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 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 all control over the resources required to orchestrate, control the execution and manage the worker processes. Reading of 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 to the driver process. Figure 2.6 shows the communication pathways between the Spark driver program and worker processes and a typical execution pattern of a Spark application program.

Distributed Execution Model Spark supports cluster computing via Mesos[88], YARN[156] and a built-in standalone[29] mode. Spark applications can be deployed in either client or cluster mode. In client mode, Spark driver program runs as a child process of the cluster manager and in cluster mode Spark driver is run on one of the worker nodes. Cluster mode of deployment is safer compared to client as in the client mode if a Spark driver program crashes then cluster manger crashes along with it and the application must be re-deployed. However, in the cluster mode, cluster manager remains unaffected and may launch another worker node to restart the execution. Spark driver and worker nodes communicate several times during the execution.

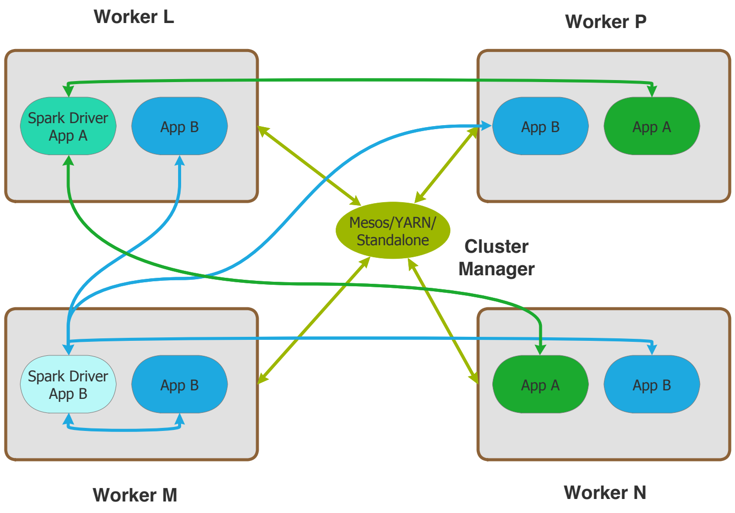


Figure 2.6: Spark application program: execution pattern in cluster mode

Evaluation & Execution Style When a Spark application is submitted for execution, it is first inspected by Spark driver program to generate an execution map. Only computations whose output is sent out of the Spark execution environment are scheduled for actual execution. This is known as *'Lazy Evaluation'*. Advantages of lazy evaluation is that computations which do not end in sinks, i.e. push-data out, are not executed thereby reducing computations.

2.6.3 Spark Ecosystem

Spark offers many libraries centred around Spark core to cater to different aspects and requirements of Big Data analytics. Recent developments in the programming model of Spark have increased its flexibility and power. Here, we describe some of the libraries of the Spark ecosystem. Figure 2.7 gives a graphical summary of the different libraries and their runtime interactions.

Spark SQL Although Spark with its built-in RDD abstractions provides fault-tolerance, but over a period of time several shortcomings of Spark core were identified. One such shortcoming was that Spark core treats data as an unstructured stream. Streams of data read from external sources such as files, for example, is mapped into Java/Python objects which is inefficient for structured and semi-structured data as they are present in different file formats such as 'CSV', 'JSON' etc. Applying relational queries on data read from external sources or using machine learning and graph processing of Spark on relational data is not feasible with this approach. Spark SQL is a library which was developed to address this shortcoming.

Spark SQL addresses the gap between procedural model of Spark and relational model of structured and/or semi-structured data by providing DataFrame API[34]. DataFrames

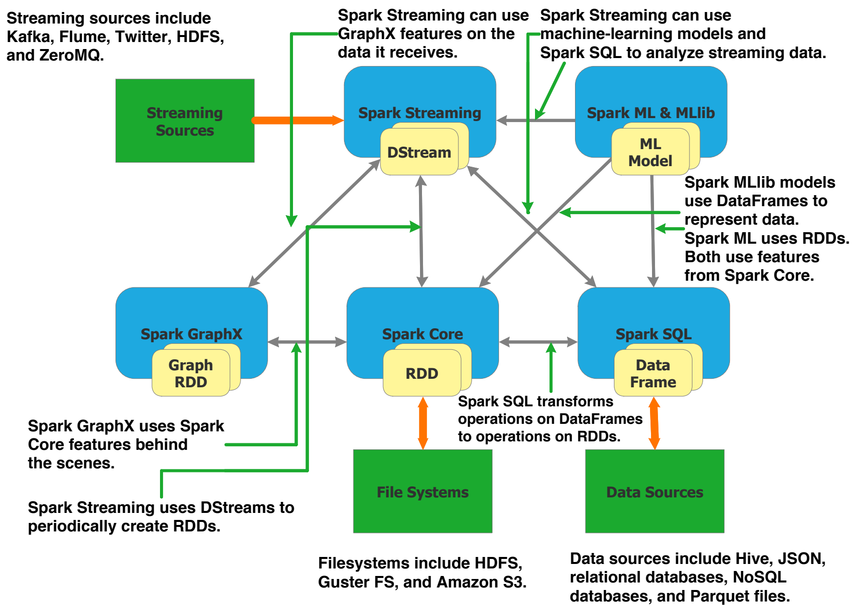

Streaming sources include Kafka, Flume, Twitter, HDFS, and ZeroMQ.
Spark Streaming can use GraphX features on the data it receives.
Spark Streaming can use machine-learning models and Spark SQL to analyze streaming data.
Streaming Sources
Spark Streaming
DStream
Spark ML & MLlib
ML Model
Spark MLlib models use DataFrames to represent data. Spark ML uses RDDs. Both use features from Spark Core.
Spark GraphX
Graph RDD
Spark Core
RDD
Spark SQL
Data Frame
Spark SQL transforms operations on DataFrames to operations on RDDs.
Spark GraphX uses Spark Core features behind the scenes.
Spark Streaming uses DStreams to periodically create RDDs.
File Systems
Data Sources
Filesystems include HDFS, Guster FS, and Amazon S3.
Data sources include Hive, JSON, relational databases, NoSQL databases, and Parquet files.

Figure 2.7: Different Spark components, various runtime interactions and storage options, following [171]

store user input in compact columnar format which is more efficient compared to data stored in objects. In addition Spark SQL introduced Catalyst, an optimisation engine on top of Spark which uses features of Scala programming language to optimise user query by generating composable rules in a Turing complete language[34]. Spark SQL builds a logical plan for the DataFrame operations which is evaluated eagerly, i.e. to identify if column names used in transformation is present in the DataFrame or not. When operations involving generating results are encountered, physical plan is generated including optimisations from Catalyst[34] and gets executed on the worker nodes. Figure 2.8 shows how using a Data Frame API in Spark along with a supplied schema, data-read from various sources can be stored in the form of a tabular structure, thereby allowing relational queries.

Spark Streaming *'Discretized Streams'* (DStreams) is a library developed to overcome the limitations of continuous operator model of processing streaming data [169]. Programming model of DStreams is to "structure streaming computations as a series of stateless, deterministic batch computations on small time intervals" [169]. DStreams use RDD abstractions to provide fault tolerance, stateless and deterministic computations on streaming data. Programming model of DStreams groups real-time streaming data into micro-batches for applying operations provided by Spark Core on them. This is advantageous in scenarios where machine learning models prepared from historical datasets are applied on streaming data. However, it does not provide event-time based processing features or late data handling capabilities. Figure 2.9 shows the programming model of Spark Streaming which creates incremental RDDs with the continuous inflow of datasets.

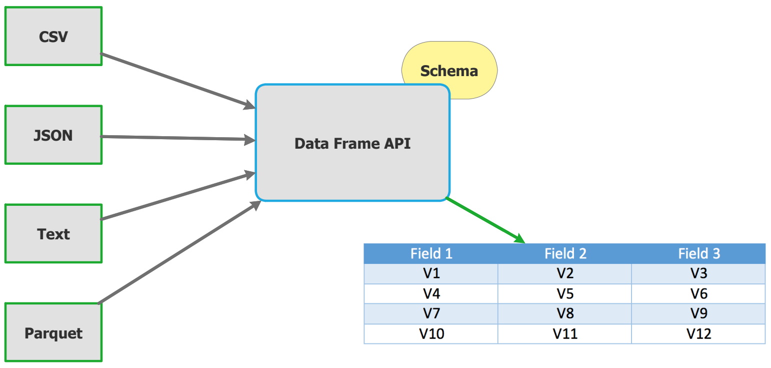


Figure 2.8: DataFrame API produces a tabular structure

Spark Structured Streaming Spark Structured Streaming is a library built on Spark SQL engine and uses DataFrame abstractions for processing data. Its programming model handles streaming data as an unbounded table which grows infinitely [26]. Every new data received from streaming source is appended as a new row to the unbounded table. Computations provide incremental updates to existing results. For instance, a running count on fields A and B is stored in a results table. This results table is updated in each processing interval (event-time based or processing time based). Structured Streaming provides DataFrame APIs to express computations which is very similar to computations specified on static data. SQL engine applies the computation incrementally over the unbounded table and results are updated accordingly [26]. Streaming computations are evaluated eagerly but executed lazily. Structured Streaming supports window operations based on processing time and event-time of the data.

Although, the programming model of Spark Structured streaming is good for streaming aggregations, it is incompatible with APIs of Spark ML library. For instance, feature transformation APIs which map categorical values to continuous values or machine learning algorithms such as logistic regression which perform iterations over the entire dataset cannot be used with Spark Structured streaming. Structured streaming aggregations are executed as incremental queries on the unbounded table and fail in scenarios where iteration over entire dataset is necessary.

Trigger Time Streaming aggregations are applied on unbounded table at a frequency indicated by user in the form of *'trigger time'* in every streaming query. It supports following kinds of triggers[26]:

Immediate Processing Streaming data is processed immediately if workers are idle. Otherwise, data is grouped into a micro-batch and aggregations are applied on the batch once workers have finished their ongoing computations.

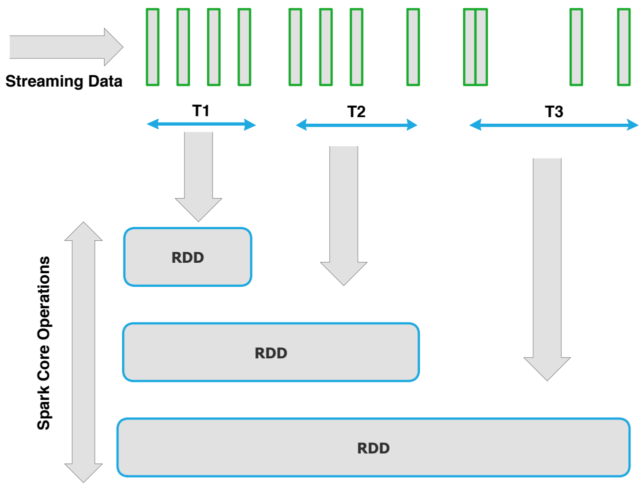


Figure 2.9: Spark Streaming programming model

Fixed-interval Processing Processing is triggered at a time interval specified by user. Data collected between two consecutive time-intervals is treated as a micro-batch and it is subjected to computations if it has data in it.

One-time Processing Processing is triggered only-once on completion of data read operation from a static source like HDFS.

Handling of Late Data In real-world scenarios, streaming data is delivered to stream processing engines through message brokers such as Kafka[28, 102], Kinesis[10] etc. Event-time based window operations perform aggregations by grouping data on the basis of time stamp present in received messages. It is fairly common for messages to arrive late. Therefore, the stream processing engine provides a mechanism for the user application to handle late data which requires the stream processing engine to store data/state for longer periods of time than usual. Stream processing applications are usually long running applications and therefore, streaming data cannot be stored infinitely. This also requires a purging mechanism as the system can run out of memory very soon! Spark Structured Streaming handles late data through *'watermarking[26]'*. Watermark indicates the duration up-to which a data can arrive late beyond which it is not considered for processing.

Spark Machine Learning Spark supports distributed machine learning via:

Spark MLlib Spark MLlib, has been built on top of Spark Core using the RDD abstractions, offers a wide variety of machine learning and statistical algorithms. It supports various supervised, unsupervised and recommendation algorithms. Supervised learning algorithms include *decision trees*, *random forest* etc., while some of the unsupervised learning algorithms supported are *k-means clustering*, *support vector machine* etc.

Spark ML Spark ML is the successor of Spark MLlib and has been built on top of Spark SQL using the DataFrame abstraction. It offers Pipeline APIs for easy development, persistence and deployment of models. Practical machine learning scenarios involve different stages with each stage consuming data from preceding stage and producing data for the succeeding stage. Operational stages include transforming data into appropriate format required by the algorithm, converting categorical features into continuous features etc. Each operation involves invoking declarative APIs which transform DataFrame based on user inputs[115] and produce a new DataFrame for use in the next operation.

2.7 Flink

Apache Flink is a processing platform for distributed stream as well as batch data. Its core is a streaming data-flow engine, providing data distribution, communication and fault tolerance for distributed computations over data streams [150]. It is a distributed engine [96], built upon a distributed runtime that can be executed in a cluster to benefit from high availability and high-performance computing resources. It is based on stateful computations[74]. Indeed, Flink offers exactly-once state consistency, which means it can ensure correctness even in the case of failure. Flink is also scalable because the state can be distributed among several systems. It supports both bounded and unbounded data streams. Flink achieves all this by means of a distributed data-flow runtime that allows a real-stream pipelined processing of data [96].

A streaming platform should be able to handle time because the reference frame is used for understanding how the data stream flows, that is to say, which events come before or after another. Time is used to create windows and perform operations on streaming data, in a broad sense. Flink supports several concepts of time (Figure 2.10):

Event time It refers to the time at which an event was produced in the producing device.

Processing time It is related to the system time of the cluster machine in which the streams are processed.

Ingestion time It is the wait time between when an event enters the Flink platform and the processing time.

Windows are a basic element in stream processors. Flink supports different types of windows and all of them rely on the notion of time as described above. Tumbling windows have a specified size, and they assign each event to one and only one window without any overlap. Sliding windows have fixed sizes, but an overlap, called the slide, is allowed. Session windows can be of interest for some applications, because sometimes it is insightful to process events in sessions. A global window assigns all elements to one single window. This approach allows for the definition of triggers, which tell Flink exactly when the computations should be performed.

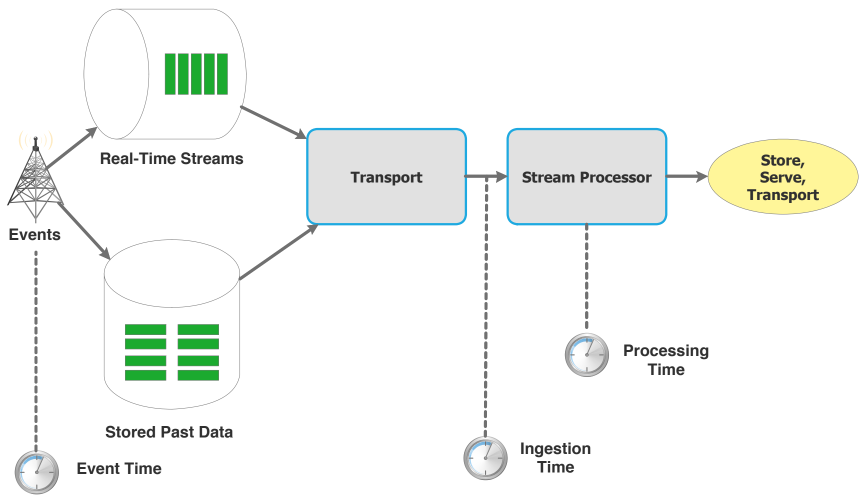


Figure 2.10: Different concepts of time in Flink, as in [119]

2.7.1 Flink Ecosystem

The Flink distributed data-flow programming model together with its various abstractions for developing applications, form the Flink ecosystem. Flink offers three different levels of abstraction to develop streaming/batch applications as follows:

Stateful stream processing The lowest level abstraction offers stateful streaming, permitting users to process events from different streams. It features full flexibility by enabling low-level processing and control.

Core level Above this level is the core API level of abstraction. By means of both a DataStream API and a DataSet API, Flink enables not only stream processing but also batch analytics on ‘bounded data streams’, i.e. datasets with fixed lengths.

Declarative domain-specific language Flink offers a Table API as well, which provides high-level abstraction to data processing. With this tool, a dataset or data stream can be converted to a table that follows a relational model. The Table API is more concise, because instead of the exact code of the operation, defined logical operations [150] are less expressive than the core APIs.

In the latest Flink releases, an even-higher-level SQL abstraction has been created as an evolution of this declarative domain-specific language. In addition to the aforementioned user-facing APIs, some libraries with special functionality are built. The added value ranges from machine learning algorithms (currently only available in Scala) to complex event processing (CEP) and graph processing.

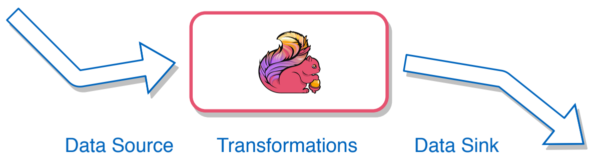


Figure 2.11: Overall structure of a Flink program [150]

2.7.2 Flink Application: Internals & Execution

The structure of a Flink program (especially when using the core-level APIs) begins with data from a source entering Flink, where a set of transformations is applied (window operations, data filtering, data mapping etc.). The results are subsequently yielded to a data sink, as shown in Figure 2.11. A Flink program typically consists of streams and transformations. Simplistically, a stream is a never-ending flow of datasets, and a transformation is an operation on one or more streams that produces one or more streams as output.

On deployment, a Flink program is mapped internally as a data-flow consisting of streams and transformation operators. The data-flow typically resembles directed acyclic graphs (DAGs). Flink programs typically apply transformations on data-sources and save the results to data-sinks before exiting. Flink has the special classes DataSet for bounded datasets, and DataStream for unbounded data-streams, to represent data in a program. To summarise, Flink programs look like regular programs that transform data collections. Each program consists of:

1. Initialising the execution environment.
2. Loading datasets.
3. Applying transformations.
4. Specifying where to save the results.

Flink programs use a lazy evaluation strategy, i.e. when the program's main method is executed, the data loading and transformations do not happen immediately. Rather, each operation is added to the program's plan, which is executed when its output needs to be used immediately.

2.8 Internet of Things

Internet of things (IoT) has been defined as the interconnection of ubiquitous computing devices for the realisation of value to end users [36]. This includes data collection from the devices for analysis leading to better understanding of the contextual environment as well as automation of tasks for optimisation of time and enhancing the quality of human life to the next level. IoT has already pierced into fields like health care, manufacturing, home automation etc. [66]. But to truly exploit the possibilities offered by IoT is to rapidly enhance the application landscape.

2.8.1 Importance of Data Analytics in IoT

In a world of connected devices, there will be a huge amount of data that will be constantly recorded and used for real-time and/or historical analysis. Analytics of the IoT data generated is gaining prominence, as this leads to immediate uncovering of potentially useful insights. Big Data technologies can be employed in this context to generate insights. Such analytics can lead to important insights regarding individual and group preferences and patterns of end-users (e.g. mobility models), the state of engineering structures (e.g. as in structural health monitoring), 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 in 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.

2.8.2 Role of End-User Development in IoT

IoT solutions have the potential to add value to different aspects of human life and environment. In several scenarios identifying IoT use cases requires significant domain knowledge at the same time developing solutions for identified scenarios requires programming skills. Often, it is the case that domain experts have a good understanding of the problem but have little or nil programming skills required for prototyping the use case. Hence, one of the challenges in IoT is to enable domain experts who could be non-programmers to design and prototype IoT solutions. Graphical programming tools designed with *'end-user development'* as one of the design goals have a significant role in bridging this gap in expertise and domain knowledge.

End User Development as defined by EUD-Net "is a set of activities or techniques that allow people, who are non-professional developers, at some point to create or modify a software artefact" [57]. Given the fast paced development of new technologies and libraries it is hard for professional developers to develop expertise in different competing technologies. Apache Spark and Apache Flink for example, are contemporary platforms providing Big Data analytics. There is a significant learning curve and involves considerable investment of time. In such situations, graphical tools designed with end-user in mind are useful for quick prototyping of Big Data applications.

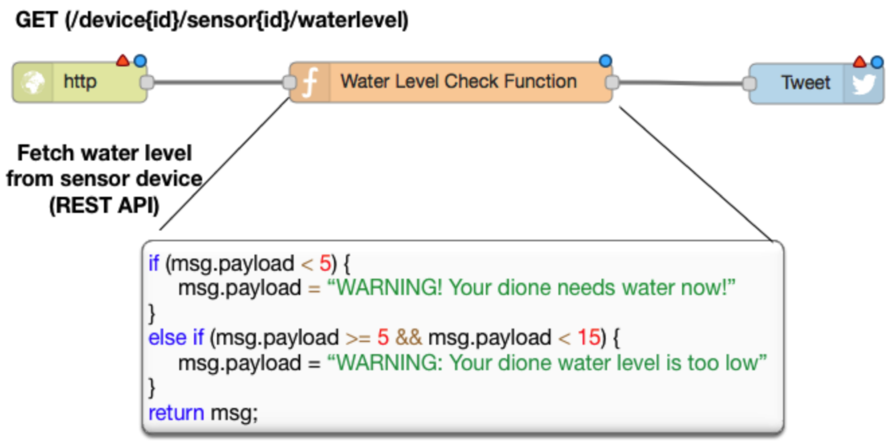
2.8.3 Application Development for IoT

The development of IoT applications is not a straightforward process because developers have to write complex code to access the datasets from the sensors of different devices and also perform data mediation before actually using the data in applications. Special graphical tools called IoT mashup tools have been proposed as a way to simplify this. Mashup tools typically support a graphical interface to specify the control-flow between different sensors, services and actuators [110, 130]. The resulting application follows the flow-based programming model, where outputs from a node in the flow become inputs of the next node, i.e. they offer a dataflow-based programming paradigm where programs form a directed graph with 'black-box' nodes which exchange data along connected arcs.

2.9 Mashups: Enabling End-Users

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. The mashup logic is the internal logic which defines how a mashup operates or how the mashup components have been orchestrated [126]. It specifies which components are selected, the control-flow, the data-flow and data mediation as well as data transformation between different components [61]. 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. Figure 2.12 shows a typical mashup application (created in Node-RED [92]). In addition, this also highlights the typical outlook of a mashup application, i.e. the flow-based programming paradigm it follows. The example of the figure first fetches data from a REST API, then, checks for certain conditions in the second component, and, finally, the control moves to the third component to initiate actions corresponding to the input received from its preceding component.

Mashups are quite broad and are generally classified based on their composition, domain and the environment. Composition of a mashup extensively deals with the kind of components that make it up. The application stack has been broadly classified into data, logic and presentation (user interface) layer. The mashup created accordingly is called either a data, logic or user interface mashup. Similarly, domain of a mashup explains the functionality of a mashup like social mashups or mobile mashups etc. Lastly, the environment explains the context where it is deployed. For instance, it can be web mashups or enterprise mashups. The difference between web and enterprise mashups is very subtle and it is not the area of focus here. But it would be sufficient to know that web mashups are generally targeted for end users on the Internet while enterprise mashups are specifically used in business contexts. These need to adhere additional security guidelines and other business specific requirements which the normal web mashups need not adhere to [61].


GET (/device{id}/sensor{id}/waterlevel)
http
Water Level Check Function
Tweet
Fetch water level from sensor device (REST API)

```
if (msg.payload < 5) {
    msg.payload = "WARNING! Your dione needs water now!"
}
else if (msg.payload >= 5 && msg.payload < 15) {
    msg.payload = "WARNING: Your dione water level is too low"
}
return msg;
```

Figure 2.12: Model of a mashup application in Node-RED, as in [130]

2.9.1 Mashup Components

Mashup components are the building blocks of a mashup. In practice, several technologies and standards are used in the development of mashup components. Simple Object Access Protocol (SOAP) web services [133], RESTful web services, JavaScript APIs, Really Simple Syndication (RSS) [107], Comma-Separated Values (CSV) [159] etc. are some of the prominent ones. Depending on their functionality the mashup components have been broadly classified into three categories (Figure 2.13):

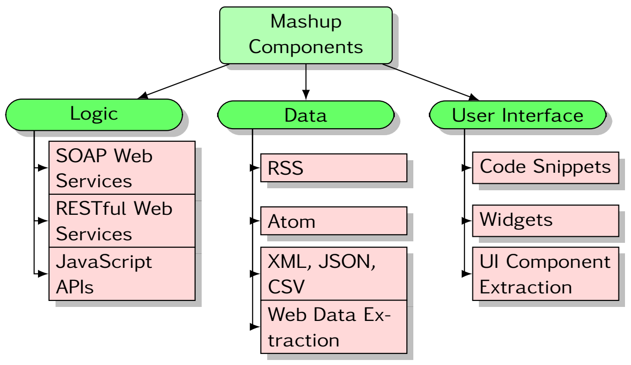


Figure 2.13: Classification of mashup components, following [61]

1. Logic components provide access to functionality in the form of reusable algorithms to achieve specific functions.

2. Data components provide access to data. They can be static like (RSS) feeds or dynamic like web services which can be queried with inputs.

3. User interface components provide standard component technologies for easy reuse and integration of user interfaces pieces fetched from third-party web applications with in the existing user interface of the mashup application.

2.9.2 Classification of Mashups

Mashups are classified according to the compositional model [167] which basically governs how the components are orchestrated to form a mashup application. Based on the compositional model, we can classify mashups as:

Output Type A composition consisting of a number of mashup components can finally provide either data, logic or user interface as output.

Orchestration Style Orchestration style governs how the execution of various components in a mashup flow are synchronised. Accordingly, we have: (i) *flow-based* styles define orchestration as a sequence or partial order among tasks, i.e. following the patterns of flowchart like formalisms, (ii) *event-based* styles rely on publish-subscribe model for attaining tighter synchronisation between various components, (iii) *layout-based* style specifies that the components be arranged in a common layout. The behaviour of a component is governed individually by other component's reaction to user interactions.

Data Passing Style Governs how data is passed from one component to another in a mashup. Accordingly, we have: (i) *data-flow approach*: in this case, data flows from one component to another, (ii) *blackboard approach*: in this case, data is written in variables which form the source as well as target of operation invocation just like normal programming languages.

Composition Execution Pattern This governs how the mashup composition model is executed in the back-end. Accordingly, we have: (i) *instance-based*: in this case, when a new message arrives, an instance of the component is instantiated within the same thread of the composition and the message is handled, (ii) *continuous-based*: in this case, one instance of every component is instantiated in separate threads where they process the input message and send output to the next component running in a separate thread.

2.9.3 Producing an Application from a Mashup Composition

After a mashup has been orchestrated by the end-user, the main goal is to generate the final application. In this case there are two widely used approaches: (i) *Model Driven Development* (MDD): Here the focus is on re-usability of the generated code from the model, (ii) *End-User Development*(EUD): Here the focus is to abstract the implementation detail so that minimal coding skills are required from the end-user. The code generation technique receives inputs from the end-user and generates the mashup application for the target platform. Accordingly, it must also integrate user supplied custom codes in the target application and the code generation technique must be extensible to support incorporation of new features as well as generic to cover most aspects of the target platform. The minimal requirements which a code generation technique [146] needs to fulfil are:

1. A meta-model or implicit abstract syntax which governs how the components can be orchestrated on the front-end by the user.

2. A set of transformations which are build on the meta-model.

3. A mechanism to read the user-input and deliver the specifications to the transformations and generate the target application.

There are a large number of code generation techniques available like *'templates and filtering'*, *'templates and metamodel'*, *'frame processors'*, *'API based generators'*, *'in-line generation'*, *'code attributes'*, *'code weaving'* etc. [146]. Of these, the API based code generation technique is the most popular one [146]. Typically, these code generators provide extensive APIs through which various elements of the target language can be generated. Since, the code generation is done using a meta-model of the target language, the generated code is always syntactically correct. However, it is restricted to generating code in one target language only. JavaPoet [94] is one such API based code generator for Java.

2.10 Flow-based Programming (FBP)

It is a programming paradigm invented by J. Paul Rodker Morrison in late 1960s which uses data processing pathways to design user applications [120]. It defines user applications as networks of 'black-box' processes communicating via data chunks travelling across well-defined pathways [121]. Conventional programming paradigm, i.e. 'control-flow' programming, typically concentrates on processes and gives secondary preferences to data. In contrast to this, business applications are typically concerned with how data is processed and handled, i.e. have very strict requirements on how the data moves in the system. An example here would illustrate the fundamental differences in both approaches to problem solving. For instance, consider a program which needs to be developed to read records and if these records match a certain criteria they are send for further processing/output or else they get discarded. A typical function written in 'control-flow' programming paradigm would have the following structure (Listing 2.1), i.e. taking care of needed records and ignoring other records. On the contrary, in strict flow-based programming paradigm the function would look something like Listing 2.2.

```
read into a from IN
  do while read has not reached end of file
    if c is true
      write from a to OUT
    endif
    read into a from IN
  enddo
```

Listing 2.1: Handling data in control-flow programming paradigm (as in [121])

```
receive from IN using a
  do while receive has not reached end of data
    if c is true
      send a to OUT
    else
      drop a
    endif
    receive from IN using a
enddo
```

Listing 2.2: Handling data in data-flow programming paradigm (as in [121])

The Listing 2.2 can be represented as a black-box, i.e. a node while the 'IN' and 'OUT' can be represented as ports of the node via which the node consumes input and yields output. A connected pathways between such data processing black-boxes forms the core of flow-based programming paradigm. The Listing 2.2 also introduces some preliminary concepts fundamental to flow-based programming paradigm which are:

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. Typically, a dataset is immutable.

Data-flow graph The directed graph formed by considering the connection between different nodes via valid pathways.

Execution of a 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 as '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 'push' mechanism, a node pipes its output as soon as it is available while in 'pull' mechanism a node typically pulls its input from its preceding node when required.

It is special case of dataflow programming. Dataflow programming paradigm specifies that data-flow controls the execution pathway of a program [48]. The Actor Model [3] 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 [48]. 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 and are processed concurrently by the receiving actor. It is an asynchronous dataflow mode with nodes, i.e. individual actors, being strictly functional in design.

2.11 IoT Mashup Tools

Mashup tools help the users to develop a mashup application. They typically have a graphical editor permitting the user to model how the control flows between a set of components. A description of how a typical mashup looks like will make things clear. For instance, consider that the weather data is available with the help of REST APIs. A user wants to get this data, apply some transformation and post it to twitter. The mashup depicting the flow for this scenario is given in Figure 2.14. The 'Fn.' block in the figure contains code (business logic, depicted by the 'if' block) which accomplishes the data transformations. The orchestration of 3 components namely data from web, a function block and a tweeter block clearly depicts how the control flows through them to fulfil the application objective. These components are generally represented by GUI blocks in a mashup tool which have to be connected suitably to represent the entire business logic. *These tools are based on the flow-based programming paradigm.* Some of the most prominent IoT platforms which also house a mashup tool for service composition include glue.-things [100], Thingstore [8], OpenIoT [98], ThingWorx [66], Paraimpu [127], Xively [66] etc. Node-RED is a visual programming environment developed by IBM which supports the creation of mashups. It is very popular these days. However, it is important to note that Node-RED is not a complete IoT platform by itself as it does not support device registration and management. glue.things uses an improved version of Node-RED as a mashup-environment along with device management features.

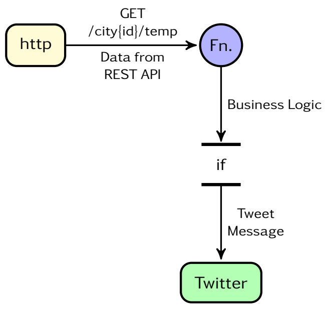


Figure 2.14: Outline of a typical mashup

2.11.1 Node-RED

Node-RED is an open-source mashup tool developed by IBM and released under Apache 2 license. It is based on the server side JavaScript platform framework Node.js[1] (that is why the 'Node' in its name). It uses an event-driven, non-blocking I/O model suited to data-intensive, real-time applications that run across distributed devices.

Node-RED provides a GUI where users drag-and-drop blocks that represent components of a larger system which can either be devices, software platforms or web services that are to be connected. These blocks are called nodes. A node is a visual representation of a block of JavaScript code designed to carry out a specific task. Additional blocks (nodes) can be placed in between these components to represent software functions that manipulate and transform the data during its passage [84].

[1] https://nodejs.org/

Two nodes can be wired together. Nodes have a grey circle on their left edge, which is their input port and a grey circle on their right edge represents their output port. To connect two nodes, a user has to link the output port of one node to the input port of the other node. After connecting many such nodes, the finished visual diagram is called a flow.

IoT solutions often need to wire different hardware devices, APIs, online web services in interesting ways. The amount of complex code that the developer has to write to wire such different systems, e.g. to access the temperature data from a sensor connected to a device's serial port or to manage authentications using OAuth [56], is typically large. In contrast, to use a serial port using Node-RED, all a developer has to do is to drag on a node and specify the serial port details. Hence, with Node-RED the time and effort spent on writing complex code is greatly reduced and the developer can focus on the business parts of the application.

Node-RED flows are represented in JSON and can be serialised, in order to be imported as new nodes to Node-RED or shared online. There is a new concept of 'sub-flows' that is being introduced into the world of Node-RED. Sub-flows allow creating composite nodes encompassing complex logic represented by internal data flows.

Since in Node-RED nodes are blocks of JavaScript code, it is — technically — possible to wrap any kind of functionality and encapsulate that as a node in the platform. Indeed, new nodes for interacting with new hardware, software and web services are constantly being added, making Node-RED a very rich and easily extensible system. Lastly, note that the learning curve to develop a new node for the platform is low for Node.js developers since a node is simply an encapsulation of Node.js code.

To make a device or a service compatible with Node-RED, a native Node.js library capable to talk to the particular device or service is required. However, with the growing acceptance of REST style in Web and IoT systems, more and more devices and services provide RESTful APIs that can be readily used from Node-RED.

2.11.2 glue.things

The objective of 'glue.things' is to build a hub for rapid development of IoT applications. 'glue.things' heavily employs open source technologies for easy device integration, service composition and deployment [100]. TVs, phones and various other home/business tools can be hooked up to this platform through a wide range of protocols like Message Queue Telemetry Transport (MQTT) [149], Constrained Application Protocol (CoAP) [149] or REST APIs over HTTP.

The development of mashup applications in glue.things roughly goes through three stages [100].

Firstly, the devices are connected to the platform to make them web accessible using protocols like MQTT, CoAP or HTTP/TCP etc. Device registration and management is handled by the 'Smart Object Manager' layer in the glue.things architecture as explained in sub-section below 2.11.2.1. REST APIs provide communication capabilities and JSON

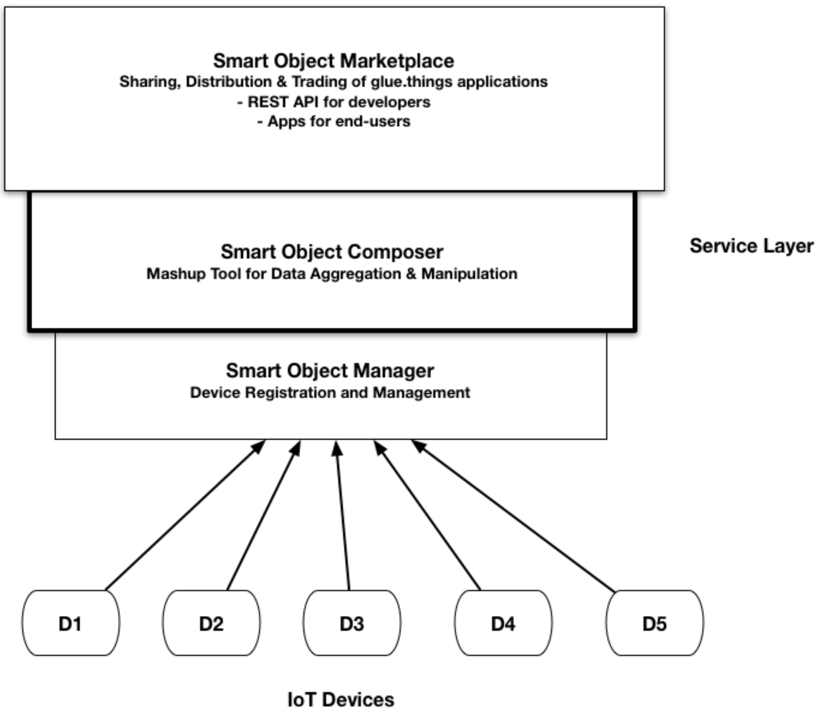

Smart Object Marketplace
Sharing, Distribution & Trading of glue.things applications
- REST API for developers
- Apps for end-users
Smart Object Composer
Mashup Tool for Data Aggregation & Manipulation
Service Layer
Smart Object Manager
Device Registration and Management
D1
D2
D3
D4
D5
IoT Devices

Figure 2.15: glue.things architecture [100]

data model is used for propagating device updates. These facilities are leveraged using the client libraries or for a more intuitive experience of device addition the web based dashboard can be used. The dashboard also features several templates for connecting devices and simplifying the tasks for the developer.

The second stage deals with creation of mashups. glue.things uses an improved version of Node-RED as a mashup tool to collect data streams from connected devices and combine them. This improved version supports multi-users, sessions and automatic detection of new registered device and makes them available on the panel. External web services like Twitter, LinkedIn etc. can also be used during mashup composition. The 'Smart Object Composer' layer in the glue.things architecture houses the mashup tool as explained in detail in sub-section 2.11.2.1.

Lastly, the created mashups are deployed as Node-RED applications including various triggers, actions and authorisation settings. These deployed mashup applications are accessible by RESTful API to the developers who may want to use them in their own custom web applications. To the normal end users, they can be browsed through a collection of mashup applications which can be used after suitable alterations to the connection settings and other environment specific values. Sharing of these mashup applications is also supported by the platform. This functionality is reflected in the 'Smart Object Marketplace' layer in the architecture.

2.11.2.1 glue.things Architecture

Figure 2.15 shows the simplified architecture based on the detailed architecture of the platform. This can be segregated into three distinct layers, namely the Smart Object Manager, the Smart Object Composer and the Smart Object Marketplace.

Smart Object Manager This layer integrates real-time communication networks to easily access a large number of IoT devices. These networks support messaging with real-time web sockets via Remote Procedure Call (RPC), MQTT and CoAP. There is also a device directory to search and query for any device on the Internet. This layer is extensible, meaning any future real-time communication network/gateway can be integrated into the platform.

Smart Object Composer This layer provides mechanisms for data and device management. The mashup development environment is build on Node-RED and is used for service composition. Mashups are JSON objects in combination with a Node.js-based work-flow engine. This layer also has a virtualised device container for managing the registered devices.

Smart Object Marketplace This layer contains all the created and deployed applications. These applications can be shared, distributed or traded. Developers can access them via REST APIs to embed them in a new application. End users can access these as normal applications.

The application layer contains all the user interfaces for device registration, configuration and monitoring. A dashboard combines all these UI in a coherent front-end accessible by both users and developers alike.

2.11.3 Other IoT Mashup Tools

Other IoT mashup tools designed to simplify application development include WoTKit [43], EcoDiF [64], IoT-MAP [85], OpenIoT [98], ThingStore [8], IoTLink [128], M3 Framework [83], ThingWorx [66] and Xively [66]. A detailed discussion on their core functionalities has been published as a report [111].

2.11.4 High-level Programming for Big Data Applications via IoT Mashup Tools

Mashup tools, based on the flow-based programming paradigm, should be enhanced in a number of directions to support high-level programming of Big Data applications as they are not designed for data analytics. The prevalent architectural limitations of single-threaded and blocking execution semantics needs to be addressed. First, they should allow developers to create components with non-blocking semantics and asynchronous communication. Second, they should allow the specification of multiple threads of operation for a single mashup. These two points will make it possible to specify components that incorporate Big Data analytics tasks, have their own life-cycle and act as 'callbacks' for receiving the analysis results and propagating them to the rest of the mashup. Third, mashup tools should incorporate visual programming of not only the data-flow in the system, but also of the Big Data analysis jobs. This will allow seamless modelling of applications involving Big Data analytics. Finally, mashup tools should provide support for both enterprise usage (code generation, extensibility requirements etc.) and data science tasks, such as visual inspection of datasets. This will facilitate their adoption by both enterprise developers and experienced data scientists.

2.12 Related Work

We did not find any mashup 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 and is designed to be used in the context of IoT, i.e. ingest data produced from IoT sensors and run analytics on them. However, there are many different solutions to reduce the challenges involved in using specific Big Data frameworks. This section lists the existing works and solutions aimed to support high-level graphical programming of Big Data applications as well as flow-based programming concepts for data analytics.

IBM Infosphere Streams IBM Infosphere®Streams is a platform designed for Big Data stream analytics and uses IBM Streams Processing Language (SPL) as its programming language [89]. It is designed to achieve high throughput as well as shorter response times in stream analytics. The key idea is to abstract the complexities in developing a stream processing application and the aspects of distributed computing by allowing the user to use a set of graph operators. The application developed can be translated automatically to C++ and Java. SPL treats the application flow as a streams graph where the edges represent continuous streams and vertices represent stream operators. Stream operators are either transformers, sources or sinks. It is not designed as a visual-flow based language though it models the application in the form of a graph for reasons of expressiveness [143]. It is a complete language and not a stream processing library within a non-streaming language in order to have improved type checking and optimisation. SPL has two main elements in its language construct, i.e. streams and operators. Operators without any input streams are called as sources while operators without any output streams are called as sinks. The operators have their own threading and get executed when there is at-least one data item in their input stream, i.e. in the edge of the stream graph. The data-items leave an operator in the same sequence in which they had arrived after processing. SPL is issued to develop streaming applications as well as batch applications because both the computing paradigms of Big Data are implemented by data-flow graphs. Additionally, SPL allows to define composite operators in order to support programming abstraction and enable development of application involving thousands of operators. It has a strong static type system and minimises implicit type conversions. Every operator can specify the behaviour of its ports, i.e. port mutability. For instance, an operator can define that it does not modify data items arriving on its input port but may permit a downstream operator to modify the same. SPL also makes use of control ports in addition to input and output ports which are used in feed-back loop. There are three main paradigms for stream processing:

Synchronous data-flow (SDF) In this paradigm, every operator has a fixed rate of data-output items per input data-items, i.e. both the cardinality of input and output sets are static and well-known. Examples include StreamIt [152] and ESTEREL [41]. This paradigm is not efficient in real-world scenarios as the input and output sets cannot be known in advance and leads to optimisation issues.

Relational Streaming This paradigm models the relational model from databases and allows to use operators like select, join aggregate etc. on data-items. Examples include TelegraphCQ [53], the STREAM system underlying CQL (Continuous Query Language) [33], Aurora [1], Borealis [52, 47], StreamInsight [39] and SPADE (Stream Processing Application Declarative Engine) [76].

Complex Event Processing (CEP) This paradigm treats input streams as raw events and produces output streams as inferred events, i.e. uses patterns to detect and gather insights. Examples include NiagaraCQ [54] and the SASE (Stream-based and Shared Event processing) [5].

SPL is not based on SDF paradigm as it allows dynamic input and output rates for each operator. It is based on relational model and can support CEP with inclusion of a CEP library within an operator.

StreamIt StreamIt [152] is a dedicated programming language for writing streams application following the paradigm of synchronous data-flow stream processing. It allows to model the application in form of a graph where vertices are operators and edges represent streams. The most basic operator is a *'Filter'* which has one input and one output port. The rate of data ingestion as well as data production is static and pre-defined before the execution which is one of the major disadvantages and restricts its usage to real-world scenarios. Additional programming constructs like *'Pipeline'*, *'SplitJoin'* and *'FeedbackLoop'* are used in conjunction to a 'Filter' to form a communicating network. A 'Pipeline' is used to define a sequence of streams while a 'SplitJoin' is used to split and join streams. Similarly, the 'FeedbackLoop' operator is used to specify loops in a stream. Representation of a stream application via arbitrary graphs, i.e. a network of filters connected via channels is difficult to visualise and optimise. The main advantage is that it imposes a well-defined structure on streams which ensures a well-defined control flow within the stream graph. It follows the constructs of flow-based programming paradigm but does not offer any high-level graphical constructs to write stream applications.

The QualiMaster Infrastructure Configuration (QM-IConf) tool The QM-IConf tool [70] supports model-based development of Big Data streaming applications. It introduces a high-level programming concept on top of Apache Storm [14]. It features a graphical-flow based modelling (Figure 2.16) of the streaming application in the form of a data-flow graph where vertices are stream operators and edges represent valid data-flow paths. A valid data-flow path from vertex $v1$ to $v2$ ensures that $v2$ can consume the data produced by $v1$. The data-flow 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.

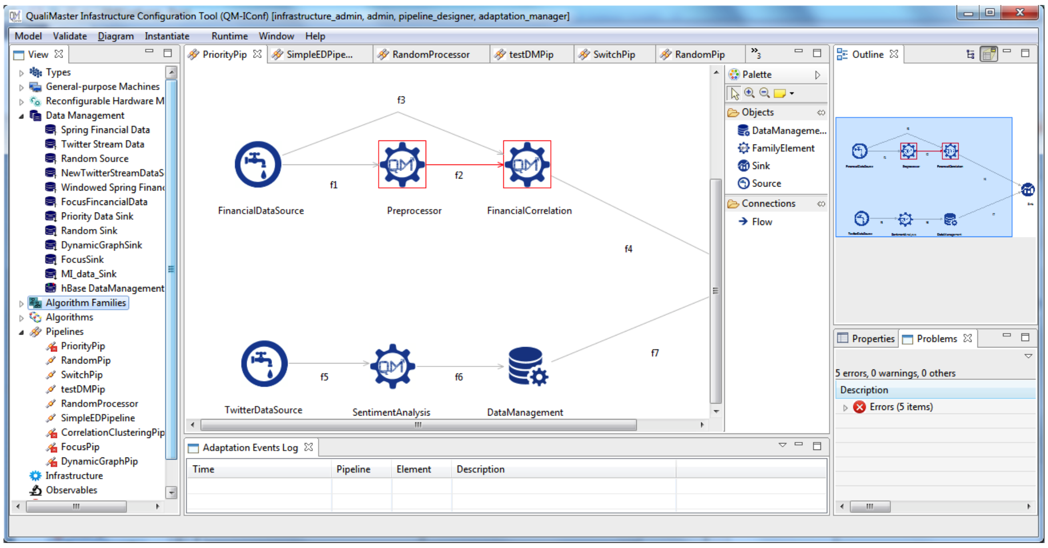


Figure 2.16: A graphical Big Data model in QM-IConf [70]

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 translate the graphical flow into a runnable Spark program [136]. 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 (Figure 2.17). A JSON object is prepared from the graphical flow and 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 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 Big Data cluster and is not designed to support analytics in the context of IoT. To summarise, it is a visual programming tool for Spark machine learning restricted to specific APIs and use cases.

QryGraph QryGraph [138] is a web-based tool which allows the user to create graphical Pig queries in the form of a flow along with simultaneous syntax checking (Figure 2.18) to support batch processing of datasets stored on HDFS. In QryGraph, a Pig query is rep-

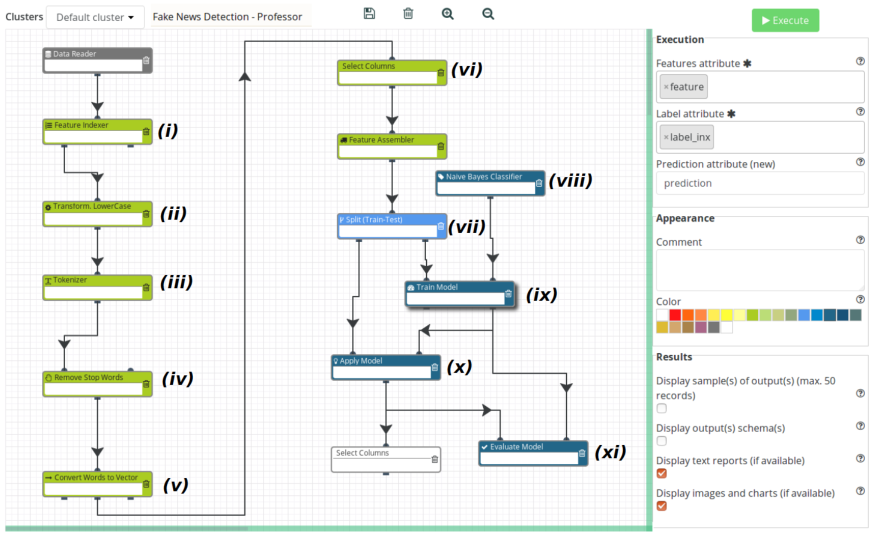


Figure 2.17: A graphical Spark flow in Lemonade [136]

resented as a data-flow graph with vertices representing data sources, sinks and transformers while edges represent valid data-flow pathways. It also allows the user to deploy the created job and manage its life-cycle.

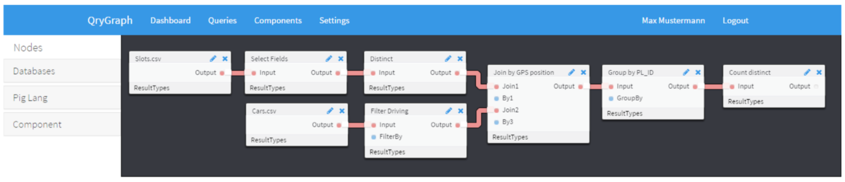


Figure 2.18: A graphical Pig flow in QryGraph [138]

Nussknacker It is an open-source tool[2] which supports model-based development of Flink applications. It also supports deployment and monitoring of Flink jobs [153]. 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 to transform 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 an 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 [12] is an 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

[2][Nussknacker] `https://github.com/TouK/nussknacker`

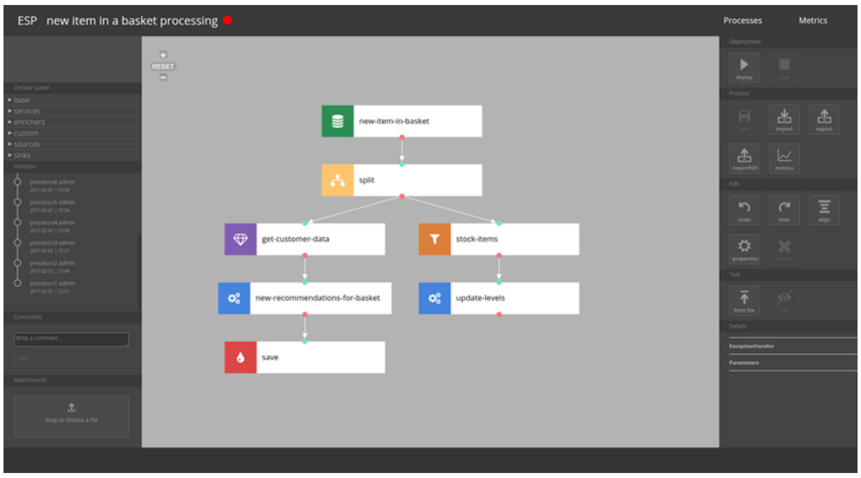


Figure 2.19: GUI of the Nussknacker tool [153]

pipeline models an application as a data-flow 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 native Big Data job and executes it in the target environment, i.e. the same beam program can be deployed on Spark, Flink, Apex [11] etc. clusters. The beam programming model support operators necessary for batch and stream operations and if a corresponding feature is supported by a target framework 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 [32] while consuming code snippets in Python/R. Nevertheless, to successfully interact with Spark, Zeppelin still requires programming skills from users since it requires 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 [116]. Nevertheless, it expects the user to have programming expertise.

IBM SPSS Modeller provides a graphical user interface to develop data analytics flows involving simple statistical algorithms, machine learning algorithms, data validation algorithms and visualisation types [91]. SPSS Modeller provides machine learning algorithms developed using Spark MLlib library which can be launched on Spark cluster by simply connecting them as components in a flow. Although SPSS Modeller is a tool built for non-programmers to perform data analytics using pre-programmed blocks of algorithms, it does not support writing new custom Big Data applications for any target framework.

Apache NiFi [19], 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 certainly provided.

We classify the related work into the following categories:

Category 1: flow-based data analytics/stream processing In this category, we categorise the related solutions of StreamIt, IBM Streams Processing Language, QM-IConf and IBM SPSS modeller.

Category 2: high-level programming for Big Data applications In this category, we categorise the related solutions of Lemonade, QryGraph, Nussknacker, Apache Zeppelin, Apache NiFi, Apache Beam and Microsoft Azure.

Section 4.5 in Chapter 4 gives a detailed comparison of the new mashup tools concepts for supporting flow-based data analytics with category 1 related solutions and discusses the improvements over the existing state-of-the-art while Chapter 7 compares the high-level graphical programming concepts for Big Data developed as part of this thesis work with existing solution falling in category 2.

3 Research Objectives and Approach

"Research is to see what everybody else has seen, and to think what nobody else has thought."

— Dr. Albert Szent-Györgyi

The original research goals **G1** and **G2** (set out in Section 1.3) are refined into the following three concrete objectives:

O1 Analyse Spark and Flink Big Data frameworks to (i) understand their programming model, (ii) extract suitable data abstractions and APIs compatible with the flow-based programming paradigm and (iii) model them as modular components.

O2 Accommodate the modular components of Spark and Flink in graphical flow-based programming tools, development of a generic approach to parse flows created with such components and generate the native Big Data program for data analytics.

O3 Identify the prevalent architectural limitations in the state-of-the-art mashup tools, design new concepts for graphical flow-based programming tools, its realisation to support creation of applications with concurrent execution semantics, support for scaling up of individual components in a modelled application and support for in-flow stream processing.

The concrete objectives **O1 — O3** are addressed in the following manner:

Regarding **O1**, to support graphical programming for Big Data via flow-based programming tools, it is required to integrate Big Data programming into flow-based programming paradigm. Big Data frameworks like Pig, Hive, Spark, Flink etc. are completely heterogeneous in nature when it comes to developing programs with them for data analytics. For instance, Pig allows a more scripting style while Hive supports SQL like queries. Spark has a number of different libraries for different functionalities like Spark GraphX [18] for graph computation, Spark Streaming to do stream analytics etc. Most of these libraries use different APIs and different data abstractions to store and transform data like DStream, DataFrame etc. All the different data abstractions are abstractions over the core data abstraction called the RDD. Nevertheless, all the data abstractions are not interoperable with each other. Hence, a thorough analysis of both Spark and Flink ecosystems is done to extract those data abstractions and APIs suitable to represent in a flow-based programming paradigm, i.e. *not supporting APIs requiring user defined data* *transformation functions or supporting code-snippets during flow creation to interact with target framework internals.* The selected APIs with their data abstractions formed a subset from the entire ecosystem. The APIs present in this small subset are modelled as modular components, i.e. a set of specific APIs bundled together and executed in a specific order to perform one data analytics operation such that the resultant components have high-cohesion with loose coupling.

The modular components are independent of the execution semantics of the implementing graphical programming tools and can be expressed in any flow-based programming tool with the requirement that these would form the basic unit of execution in the implementing tool. The objective is met by the following two contributions:

1. A thorough analysis of the Spark framework and selecting suitable data abstractions for use in a graphical flow-based programming paradigm (Chapter 5).

2. A thorough analysis of the Flink framework and identify the programming abstractions and APIs which are more amenable to be used in graphical flow-based programming paradigm (Chapter 6).

Objective **O2** basically deals with graphical programming from flow-based tools with the modular components distilled from the frameworks in objective **O1**. The objective is to develop a conceptual approach to parse a flow created from the distilled modular components and generate a compilable and runnable Big Data program. The approach should be easily extensible, i.e. to add support for new components and keep the code-generation process generic. This is met by the following two contributions:

1. Devising a novel, generic approach for programming Spark from graphical flows that comprises early-stage validation with feedbacks and code generation of Java Spark programs (Chapter 5).

2. The conceptual idea and the technical realisation of mapping a graphical flow designed in a flow-based programming tool to a Flink program and providing basic flow validation functionalities at the level of the tool (Chapter 6).

For **O3**, a thorough analysis of the existing graphical flow-based tools is done and their architectural limitations for supporting in-flow Big Data analytics are pinpointed. New concepts for graphical flow-based tools are designed based on the actor model and has been prototyped via a new flow-based tool called *aFlux*. Based on the actor semantics, every component used in a flow are actors which can be scaled up by defining a concurrency factor. These actors react when they receive a message in their mailbox and process the messages asynchronously thereby facilitating modelling of applications that are multi-threaded and have concurrent execution semantics in them. The actor model has also been adapted to support components which react on arrival of messages, process them and send their output to the next actor but do not stop their execution, rather they listen for messages continuously. This helps to include components which can stream data continuously in a flow to succeeding components thereby facilitating the creation of stream analytics applications. Additionally, factors which govern the performance of such stream analytic applications, for instance the buffer management options, i.e. how to regulate the buffer of a component when it is over-flooded with incoming streams of data, are user-configurable and the application generation, deployment complexities are abstracted from the end-user to support easy prototyping of stream analytics applications without getting into the nuances of it. The objective is met by two contributions as listed below:

1. Design of new graphical flow-based programming concepts based on the actor model with support for concurrent execution semantics to overcome the prevalent architectural limitations of mashup tools (Chapter 4).

2. Supporting built-in user-configurable stream processing capabilities for simplified in-flow data analytics. (Chapter 4).

It has been demonstrated that a flow-based programming model with concurrent execution semantics is suitable for modelling a wide range of Big Data applications currently used in Data Science.

The objectives **O1 & O2** which specifically deal with supporting high-level graphical programming for Big Data applications are evaluated via use cases to demonstrate the ease of use, code-abstraction from user, automatic conversion between different data abstractions and automatic Big Data program generation. The use cases have been prototyped using the outcome of the objective **O3** as a test-bed. Therefore, **O3** has been discussed first before discussing **O1 & O2**.

4 Flow-based Programming for Data Analytics

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."

— C.A.R. Hoare

This chapter introduces essential concepts to support data analytics in flow-based programming paradigm and caters to the research contributions by (i) design of new graphical flow-based programming concepts based on the actor model with support for concurrent execution semantics to overcome the prevalent architectural limitations of mashup tools, (ii) supporting built-in user-configurable stream processing capabilities for simplified in-flow data analytics. Parametrising the control points of stream processing in the tool enables non-experts to use various stream processing styles and deal with the subtle nuances of stream processing effortlessly. The effectiveness of parametrisation in simplifying stream analytics has been validated in a real-time traffic use case. **aFlux** [108, 109, 112] is a JVM based mashup tool prototyped to realise the proposed concepts. *The concepts have been explained by using the prototype as a running example.*

Despite promised benefits in having data analytics in graphical mashup tools, *there are several limitations of current approaches* [110, 111]. So far, mashup tools have been successful in supporting application development for Internet of Things. At the same time, Big Data analytics tools have allowed the analysis of very large and diverse datasets. But Big Data systems are complex to write applications for. High-level Big Data programming would help lower the learning curve similar to what mashup tools have done for IoT application development. Having graphical flow-based tools for data analytics and application development would be useful to prototype applications involving data analytics. Such scenarios should go beyond merely specifying IoT mashups that only act as data providers. *Mashup developers should also be able to specify Big Data analytics jobs and consume their results within a single application model.*

Supporting Big Data analytics in the flow-based programming of IoT mashup tools involves overcoming the existing architectural limitations in the current state-of-the-art IoT mashup tools such as:

Blocking execution and synchronous communication in mashups Mashups developed in current mashup tools have blocking execution and synchronous communication semantics. This effectively means that the execution cannot get transferred to the next component in a data flow before the logic of the current component gets executed. This becomes a limitation in cases where a mashup needs to run an analytic job in the background, while listening for further inputs from various sources like HTTP, Message Queue Telemetry Transport (MQTT) [149] etc. Since Big Data analytics jobs are typically time-consuming, there is a clear need for non-blocking semantics on the mashup that invokes it, so that it can continue its operation, and for asynchronous communication between the Big Data analytics tools and the mashup, so that the analytics results are communicated to the mashup when they become ready.

Single-threaded mashups Since most mashup tools use JavaScript technologies for application development and deployment, mashups developed with such tools are single-threaded. This can be a serious limitation when a mashup involves the execution of a number of Big Data analytics jobs. In such a case, invoking each of them in a separate thread can speed up the execution of the data analytics part by a factor equal to the number of jobs (assuming jobs with same duration). Besides integration with Big Data analytics tools, multi-threading in mashups would could be beneficial in cases involving heavy database querying and/or file IO where read or write latency is not negligible. With this, it should be possible to define concurrency factor of each individual component used in a mashup which is not supported in current mashup tools. This would speed up processing of individual components which is of paramount importance in case of stream processing as we might need to process data quickly if the load increases to reduce waiting time and ensuring quick response time of the application.

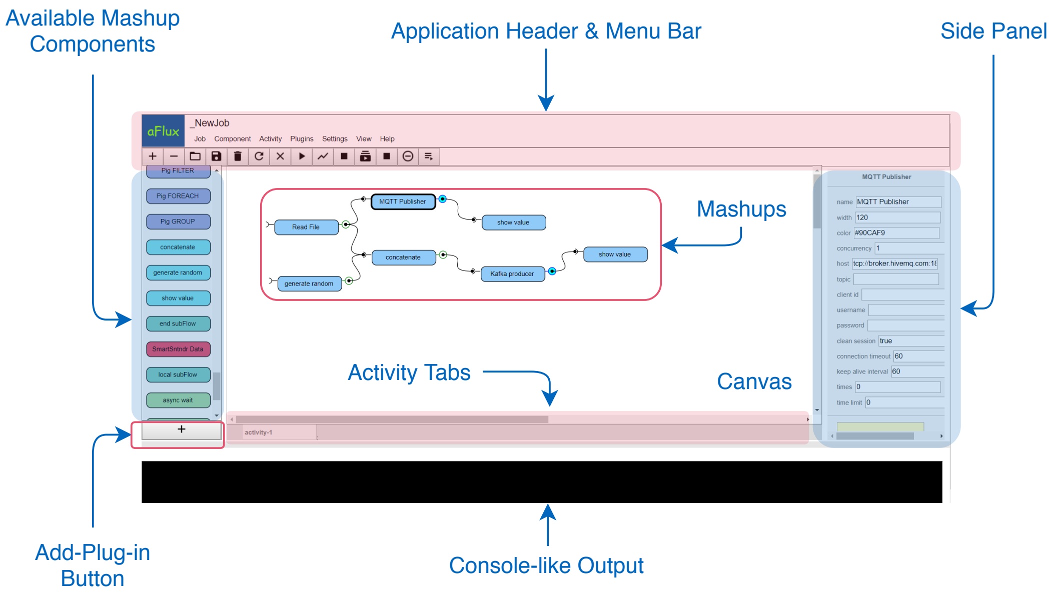
4.1 Essential Concepts to Support Flow-based Data Analytics

Existing tools allow users to design data flows which have synchronous execution semantics. This can be a major obstacle since a data analytics job defined within a mashup flow may consume great amount of time causing other components to starve or get executed after a long waiting time. Hence, *asynchronous execution patterns* are important in order for a mashup logic to invoke an analytics job (encapsulated in a mashup component) and continue to execute the next components in the flow. In this case, the result of the analytics job, potentially computed on a third party system, should be communicated back to the mashup logic asynchronously. Additionally, mashup tools restrict users in creating single-threaded applications which are generally not sufficient to model complex repetitive jobs. To summarise, the main motivation behind aFlux, i.e. coming up with improved design concepts for flow-based programming tools (a.k.a. mashup tools), are to support the following concepts:

1. concurrent execution of components in flows.
2. support for modelling complex flows via flow hierarchies (sub-flows).
3. support inbuilt stream processing.

4. model Big Data analytics via graphical flows and translate the flows to native Big Data programs.

aFlux is a an IoT mashup tool prototyped as part of this thesis work based on the proposed concepts that offers several advantages compared to existing solutions. It features a *multi-threaded execution model*, and *concurrent execution of components*. It primarily aims to support in-flow Big Data analytics when graphically developing services and applications for the IoT.



MQTT Publisher

name: MQTT Publisher
width: 120
color: #90CAF9
concurrency: 1
host: tcp://broker.hivemq.com:18
topic:
client id:
username:
password:
clean session: true
connection timeout: 60
keep alive interval: 60
times: 0
time limit: 0

Figure 4.1: Graphical user interface of aFlux [119]

aFlux consists of a web application and a back-end developed in Java and the Spring Framework[1]. The web application is composed of two main entities: the front-end and back-end, based on REST API. The front-end of aFlux (Figure 4.1) provides a GUI for the creation of mashups. It is based on React[2] and Redux[3] frameworks. Mashups are created by dragging-and-dropping available mashup components from the left panel on to the canvas and wiring them. New mashup components are loaded from *plug-ins* [119]. The application shows a console-like output in the footer, and the details about a selected mashup component are shown on the right-hand side panel. The 'Application Header & Menu Bar' contains functionalities to control the execution of a mashup like start execution, stop execution, saving the mashup etc. Using the aFlux front-end, a user can create a flow by wiring several mashup components (or sub-flows) together.

[1] `https://spring.io/`
[2] `https://reactjs.org/`
[3] `https://redux.js.org/`

4.1.1 Programming Paradigm

Based on previous analysis, we decided to go with the *actor model* [3, 86], a paradigm well suited for building massively parallel [4, 148], distributed and concurrent systems [157, 155]. Actors communicate with each other using asynchronous message passing [122]. The actor model was originally a theoretical model of concurrent computation [78]. The actor model is one of the ways of realising dataflow programming paradigm which is a special case of flow-based programming [48].

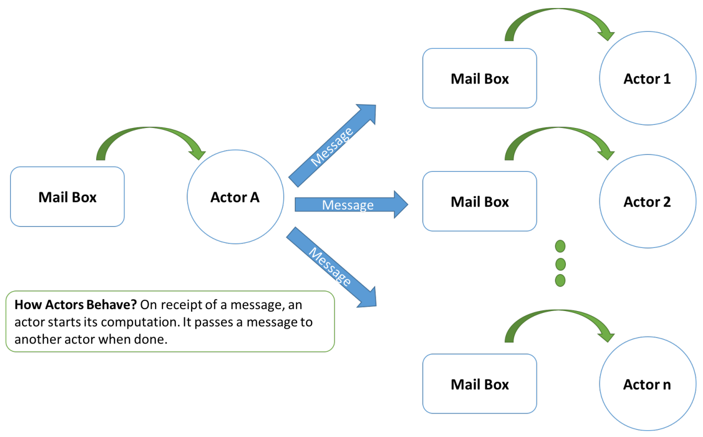

Mail Box
Actor A
Message
Message
Message
Mail Box
Actor 1
Mail Box
Actor 2
Mail Box
Actor n
How Actors Behave? On receipt of a message, an actor starts its computation. It passes a message to another actor when done.

Figure 4.2: Actor model: working

In the actor model (Figure 4.2), *an actor is the foundation of concurrency* or rather like an agent which does the actual work. It is analogous to a process or thread. Actors are very different from objects because in an object-oriented programming paradigm, an object can interact directly with another object, i.e. changing its values or invoking a method. This causes synchronisation issues in multi-threaded programs and additional synchronisation locks are necessary to ensure proper functioning of the program [78]. In contrast to this, the actor model provides no direct way for an actor to invoke or interact with another actor. Actors respond to messages. In response to a message, an actor may change its internal state, perform some computation, fork new actors or send messages to other actors. This makes it a unit of static encapsulation as well as concurrency [67].

Message passing between actors happens asynchronously. Every actor has a mailbox where the received messages are queued. An actor processes a single message from the mailbox at any given time, i.e. synchronously. During the processing of a message, other messages may queue up in the mailbox. A collection of actors, together with their mailboxes and configuration parameters, is often termed an *actor system*.

The main intuition is that *when a user designs a flow, the flow is modelled internally in terms of actors*, i.e. an actor is a basic execution unit of the mashup tool. For instance, the flow depicted in Figure 4.3 corresponds to three actors namely A, B and C with the computation starting with actor A. On completion, it sends a message to actor B and so on.

In the realisation of aFlux, Akka [7], a popular library for building actor systems in Java and Scala, has been used. Since Akka can be configured in many different ways for parallel and distributed operations and governs how the actors would be spawned and executed. This shields the actors from worrying about synchronisation issues.

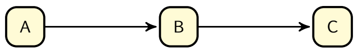


Figure 4.3: A typical mashup flow

4.1.2 Execution of a Flow

In aFlux, a user can create a mashup flow called *flux*. A flux is analogous to a flow in IBM Node-RED. The only requirement for designing a flux is that it should have a start node and an end node. A flux by default is tied down to a logical unit called *job*. Every job can have one or more fluxes. When a job containing a flux like in Figure 4.3 is designed in aFlux, the control flows through a number of parties before final execution. Firstly, the job must be saved which allows the mashup tool to parse the flux diagram created on the front-end by the user. The parsing involves creating and saving a graph model for the job—the *Flux Execution Model*. The parser does not care how many fluxes are present in the job because it scans for special nodes in it. These special nodes are *start nodes*, i.e. *specialised actors which can be triggered without receiving any message.* Other nodes are normal actors which react to messages. On detection of all start nodes in an activity, the graph model is built by simply traversing the connection links between the components as designed by the user on the front-end. A flux execution model of a job contains as many graphs as the number of fluxes present in it.

On deployment, the control flows from the front-end to the controller responsible for starting the actual execution of the job. This involves invoking the runner which fetches the flux execution model of the job. For every flux in the job, the runner environment proceeds to:

1. identify the relevant actors present in the graph.
2. instantiate an actor system with the actors identified in step 1.
3. trigger the start nodes by sending a signal.

After this, *the execution follows the edges of the graph model*, i.e. the start actors upon completion send messages to the next actors in the graph, which execute and send messages to the next actors and so on.

4.1.3 Logical Structuring Units

To abstract away independent logic within a main application flow, the system supports logical structuring units called *sub-flows*. A sub-flow encompasses a complete business logic and is independent from other parts of the mashup. A good candidate for a sub-flow is for example a reusable data analytics logic which involves specifying how the data should be loaded and processed and what results should be extracted. They encompass within themselves a complete flow of graphical components.

4.1.4 Concurrent Execution of Components

Every *component* in aFlux has a special *concurrency parameter* attached to it which can be configured by the user while designing a flux. The idea is that in an actor system, every actor processes one message at a time. During its processing, new messages are queued on their arrival. To avoid this and facilitate faster processing, every component in aFlux can be made to execute concurrently by specifying the upper threshold value of concurrency. If a component has concurrency level of n, messages arrive quickly and the component takes quite some time to process a message, then the actor system can spawn multiple instances of that component to process the messages concurrently up-to n or up-to the global threshold value defined in the system, whichever is minimum. Beyond that the messages are queued as usual and processed whenever any instance finishes its current execution. This specification of concurrency parameter is applicable to individual components as well as sub-flows in aFlux and is decided by the user creating a flux. In the case of sub-flows, basically all the components used within it adhere to the concurrency limit of the sub-flow which means that the actor system can spawn multiple instances of every component used inside the sub-flow as the need arises during run-time.

4.2 Working of the Prototype

4.2.1 Component: Essential Constituents

A component is the foundational unit of a flow designed in a mashup tool. In aFlux, every *'component'* on the front-end which the user can use as a mashup component to create a *flux* is internally an actor. Every *aFlux actor* has an internal *business logic* and *a set of properties* for customisation of its behaviour.

Business logic The *business logic* is self-contained, i.e. it can execute to completion if invoked by passing its required data via a message. In fact, these actors react only to messages. The *business logic* can be broadly classified into three distinct parts as discussed below.

Type checking of messages When an actor starts its execution, the first step it does is to ensure that the message received in its mailbox can indeed be processed. This is done by checking the type of the received message and ensuring that it is compatible. An actor can support multiple data-types and this is left to the developer on how to resolve or may do relevant type conversions if required. The man idea is that, if the message received is of compatible type then, the actor proceeds to execution of the actual computational logic. If it is incompatible then the message is ignored and the output for the message is decided by the developer, i.e. whether not to produce any output or send an error message.

Computational logic This part houses the actual computational logic of the actor, i.e. what it does and how it does. Examples of it can be to read from a database, fetch data from a specific REST API etc.

Passing of output After completion of execution and production of results, every actor needs to do something with it, i.e. either pass it through its output port(s) such that the message is received in another connected actor's mailbox or pass the output to be printed on the front-end console so that user can know about the result. If an actor has multiple output ports then the developer of the actor can decide whether to send same output through all the ports or different set of outputs through different ports.

Properties Every actor has two kinds of properties: (i) *user-configurable properties* and (ii) *non-configurable properties*. The properties of components play a vital role in supporting user customisation as well as deciding the execution sequence.

non-configurable properties The *'non-configurable'* properties typically govern how the actor behaves within the actor system. Example of an 'non-configurable' property would be the number of input and output ports the actor accepts which the developer must define while creating a new actor for aFlux. This is a property which the user on the front-end cannot alter while creating a flux using its visual representation, i.e. 'component'. Another example would be the *method of invocation*.

An actor can start its execution as soon as it receives a message in its mailbox then a question arises how the actor used as the very first component in a flux will ever start its execution? To solve this dilemma, aFlux supports a property called *'method of invocation'*. *'method of invocation'* is of two types, i.e. *'triggered by system'* and *'triggered by data'*. If an actor has its *'method of invocation'* property set to *'triggered by system'* then it can be used as a first mashup component in a flux because it does not require any arrival of message in its mailbox to trigger its execution. Such actors are triggered by the system for invocation when a flux is executed which triggers a series of chain reaction since the first actor on its completion sends a message to the next connected actor and the execution continues the path of the flux designed by the user. On the other hand, if an actor's *'method of invocation'* property is set to *'triggered by data'*, then it cannot be used as the first component in a flux because it requires arrival of a message in its mailbox before it can start its execution. Putting things in a different way, those actors have their *'method of invocation'* set to *'triggered by system'* do not support any input port as they don't depend on any input from preceding component or rather they don't have any. Finally, every actor has a *unique name* assigned to it in the system for identification purposes.

user-configurable properties The *'user-configurable'* properties are those which can be presented on the front-end to the user to configure and these are typically passed as parameters to the business logic to alter its execution as well as its output. For instance, if we consider that an actor has been developed to read datasets from a MongoDB then examples of *'user-configurable'* properties may include IP address of the database server, port number, which database to connect to and which collection to fetch datasets from etc. One interesting *'user-configurable'* property is the *'concurrency'* property. This basically specifies how many instances of the actor can be spawned by the system in case this actor receives multiple messages in one unit of time. By default, the *'concurrency'* of every actor is set to one which means that only one instance of the actor is running which processes one message at a time. However, if set to more than one and the actor has a huge influx of messages in its mailbox then the system can spawn multiple instances of the actor to process the messages concurrently. It is interesting to note that an actor may not support any *'user-configurable'* properties if such is the design decision of the developer. Most of the *'user-configurable'* properties typically have some default values assigned to them by the developer of the actor. Again, the *'user-configurable'* properties can be further classified into *'essential user-configurable'* properties and *'non-essential user-configurable'* properties. The *'essential user-configurable'* properties must be configured by the user on the front-end as the correct execution of the actor depends on values of these properties while *'non-essential user-configurable'* properties are totally optional and do not influence the actual execution of the actor. An example of *'essential user-configurable'* property would be the location of the database server for a MongoDB actor while an example of *'non-essential user-configurable'* property for the same actor would be a name to be displayed on the front-end for the convenience of the user.

An interesting point to note is that if an actor is used as the first component in a flux and is a normal one then, it typically completes its execution, passes its output and exits from the actor system. On the other hand, if it is the first actor but it is also a streaming actor (discussed in Section 4.3) then, it continues its execution indefinitely as it needs to stream data continuously to the next set of connected actors.

Figure 4.1 shows the front-end of aFlux with which the user interacts to create a flux. All the components available in the system are listed on the left-hand panel from where they are dragged to the canvas and connected to describe the control-flow of the flux. For every component dragged on to the canvas and selected, its *'user-configurable'* properties are displayed on the right-hand side panel where the user can configure them. When the user connects two components, a state change is observed on the front-end and the system captures the flux present in the canvas, i.e. the entire set of components connected together with their *'user-configurable'* properties. This state is stored on the front-end and continuously updated whenever the user changes anything on the canvas, i.e. on observation of a state change. When the user decides to save the flux or execute it, a series of interaction occurs between the user interface and mashup engine.

The first interaction between the mashup engine and user interface occurs when the user decides to save a newly created flux. The state stored on the UI is passed to the mashup engine and stored in the form of a DAG. This DAG is also called as the *'flux execution model'*. In the case where the user wants to load a saved flux and display it on the UI, the mashup engine fetches its *'flux execution model'* and passes it. A state is populated from this information and the flux diagram on the canvas is displayed automatically with the relevant actors already dragged, connected and their properties modified accordingly.

4.2.2 Component Execution

The diagram depicted in Figure 4.4 is an *Identify, Down, Aid, and Role* (IDAR) *graph* [124] which summarises the execution of a flux consisting of components, i.e. actors within aFlux. IDAR graphs offer a more readable and comprehensible way of representing how system components communicate and interact in comparison to Unified Modelling Language (UML) [124]. In an IDAR graph, objects typically communicate either by sending a *command message* (control messages) or a *non-command message*, which is called a notice. The controlling objects always remain at a higher level in the hierarchy compared to the objects being controlled. An arrow with a bubble (circle) on its tail stands for an indirect method call while a dotted arrow indicates data-flow. Other subsystems having their own hierarchy are represented with hexagons denoting the subsystem manager.

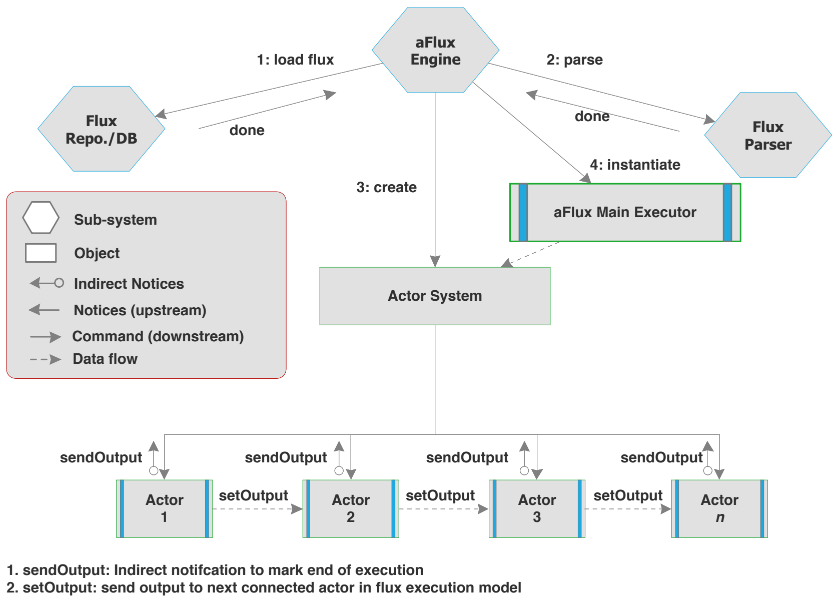


Figure 4.4: Flow execution in aFlux: IDAR representation

The execution of a flux typically follows the following sequence:

1. When the user executes a flux, the main component in the back-end called as *'aFlux Engine'* sends a command to the *'Flux Repository'* subsystem which reads the stored *'flux execution model'* and returns it.

2. *'aFlux Engine'* sends a command to the *'Parser'* subsystem for parsing. The *'flux execution model'* is checked for consistency, i.e. if the first component in the flux contains an actor whose *'method of invocation'* property has been defined as *'triggered by system'*. The relevant actors used in the flux are identified and after completion of this operation the *'aFlux Engine'* is notified.

3. The *'aFlux Engine'* creates an actor system where the actors used in a flux would be executed.

4. The *'aFlux Engine'* instantiates *'aFlux Main Executor'* by passing the set of actors to be executed.

5. This data flows from *'aFlux Main Executor'* to *'Actor System'* where the relevant actors are instantiated and the first actor is triggered by the *'Actor System'*.

6. The first actor completes its execution, notifies to the *'Actor System'* about its completion of execution by sending a notification via an indirect method call and at the same time sends its output to the next connected actor.

7. This process is repeated till the last actor. When it notifies the *'Actor System'* about its completion of execution, then the *'Actor System'* removes all inactive actors and frees up memory.

4.3 Stream Processing with Flow-based Programming

The flow based structure of mashup tools, i.e. passage of control to the succeeding component after completion of execution of the current component is very different from the requirements of stream processing where the component fetching real-time data (aka the listener component) cannot finish its execution. It must listen continuously to the arrival of new datasets and pass them to the succeeding component for analysis. Also, the listener component has many behavioural configurations which decide when and how to send datasets to the succeeding component for analysis.

In aFlux, *the actor model has been extended to support components which need to process streaming data.* The implementation of streaming components relies on the Akka streams library, an extension of the Akka library. Applications based on Akka streams are formulated as building blocks of three types: *source*, *sink*, and *flow*. The source is the starting point of the stream. Each source has a single output port and no input port typically. Data is fetched by the source using the configuration parameters specified and it comes out from its output and continues to the next component that is connected to the source. The sink is basically the opposite of the source. It is the endpoint of a stream and therefore consumes data. Basically, it is a subscriber of the data sent or processed by a source. The third component, the flow, acts as a connector between different streams and is used to process and transform the streaming data. The flow has both inputs and outputs. A flow can be connected to a source, the outcome of which results in a new source or even after a sink which creates a new sink. A flow connected to both a source and a sink results in a runnable flux (Figure 4.5), which is the blueprint of a stream.

Each streaming component in aFlux offers a different stream analytics functionality (e.g. filter, merge) and can be connected to other stream analytics components or to any common aFlux component. The stream analytics capabilities make use of three categories of components, i.e. *fan-in*, *fan-out* and *processing* components. *Fan-in* operations allow joining multiple streams into a single output stream. They accept two or more inputs and

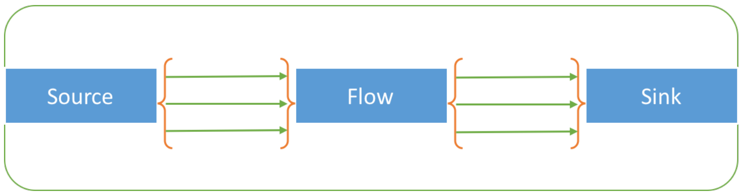


Figure 4.5: Runnable flux with streaming actors

give one output. *Fan-out* operations allow splitting the stream into sub-streams. They accept one stream and can give multiple outputs. *Processing* operations accept one stream as an input and transform it accordingly. They then output the modified stream which may be processed further by another processing component. The transformation of the stream is done in real-time, i.e. when the stream is available on the system for processing and not when it is generated at source. Every component is internally composed by a source, a flow and a sink. When a component is executed by aFlux, a blueprint that describes its processing steps is generated. The blueprints are only defined once, the very first time the component is called, e.g. create a queue where the new incoming elements of the stream get appended for a component to process.

Every stream analytics component has some attributes that can be adjusted by the user at run-time. For example, for the processing components the user can optionally define windowing properties such as window type and window size. The internal source of every stream analytics component has a *queue* (buffer), the size of which can be defined by the user (default is 1000 messages). The queue is used to temporarily store the messages (elements) that the components receives from its previous component in the aFlux flow while they are waiting to get processed. Along with the queue size, the user may also define an *overflow strategy* that is applied when the queue size exceeds the specified limit. Figure 4.6 shows the interface of aFlux where the user can define buffer size and overflow strategy. The overflow strategy determines what happens if the buffer is full and a new element arrives. It can be configured as:

drop buffer drops all buffered elements to make space for the new element.

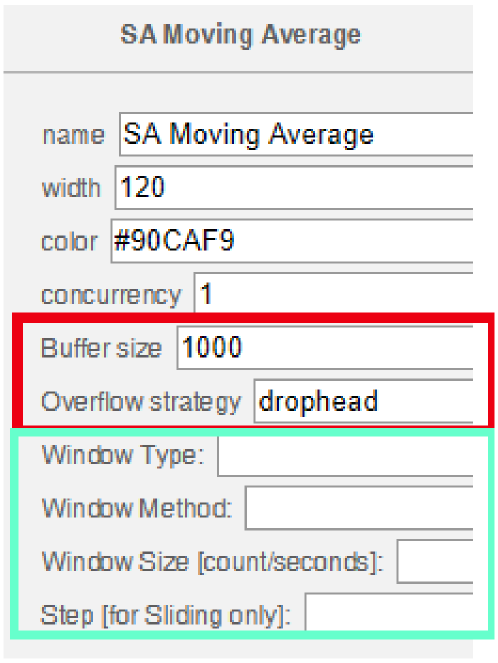
drop head drops the oldest element from the buffer.

drop tail drops the newest element from the buffer.

drop new drops the new incoming element.

The internal flow part of a streaming component describes its logic and defines its behaviour. This is where the whole processing of messages takes place. The source sends the messages directly to the flow when it receives them. As soon as the processing of a message has finished, the result is then passed to the sink. By default, the analysis of messages is done in real-time and each message is processed one-by-one (e.g. count how many cars have crossed a given junction). However, the user can also select windowing options.

Figure 4.6 shows the interface of aFlux where the user can define windowing properties. The implementation supports *content-based* and *time-based* windows. For both of these


SA Moving Average

name SA Moving Average
width 120
color #90CAF9
concurrency 1
Buffer size 1000
Overflow strategy drophead
Window Type:
Window Method:
Window Size [count/seconds]:
Step [for Sliding only]:

Figure 4.6: aFlux GUI to specify buffer size, overflow strategy & window parameters

types of windows, the user can also specify a *windowing method* (tumbling or sliding) and also define a *window size* (in elements or seconds) and a sliding step (in elements or seconds).

In a nutshell, a window is created as soon as the first element that should belong to this window arrives, and the window closes when the time or its content surpasses the limit defined by the user. A window gathers all messages that arrive from the source until it is closed completely. Finally, the component applies the required processing on the data in the window and passes the result(s) to the sink. The first thing is to choose whether the window should be content or time-based. A content-based window has a fixed size of a number of elements n. It collects elements in a window and evaluates the window when the n^{th} element has been added. On the other hand, a time-based window groups elements in a window based on time. The size of a time window is defined in seconds. For example, a time window of size 5 seconds will collect all elements that will arrive in 5 seconds from its opening and will apply a function to them after 5 seconds have passed.

In stream analytics, there are different notions of time like:

processing time windows are defined based on the wall clock of the machine on which the window is being processed.

event time windows are defined with respect to timestamps that are attached to each element.

hybrid time combines processing and event time.

In the implementation, currently processing time is used to interpret time in our processor.

For instance, a time window of size 60 seconds, will close exactly after 60 seconds. After deciding on using content or time windows, the user has to decide how to divide the continuous elements into discrete chunks. Here the user has the following two options. The first is tumbling window, where stream elements are divided into non overlapping parts and each element can only belong to a single window. The second option is sliding window, which is parametrised by length and step. These windows overlap and each element may belong to multiple windows. Windows can be either tumbling or sliding. A tumbling window tumbles over the stream of data. This type of window is non overlapping, which means that the elements in a window will not appear in other windows. A tumbling window can be either content-based (e.g. "Calculate the average speed of every 100 cars") or window-based (e.g. "Find the count of tweets per time zone every 10 seconds") whereas a sliding window slides over the stream of data. Due to this reason, a sliding window can be overlapping and it gives a smoother aggregation over the incoming stream since it does not jump from one input set to the other but it slides over the incoming data. A sliding window has an additional parameter which describes the size of the hop. A sliding window can as well be either content-based (e.g. "For every 10 cars calculate the average speed of the last 100 cars") or time-based (e.g. "Every 5 seconds find the count of tweets per time zone in the last 10 seconds"). Thus if the sliding step is smaller than the window size, elements might be assigned to multiple successive windows. The tumbling window can be conceived as a special case of a sliding window, where the window size is equal to the sliding step. Therefore, it does not make any sense to define a sliding step for a tumbling window.

The sink is the final stage of a stream analytics component. The sink gets the results from the flow and decides the final outcome. In this case, the results need to be send to the next component in the flux because the components should be able to pass messages to each other.

4.4 Example Flow for Stream Analytics

An open-source traffic simulation software by the name SUMO [101] has been used to demonstrate the stream processing capabilities of aFlux. The data generated from the system is random making it a perfect fit for real-time analytics and the results of the analytics affect the system performance, i.e. traffic congestion in SUMO. For the evaluation purposes, the traffic of A9 highway[4] near Munich has been used for simulation. TraCI [160], a python based interface has been used for data-exchange between SUMO and Kafka [117, 75, 69].

Scenario In the scenario, all cars run on a straight line on the A9 highway and in the same direction from south to north. At a certain point on the highway, there are four lanes, three of which possess a loop detector (see Figure 4.7). Loop detectors measure the occupancy rate (0-100) on the lane, i.e. how long was a car placed on the loop detector during the last tick (one tick equals to one second of simulation time in SUMO). A high occupancy rate signals a more busy lane and therefore the possibility of a traffic conges-

[4]A9 public github project, available at `https://github.com/iliasger/Traffic-Simulation-A9`

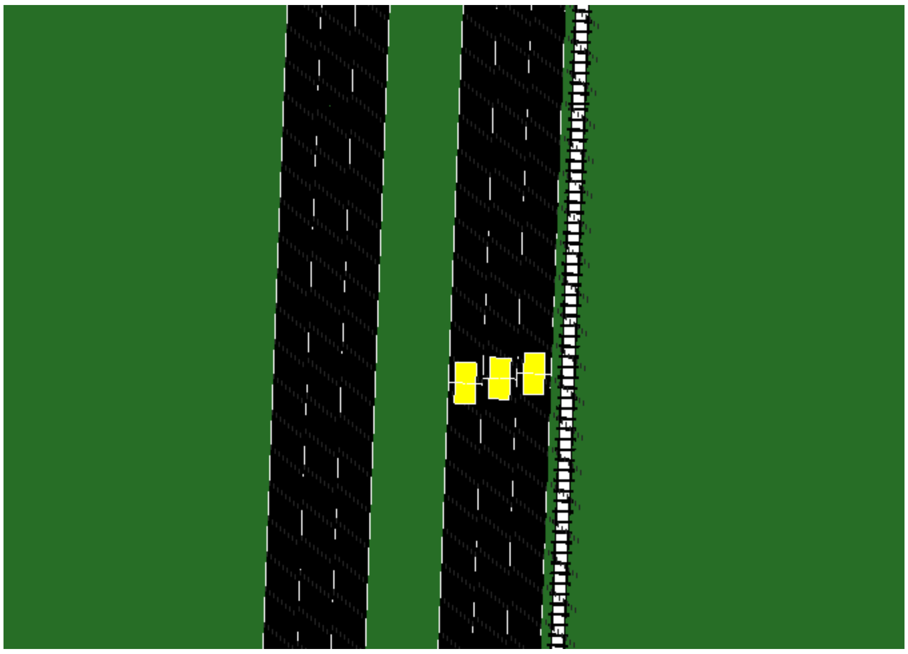


Figure 4.7: The 4 lanes used in the experiment. Each of the 3 left lanes possess a loop detector. The 4^{th} lane is initially closed

tion. The fourth lane of the highway, further referred as shoulder-lane, is initially closed which means that no cars can run on it. However, if the total average of the occupancy rates of the three other lanes exceeds the threshold of 30, the shoulder-lane opens to reduce the traffic. When the average of the occupancy rates falls below 30, the shoulder-lane closes again. On the 500^{th} tick of the simulation, it is assumed that a car accident happens and a lane, ahead of the four previously mentioned lanes, gets closed at the same moment and remains closed for the rest of the experiment. This builds up a congestion on the highway, causing the occupancy rates of the loop detectors to increase and makes it meaningful to open the shoulder-lane at some point to alleviate the congestion.

Goal The goal of this experiment is to compare different stream processing methods on data coming from the simulation environment. Particularly, the one-by-one method for processing data, tumbling window processing with three different window sizes (50, 300 and 500) and sliding window processing are compared. The user can define the method of data processing and change various associated parameters on aFlux UI. The loop detector occupancy, lane state, mean speed of cars and time values from SUMO are captured via TraCI and published to Kafka. The lane state is a binary value that indicates the state of the shoulder-lane at the current tick(0 means closed and vice-versa). Mean speeds are used as an indicator of a traffic congestion, i.e. a low mean speed on a lane indicates a traffic congestion. The average mean speeds of the three lanes are plotted to demonstrate the effect of the shoulder-lane in the relief of a traffic congestion. Finally, time measures the duration the shoulder-lane state takes to reach 1 for the first time and the duration it needs to reach 0 again for the last time. This factor indicates the responsiveness of each method on traffic changes, e.g. how fast the system perceives and reacts to a traffic congestion. All runs of the experiment are based on the exact same conditions. The routes of the cars and the way that they are simulated in the simulation have the same

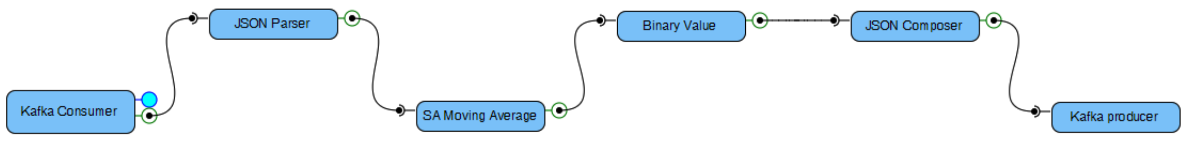


Figure 4.8: aFlux flow used in the experiment - subscribes to a Kafka topic that publishes the occupancy rates of loop detectors and calculates their moving average in real-time

randomness for all runs and therefore do not impact the experiment results. The only factor that influences the results of the experiments are the decisions to open and close the shoulder-lane.

Flow-based data analytics For the reliability of the results, the experiment have been run twice for every stream processing method. In order to make decisions to change the state of the shoulder-lane based on the occupancy of the loop detectors, a flow in aFlux has been designed (Figure 4.8). The first component of the flow is a Kafka subscriber that listens to the topic where TraCI publishes the occupancy rates of each loop detector on every tick. The data is parsed using a JSON parser and the results are passed to the moving average component. The moving average component receives the occupancy values and calculates their average on real-time and based on the user-specified method (windowing or simple processing). The results are then passed to the binary value component which outputs 0 if the average is below the user-defined threshold (e.g. 30) or 1 otherwise. Finally, the result is transformed into a JSON file and published to a Kafka topic, where TraCI listens, to decide whether to open or close the shoulder-lane.

Evaluation parameters The data coming from SUMO is analysed in a number of stream processing methods via a number of configurable parameters. The result of such analytics affect the performance of the system, i.e. SUMO. To measure the affect on system performance, the following aspects are considered:

Responsiveness indicates how fast the system can detect a traffic congestion and open the shoulder-lane to alleviate it.

Settling time refers to the time the system needs to reach a steady state [72]. In the experiment, the shoulder-lane may open and close successively. We define the settling time as the time the shoulder-lane needs to reach a steady state after a change occurs. It is estimated based on the shoulder-lane state parameter.

Stability refers to the ability of the system to reach a stable state without overshoots when a change occurs. An overshoot occurs when the system exceeds a certain target point before convergence [72]. In our case, stability is tested when the shoulder-lane changes state. Stability is in inverse proportion to settling time, i.e. short settling time infers to a higher system stability.

The occupancy rates of the loop detectors have not been considered for result-analysis

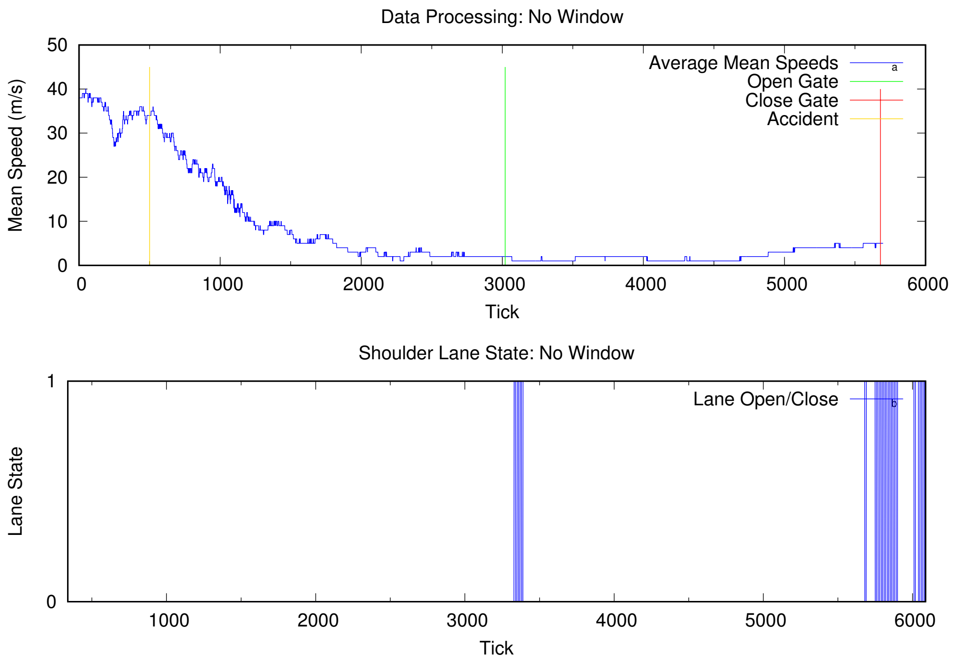


Figure 4.9: Average mean speed of cars moving on the 3 lanes. (a) Data analysis without windows (one-by-one), (b) Shoulder-lane state without windows (one-by-one)

since they are used to make decisions in aFlux and the focus is to examine the impact of these decisions to other factors in a traffic system.

Analysis of Mean Speed First, for every processing method the average of the mean speeds of the cars moving on the 3 previously mentioned lanes, per tick is analysed. The mean speed of the vehicles running on a lane at a certain point of time discloses information about the current congestion of this lane. Through this analysis, the responsiveness of each method to changes and their effectiveness to solve a problem is determined; in this case to alleviate the traffic congestion.

From Figure 4.9 (a), when the accident happens at tick 500 the average mean speeds of cars moving on the 3 particular lanes that we examine, falls significantly. This means that a congestion starts to build-up on these lanes. The loop detectors send their occupancy rates to aFlux every tick and they are getting averaged by the moving average component one-by-one. Since we do not use any window to process the incoming data, each average occupancy value depends on all previous occupancy values, even on the low occupancy rates before the accident. As a result, the moving average value cannot reflect new environment changes fast enough and hence it reaches the threshold of 30 on the 3020^{th} tick for the first time to open the shoulder-lane. By observing the Figure 4.9 (a) one can see an up-trend of the average speeds on the 4600^{th} tick but the moving average value falls below 30 only on the 5680^{th} tick for the last time when the shoulder-lane gets closed as well, a fact that shows a slow reaction time.

Figure 4.10 (a) shows the average mean speeds of the lanes when using a content-based tumbling window of size 50 to process the occupancy rates of the loop detectors. Using a window of size 50 means that only the 50 latest occupancy values are aggregated and

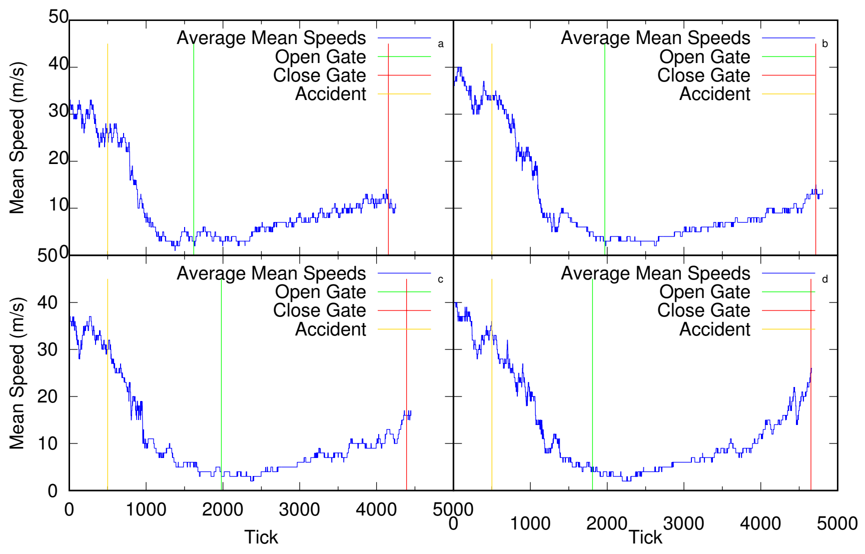


Figure 4.10: Data analysis with content-based tumbling window of size (a) 50, (b) 300, (c) 500 and (d) content-based sliding window of size 500 and step 250

averaged and that the average does not keep the state of the previous values. We consider 50 to be a small window size, as it lasts for about 17 ticks. The difference in reaction time to a non-window processing is significant since the system perceives much earlier the traffic congestion and opens the shoulder-lane on tick 1620 for the first time. When the traffic congestion is alleviated, the system closes the shoulder-lane on the 4145th tick which is also a much faster reaction in comparison to 5680 ticks that it took for the non-window processing.

Figure 4.10 (b) depicts the average mean speeds of cars when using a content-based tumbling window of size 300. Using this method, the system opens the shoulder-lane on tick 1970 and closes the shoulder-lane on the 4715th tick for the last time. This processing method responds to changes faster than the no-window processing but a bit slower than the tumbling window of size 50. This performance is expected since a larger window size takes longer to aggregate more values (100 ticks) and hence adapts slower to changes in comparison to smaller window sizes. In Figure 4.10 (c), we present the results of the tumbling window with size 500 (167 ticks). In comparison to window size 300 this processing method is slightly slower (shoulder lane opens at tick 1979 and closes at 4391). The distribution of the mean speeds is quite similar to window size 300 though which suggests that the system shows a similar behaviour in both cases. Figure 4.10 (d) shows a sliding window of size 500 with a sliding step of 250. The difference of the previous window processing is that the sliding window takes into consideration the previous state as well by overlapping on previous values. In our case, the sliding window overlaps the 250 latest elements of the previous window. In general a sliding window gives smoother and in some cases faster results, since it is moving faster (emits more values than a tumbling window). By comparing the figures of the sliding and tumbling window of size 500, one can observe that the distribution of the average of the mean speeds are quite similar in both graphs. The sliding window seems to be faster in opening the shoulder lane for the first time (tick 1810) but it is a bit slower in closing it (tick 4652). We consider this differentiation to be dependent on the variation of data in each experiment.

Analysis of Shoulder-lane state The shoulder-lane state depends completely on the average of the occupancy rates of the lanes. If the average occupancy rate is above 30 the shoulder-lane state turns to 1 (lane opens) otherwise it is 0. The analysis of the state of the shoulder lane shows the variability of each method. When the occupancy rate reaches 30 it may climb above 30 (overshoot) and then it may fall below 30 (undershoot) again on the next tick. In control theory, overshoot refers to an output that exceeds its target value, whereas the phenomenon where the output is lower than the target value is called undershoot. In our case, it is normal to have an overshoot as we expect the occupancy rates to rise above 30 but here we want to examine the overshoot followed by an undershoot ratio which leads to an unstable state where the shoulder-lane opens and gets closed on successive ticks. We also focus on the settling times of each method. A stable system must have short settling times [2], i.e. converge quickly to its steady value, and must not overshoot.

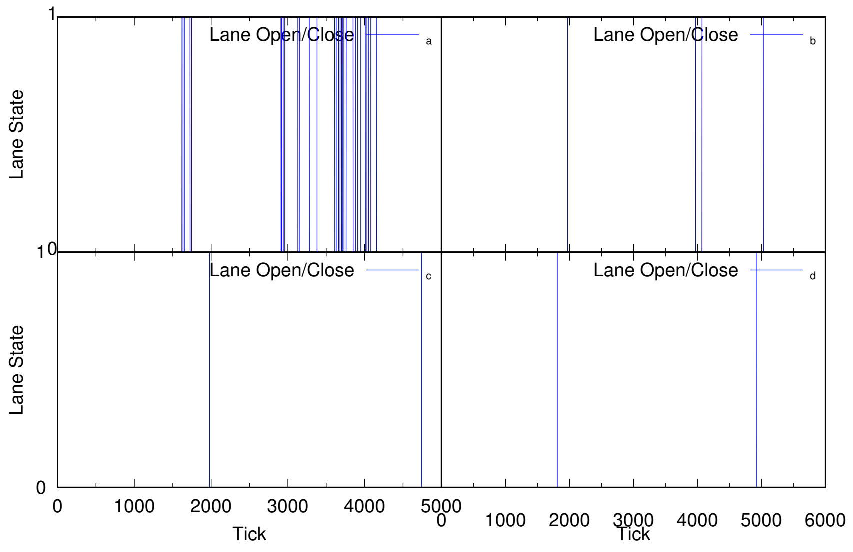


Figure 4.11: Shoulder-lane state. 0 means lane closed. 1 means lane open. Data analysis with content-based tumbling window of size (a) 50, (b) 300, (c) 500 and (d) content-based sliding window of size 500 and step 250

Figure 4.9 (b) shows the variation of the state of the shoulder lane when occupancy rates are processed one-by-one. On tick 3020 the lane opens for the first time and we observe an overshoot-undershoot case which lasts for 3 ticks before the lane state value settles on 1. Thus, the settling time when the shoulder-lane opens for the first time is 3 ticks. When the traffic is about to be alleviated and just before the shoulder-lane closes for the last time on tick 5680, we see another overshoot-undershoot incident with a longer settling time. Concerning the variation of the shoulder-lane state, Figure 4.11(a) shows that this particular method has many overshoot and undershoot incidents causing the shoulder-lane to open and close many times successively. This fact implies an unbalanced system with long settling times. We can attribute this lack of stability to the small window size which is sensitive to the behaviour of a small sample of data. Figure 4.11(b) depicts the shoulder-lane states during the experiment. In this method, there is no big variation between the states and almost no overshoot-undershoot incidents. The settling time is short and the system seems to be balanced. The big window size allows the system to make a decision, based on a bigger sample of data and hence it is more stable than the two previously mentioned methods. In Figure 4.11 (c), the results of the tumbling window with size 500 can be seen. This window size is considered very big and it is used as an extreme case here. As expected there are no overshoot-undershoot incidents and the system seems to be very balanced. The shoulder-lane opens and closes only once when needed and there is no settling time. This is the best window size compared to the previous one. In the following section, we will examine the same window size for the sliding version of the tumbling window. Figure 4.11 (d) shows the results of the sliding window with size 500 and sliding step 250. As expected, there are no overshoot-undershoot incidents here as well and the system is balanced. The shoulder-lane opens and closes only once when needed and there is no settling time. This figure is quite similar to the corresponding one of the tumbling window which implies that there no big difference between a tumbling and a sliding window in this case concerning the stability of the system.

Method	Responsiveness	Settling Time	Stability
No window	very slow	long	low
Tumbling window 50	very fast	very long	very low
Tumbling window 300	slow	very short	high
Tumbling window 500	slow	none	very high
Sliding window 500	slow	none	very high

Table 4.1: Stream analytics method characteristics

Discussion The results of various stream analytics methods based on their performance in solving a traffic control problem in real-time are summarised in Table 4.1. By observing the table, two points become evident: (i) stream processing can be done in various ways, (ii) these methods perform differently thereby affecting the final outcome and performance of the application. For instance, in the above traffic use-case a small window processing method, like the tumbling window of size 50, showed a very good responsiveness as it was the fastest method to open and close the shoulder-lane when a traffic congestion occurred but it showed poor stability since its settling times were the longest of all five methods. On the other hand, the no-window processing method is the slowest method to perceive and respond to a change in the environment (e.g. traffic congestion). This method also has low stability since it is prone to long settling times and overshoot-undershoot incidents. From Table 4.1, it can be stated that the most efficient method to control traffic in our scenario is a tumbling window of a normal size (not too big or small), but it will take further iterations to define the ideal window size.

What has been evaluated? When data needs to be processed in real-time and the result of such analysis impacts the final outcome, i.e. performance of the application,

Tools	Concurrent execution	Data analytics app. development	Flows with built-in stream processing	Execution of each component in separate threads	Scaling up of individual components	Parametrisation of component buffer	Streaming paradigm
aFlux	Yes	Batch jobs. Streaming is a special case	Yes	Yes	Yes	Yes	Actor model with support for CEP and relational streaming paradigm
StreamIt	Yes	Streaming only. Batch is a special case	No	Unknown	No	No	Synchronous dataflow paradigm
IBM Infosphere Streams	Yes	Streaming only. Batch is a special case	No	Yes	Unknown	No	Relational streaming paradigm with support for CEP
IBM SPSS	Yes	Streaming only. Batch is a special case	No	Unknown	No	No	Appears to follow relational streaming paradigm
QM-IConf	No	Streaming only. Batch is a special case	No	No	No	No	Relational streaming paradigm
Node-RED	No	No support for streaming	No	No	No	No	Not Applicable

Table 4.2: Comparison of aFlux with existing solutions

there is no easy way to know the right stream processing method with the correct parameters. Hence, it becomes very tedious to manually write the relevant code and recompile every time a user wants to try something new. By parametrising the controlling aspects of stream processing it becomes easy for non-experts to test various stream processing methods to suit their application needs. Overall, the following aspects of aFlux has been captured via the example — the integrated stream processing capabilities in a flow, parametrisation of the buffer capacity and overflow strategies and modelling of different kinds of window methods to process data, i.e. tumbling and sliding windows.

4.5 Discussion

In this section, we compare our concepts (as in aFlux) with exiting solutions supporting flow-based stream analytics. In Section 2.12, we have classified existing solutions into two categories. The category 1 solutions deals with tools supporting flow-based data analytics especially stream processing. The tools are StreamIt, IBM Streams Processing Language or the IBM Infosphere Streams, QM-IConf and IBM SPSS modeller. We also consider Node-RED for comparison as it is one of the prominent platforms used for flow-based programming in the context of IoT and is widely supported by IBM.

First, we begin by defining the scope and the parameters to compare. Figure 4.12 shows the model of a streaming application. This model is also known as dataflow model. Every stream application in high-level programming tools are modelled as a dataflow graph where vertices represent operators and edges represent valid dataflow pathways. Operators can either be data sources, data sinks or data transformers. The edge between two operators has a channel capacity Accordingly, we compare the following key parameters between the solutions (Table 4.2 summarises the compared parameters):

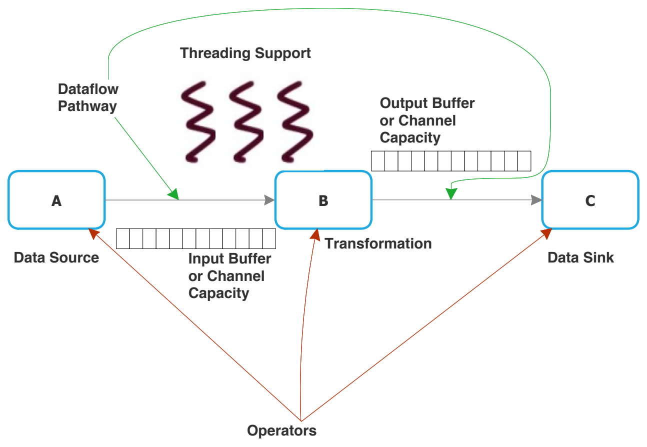


Figure 4.12: Streaming application modelled as a data-flow graph with vertices representing operators and edges representing data-flow pathways

Concurent execution of components aFlux is based on actor-model hence it supports concurrent execution semantics, i.e. one operator in the dataflow graph can start its execution in parallel before the finish of its predecessor. All tools with exception of Node-RED and QM-IConf support asynchronous execution of data operators in the dataflow graph. *The second column in Table 4.2 lists this criterion.*

Data analytics application development A second criterion for comparison is the kind of data analytics application that can be developed using these tools. *The third and fourth column in Table 4.2 list this criterion.* aFlux relies on asynchronous message passing techniques to model the dataflow graph and extends the actor model to support continuous streaming of datasets. This allows a user to model batch analytics application, a completely streaming application or a batch application with some components doing stream analytics. Streaming is considered as a special case in aFlux while other platforms treat modelling of batch jobs as a special case. Other solutions are stream only platforms and are not designed to specify batch analytic jobs though they can be modelled as both stream and batch analytics rely on the dataflow paradigm [144].

Component execution and scaling One of the important criterion for comparison is to see if the individual components used in an application flow are executed in separate threads and can scale. *The fifth and sixth column in Table 4.2 list this criterion.* Since operators, i.e. components in aFlux, are independent from other components it supports scaling up of instances of a specific component. Additionally, each actor instance of the same component is a different unit of computation hence boots performance. Other solutions do not support scaling up of instance of a component if the data load increases. Nevertheless, IBM Infosphere Streams specifies separate thread of execution for each data operators in a dataflow graph.

Parametrisation of Buffers and overflow strategy The streams of data travelling from one operator to another are stored in a buffer queue before being processed. Customisation of this buffer is an important parameter for comparison. *The seventh column in Table 4.2 lists this criterion.* aFlux supports parametrisation of this buffer and specify over-flow strategies. This concept is non-existent in all existing solution and from the running example in Section 4.4 it is clear that optimisation of this aspect affects the performance of the streaming application.

Stream processing paradigm *The eighth column in Table 4.2 lists the criterion of streaming paradigm the tool is built upon.* There are three major stream processing paradigms: (i) synchronous dataflow where the data ingestion and data production rates are predefined which makes it unrealistic in real-world use cases, (ii) relational paradigm which relies on the concept of relational databases to process streams which simplifies stream processing and (iii) complex event processing paradigm in which operators detect patterns in input streams and infer outputs. Platforms supporting relational are easier to model stream applications, scale-well while the complex event processing paradigm permits to model complex streaming applications. aFlux supports the relational and the complex event processing paradigm while StreamIt supports only synchronous dataflow paradigm.

5 Graphical Flow-based Spark Programming

"The function of good software is to make the complex appear to be simple."

— Grady Booch

We have a large number of tools that are known as Big Data analytics tools. These tools are especially used for applications like targeted advertising and social network analysis. [110]. Spark is one such prominent tool, allowing for advanced, scalable, fault-tolerant analytics and comes equipped with machine learning libraries as well as stream processing capabilities [170, 168]. Nevertheless, the learning curve associated with Spark is quite steep [170]. In response to this, this chapter focuses on supporting Spark programming via graphical flow-based programming paradigm.

To illustrate the importance of efficiently prototyping data analytics pipelines, consider a scenario where the administrator of a taxi service wishes to know the current high demand areas in the city, to redirect the available fleet accordingly and to reduce customer waiting time. Assuming that the live city data is available via REST APIs, it is still a non-trivial task to analyse and draw insights from the data since this involves data collection, data cleaning and analytics to arrive at some usable conclusions. If mashup tools supported graphical blocks for each of the tasks that the administrator needs to perform, the task would become simpler as it would only involve specifying the data-flow between various graphical components. The end result is an application which does real-time Big Data analytics on sensed traffic data and uses the analytics results to re-route the taxi fleet on the fly.

aFlux, a new actor-model based [87] mashup tool (Chapter 4), has been developed to overcome the existing limitations of the state-of-the-art mashup tools [109]. In this chapter, we validate the graphical programming concepts for Spark in aFlux, i.e. how to enable Spark (version 2.2.0) programming at a higher level with modular components in flow-based programming paradigm. Thus, the problems can be broadly summarised as:

(TB1) Problem statements for supporting flow-based Spark programming

PS1 Diverse data-representational styles, APIs and libraries centred around Spark make it difficult 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 together as modular components. The composability of these components is discussed in Section 5.6.1.3.

PS2 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 mashup tools, this difference in the computation model has been addressed by introducing additional auxiliary components in the mashup tool level called as *'bridge'* components (Section 5.8.2) and starting the code generation process at the last component of the user flow.

5.1 Structure

This chapter is structured in the following way.

- Section 5.2 discusses the design issues for supporting flow-based Spark programming.

- Section 5.3 summarises and compares the different data abstractions present in the Spark ecosystem. The selection of specific data abstractions for supporting flow-based Spark programming is also done here.

- Section 5.4 classifies the various APIs found in the Spark ecosystem based on their functionality and method signature.

- Section 5.5 summarises the conceptual approach and also enlists the design decisions taken to solve the design issues discussed in Section 5.2.

- Section 5.6 discusses key properties of a graphical component, flow compositional rules, flow validation and generation of native Spark application from a graphical flow.

- Section 5.7 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.

- Section 5.8 details the prototyping of the approach in aFlux.
- Section 5.9 discusses the evaluation of the approach.

Appendix A is attached to this chapter and contains some of the figures and code listings discussed here.

5.2 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. But in order to support such a scenario, there are several design issues which needs to be addressed by the conceptual approach.

SC1 Design of a modular Spark component (addresses PS1) 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 them. 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 so designed that their functionality is independent from each other and contain 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. [37, 95].

The first concern in this regard is deciding the *granularity level* of such a Spark component to make it *modular* in its design. The Spark APIs essentially accomplish one small task within the Spark engine like converting data from one form to another. Hence, in order 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. A developer while programming the application context configures it with various options like providing a name for the Spark application, deciding whether the application runs in a distributed mode or not, specifying the address of the

Spark cluster manager etc. 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.*

SC2 Data Transformation Approach (addresses PS1) Data transformations in Spark can be achieved via *RDD based operations* or the newer *declarative APIs* which operate on 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. But 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 a desired result.

SC3 Design of the translator model (addresses PS2) The translator model which takes the graphical flow and auto-generates the Spark application program should *parse the auxiliary components* (Section 5.8.2) properly as they do not represent any Spark API but are used to bridge the difference in computational model. Further, 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 wrong formulation of a Spark flow are handled at the mashup tool level.* Once the translator model accepts the Spark flow, it should definitely result into 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.

5.3 Data Abstractions in Spark

In this section, various transformations and data abstractions supported by Spark are discussed in detail to give an understanding of how Spark is used from a developer's perspective. The general introduction to Spark, its execution paradigm together with its varied ecosystem has been covered in Section 2.6. 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 [31], DStream [169], DataFrame [30, 171, 34] and Streaming DataFrame [26] *to manage and organise user data within its run-time environment* and several libraries have introduced data abstractions customised for different cases. Table 5.1 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 chapter, we use the term *'data abstraction'* for maintaining homogeneity and clarity in discussions. Fine-grained operations are possible through the declarative APIs of Spark, i.e. all APIs of libraries listed in Table 5.1, 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. Listing 5.1illustrates *'map'*, a higher level construct applying a UDF to accept *tuples of type-String* and produce *tuples of type-Integer*.

```java
//read data from file system
JavaRDD<String> lines = sc.textFile("data.txt");
//map each line to its length
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
public Integer call(String s) { return s.length(); }
});
```

Listing 5.1: RDD abstraction for batch processing

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 [28], 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 as shown in Figure 5.1.

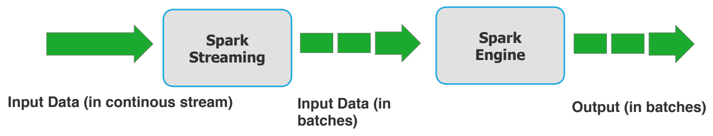


Figure 5.1: Spark Streaming: internal workings — *concept of micro-batches*, as in [25]

Spark Streaming provides a high level abstraction called *'discretized stream'* or *'DStream'* [169] 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. Internally, a DStream is nothing but a sequence of RDDs. In-order to use the DStream abstraction in Spark Streaming, there are a few sequential steps which a developer needs to follow.

Consider an example [25] where we are interested in counting the number of words in the text data received from a data server listening on a TCP socket.

1. First, the developer needs to create a streaming context specifying the number of execution threads and the batch interval. In this example, two execution threads and a batch duration of 1 second have been used.

2. Second, the developer can create a DStream using the previously specified streaming context to represent the live stream of data coming from the TCP socket. Every record in this representation is essentially a line of text.

3. Third, the developer needs to split the lines into words by spaces for which specialised DStream operation like *'flatMap'* can be used. This operation essentially creates a new DStream by generating multiple records from a single record of the original DStream representation.

4. Fourth, in order to count the number of words the DStream created in step three is mapped to a DStream pair (words,1) via a *'PairFunction'* object. After this, it is reduced to obtain the frequency of words in each batch of data.

5. Fifth, the processed output can be printed or sent for further processing.

6. Lastly, the developer needs to specify some additional steps like to start the computation and wait till the computation finishes to terminate in order to trigger the stream processing application.

Listing 5.2 shows the overall Java code which a developer needs to write in order to realise the above discussed example.

```java
// Step 1: Create a local StreamingContext with two working thread and batch
    interval of 1 second
SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("
    NetworkWordCount");
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(1)
    );

// Step 2: Create a DStream that will connect to hostname:port, like localhost
    :9999
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost",
    9999);

// Step 3: Split each line into words
JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(x.split(" ")).
    iterator());

// Step 4: Count each word in each batch
JavaPairDStream<String, Integer> pairs = words.mapToPair(s -> new Tuple2<>(s,
    1));
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey((i1, i2) -> i1
    + i2);

// Step 5: Print the first ten elements of each RDD generated in this DStream
    to the console
wordCounts.print();
```

```
// Step 6: Start the computation & Wait for the computation to terminate
jssc.start();
jssc.awaitTermination();
```

Listing 5.2: Example of using DStream in Spark

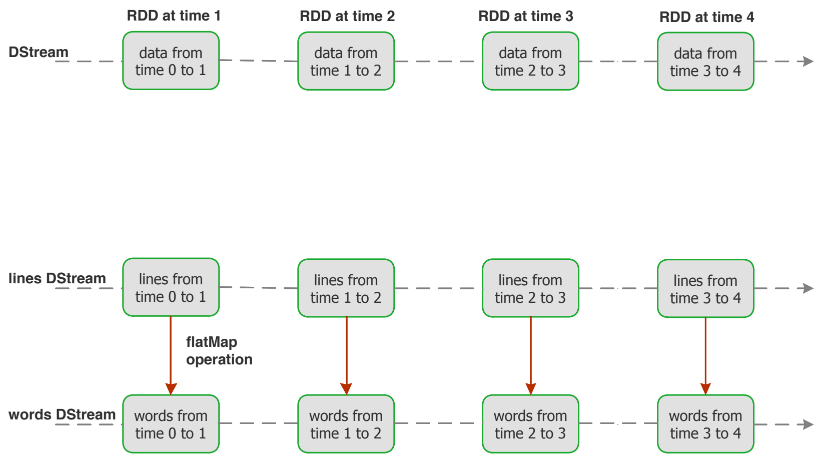


Figure 5.2: Internal working pattern of DStream data abstraction, as in [25]

DStream, the basic abstraction provided by Spark Streaming is internally a continuous series of RDDs, Spark's immutable distributed dataset abstraction. Hence, each RDD in a DStream contains data from a certain time interval only as shown in Figure 5.2. Any operation applied on a DStream is translated to operations on RDDs. For instance, to convert a stream of lines to words in the above example, the *flatMap* operation was used. Internally, this operation got translated as an operation applied to each individual RDD contained in the lines DStream which generated the corresponding RDDs of the word stream as shown in Figure 5.2. 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.

To summarise, Spark Streaming library employs the *DStream* abstraction [169], which 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 on 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* library which makes it possible to apply machine learning models learned offline, on Streaming data.

DataFrame Data Abstraction The *DataFrame API* [30], 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 environment using user-defined schema and DataFrame is created as 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. Listing 5.3 shows the overall Java code of using DataFrame in Spark where the developer creates a DataFrame from a JSON file with the schema inferred automatically by Spark. The DataFrame thus created, can be passed all kinds of SQL queries for data extraction as indicated in the code listing below. It also supports a *SQL function* which allows to run SQL queries programmatically and return the results in the form of a Dataset. For most complicated scenarios, the developer needs to specify the schema before a DataFrame can be created from the data source.

```java
Dataset<Row> df = spark.read().json("people.json");

// Displays the content of the DataFrame to stdout
df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|aaaaaaa|
// |  40|   bbbb|
// |  29| cccccc|
// +----+-------+

// Select people older than 30
df.filter(col("age").gt(30)).show();
// +---+----+
// |age|name|
// +---+----+
// | 40|bbbb|
// +---+----+

// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|aaaaaaa|
// |  40|   bbbb|
// |  29| cccccc|
// +----+-------+
```

Listing 5.3: Example of using DataFrame in Spark

Streaming DataFrame Data Abstraction *Spark Structured Streaming* is a fault-tolerant stream processing engine built on top of 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.

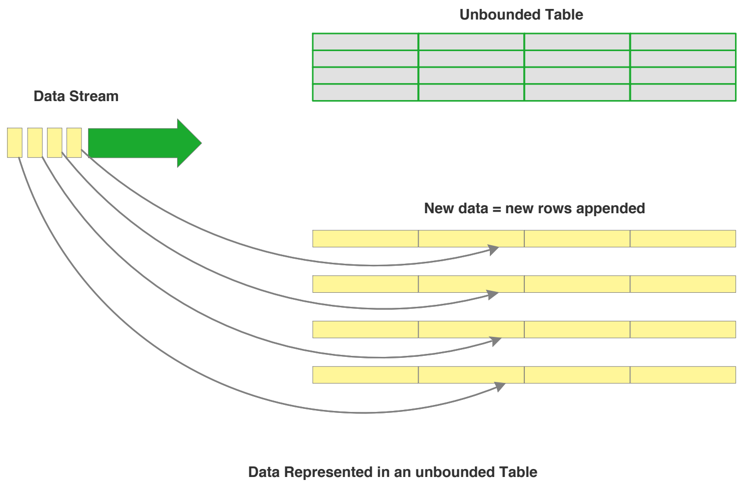

Unbounded Table
Data Stream
New data = new rows appended
Data Represented in an unbounded Table

Figure 5.3: Internal working pattern of Streaming DataFrame data abstraction, as in [26]

The key 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 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. The internal handling of streaming data, in the form of an unbounded table in this data abstraction/programming model, is shown in Figure 5.3.

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:

Complete Mode In this mode, the entire updated result table is written as output.

Append Mode In this mode, only the newly added rows to the result table since the last point of trigger are written as output.

Update Mode In this mode, only the rows that were updated in the result table since the last point of trigger are written as output. This differs from the complete mode as this outputs only the rows that were changed since the last point of trigger.

Figure 5.4 shows the overall working of the Streaming DataFrame data abstraction used in Spark Structured Streaming. Consider the same example which we discussed in the

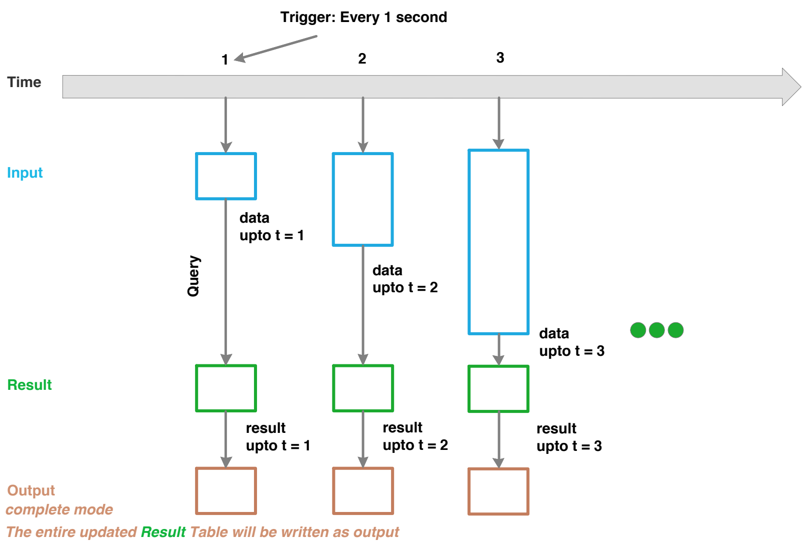


Figure 5.4: Programming model of Spark Structured Streaming, as in [26]

DStream data abstraction section, i.e. we are interested in counting the number of words in the text data received from a data server listening on a TCP socket [25]. To express the same streaming application using the Streaming DataFrame data abstraction in Spark Structured Streaming, the developer needs to follow some steps.

1. First, the developer needs to create a streaming DataFrame that represents text data received from the server which is transformed to obtain the word counts. This DataFrame represents an unbounded table containing the streaming data. The unbounded table contains one column of strings named 'value' and each line in the streaming dataset becomes a row in the table.

2. Second, the DataFrame is converted to a string Dataset, so that '*flatMap*' operation can be applied to split each line into multiple words. This Dataset contains all the words received.

3. Third, a new Streaming DataFrame needs to be created which contains all the unique values contained in the Dataset created in step 2 and counting them. Since it is a streaming DataFrame, hence it represents the running word counts.

4. Fourth, the developer needs to specify the output mode and start the streaming operation.

Listing 5.4 shows the overall Java code which a developer needs to write in order to realise the above discussed example. Figure 5.5 shows the internal workings of the Streaming DataFrame data abstraction with the help of an example.

It is interesting to note that this data abstraction is incompatible with the Spark ML library. This is because the incremental processing programming model of Spark Struc-

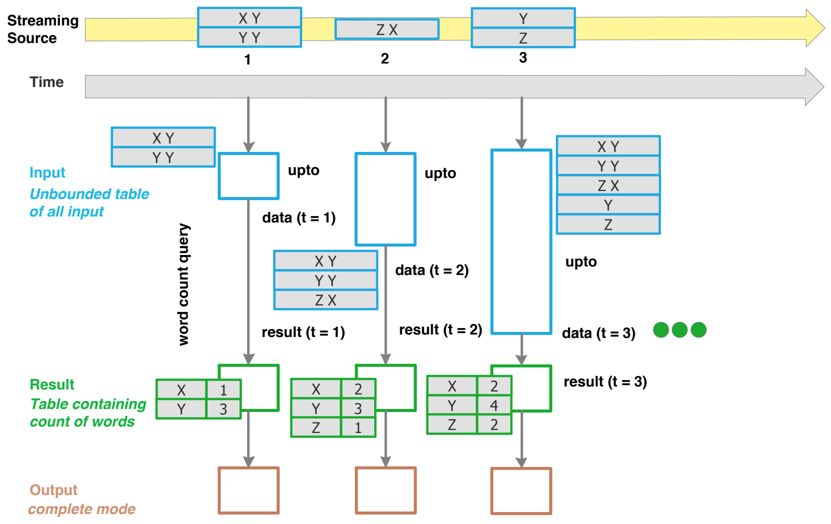


Figure 5.5: Programming model of Spark Structured Streaming instantiated with an example, as in [26]

tured Streaming programming is not compatible with the Spark ML processing model, where repeated iterations are carried out on entire datasets.

```
// Step 1: Create DataFrame representing the stream of input lines from
    connection to localhost:9999
Dataset<Row> lines = spark
  .readStream()
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load();

// Step 2: Split the lines into words
Dataset<String> words = lines
  .as(Encoders.STRING())
  .flatMap((FlatMapFunction<String, String>) x -> Arrays.asList(x.split(" ")).
    iterator(), Encoders.STRING());

// Step 3: Generate running word count
Dataset<Row> wordCounts = words.groupBy("value").count();

// Step 4: Start running the query that prints the running counts to the
    console
StreamingQuery query = wordCounts.writeStream()
  .outputMode("complete")
  .format("console")
  .start();

query.awaitTermination();
```

Listing 5.4: Example of using Structured Streaming in Spark

5.3.1 What is a Spark API?

Developers typically use a user-facing method invocation to achieve large scale data transformations. These developer-facing methods are called *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, a 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 [60]:

1. low-level transformations, actions and control of dataset is necessary.

2. the dataset to be analysed is unstructured, i.e. cannot be represented in relational format.

3. 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 DataFrames and Datasets 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.* This 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, (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 a number of benefits [60]:

1. **Static-typing and run-time safety** If we consider *'static-typing and run-time 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 the following illustration would suffice. For instance, we cannot detect any syntax errors in a SparkSQL string query until runtime where as 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 get detected at compile time. Here, even analysis errors can be detected at compile time thereby saving development time. The spectrum of type-safety along with syntax and analysis error in different APIs of Spark is shown in Figure 5.6.

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 easier than with relation algebra type expressions offered in RDDs.*

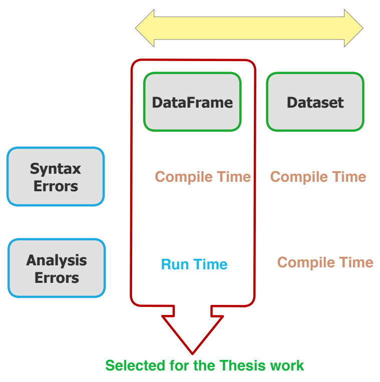


Figure 5.6: Spectrum of type-safety along with syntax and analysis error in different APIs of SparkSQL, as in [60]

When to use DataFrame/Dataset If our requirements are to have a rich semantics, high-level abstractions over datasets and easy to use domain specific APIs then DataFrame or Dataset APIs form a good candidature. Additionally, 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 the 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 the thesis contributions, *the DataFrame APIs of Spark have been selected* and are supported in the graphical Spark programming. 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 a end-user tool. Additionally, it introduces strong 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 difficult to pre-define 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 dataset, introduces easy to use domain specific APIs, offers a rich 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 fairly easy to implement in a generic manner and will be described in Section 5.5.

5.3.2 Data Abstractions in Spark: A Comparison

Spark libraries have been built on different abstractions and support various data abstractions for interaction as listed in Table 5.1. *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 5.2. There are cases where interoperability between different Spark libraries is just not possible. For example, Spark Structured Streaming library cannot be used with the DataFrame APIs of Spark ML library. This poses a serious 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, in order to apply a machine learning model on streaming data created using the Spark ML we have to do a number of internal conversions. First, the streaming data represented in DStream data abstraction needs to be converted to RDD which is supported as indicated in Table 5.2. 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. In detail, Spark ML library has been built on DataFrame API and introduces the concept of Pipelines. Pipeline is a model to pack the stages of machine learning process and produce a reusable machine learning model. Reusable models can be persisted and deployed on demand. The streaming data when handled using Spark Streaming library usually uses the DStream data abstraction. The library offers 'Transform' and 'ForEachRDD' methods to access RDDs (since DStream internally represents a continuous series of RDDs). Out of the two available methods, the 'Transform' method applies the user-defined transformation on RDDs and 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. 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.

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

Table 5.1: Spark libraries and their supported data abstractions

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	-

Table 5.2: Interoperability between different Spark data abstractions

(TB2) Analysis results: selection of data abstractions for conceptual approach

From the available data abstractions of Spark, DStream and DataFrame (including Streaming DataFrame), have been selected for the thesis work. The conceptual approach described in Section 5.5 supports only the transformations accessible via untyped, i.e. DataFrame APIs (Figure 5.6) and using the aforementioned data abstractions. Any transformation making use of RDD based approach is not supported. Nevertheless, the implementation of the conceptual approach, for reasons of interoperability between these data abstractions, makes use of RDD when converting data represented in one of the above data abstraction into another.

(TB3) Reasons for selection

1. It is possible to represent DStream and DataFrame APIs as modular graphical components that can be wired together. The different input parameters required by an API can be specified by the user from the front-end. This addresses the challenge SC1 (Table 5.3).

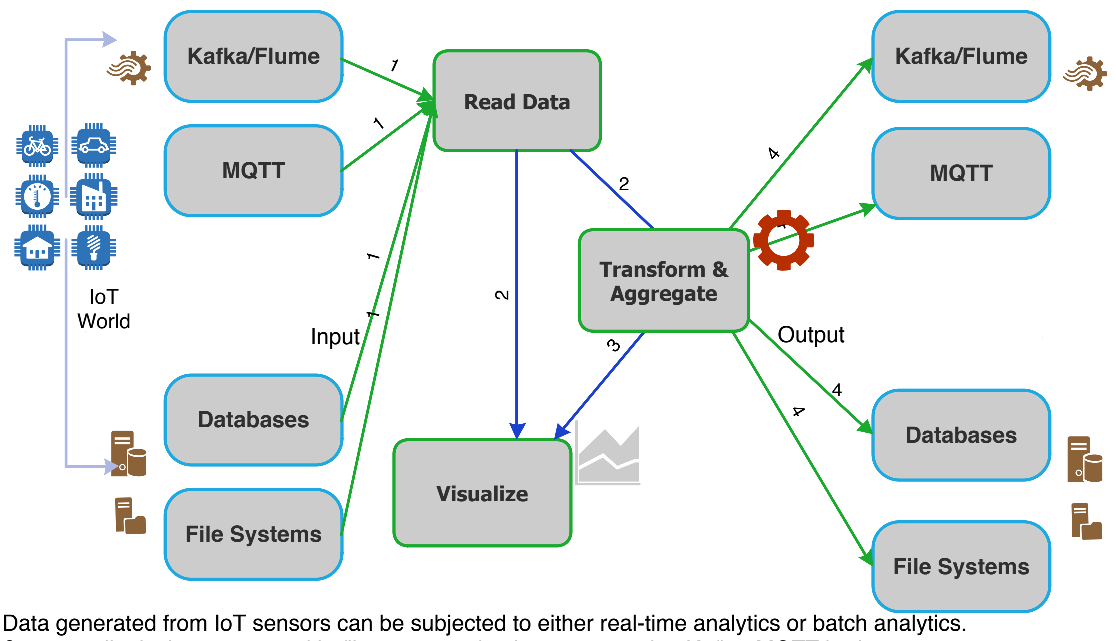
2. Data Frame APIs based on these aforementioned data abstractions prevent the usage of user-defined functions (UDFs) and provide predictable input and output types for each operation—the tool can then check the associated schema changes for compatibility during the flow validation, i.e. before code generation and compilation. This addresses the design issue SC2 shown later in Table 5.3.

5.4 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 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 same category and used in a graphical flow can be invoked in a similar generic way. Hence, every modular Spark component uses a generic method invocation statement for its standalone method invocation as listed in Table 5.4 and explained in Section 5.8.4.

RDDs support only two kinds of operations [31], 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 just remember the transformations applied on a file and are computed when the results are actually 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 Big Data application follows the model of reading data, analysing it and finally giving out the result. A typical Spark application allows follows this norm. Here, we have classified various APIs of different Spark libraries using the notion of logical dataflow model as shown in Figure 5.7. Accordingly, we have three categories as discussed below. *The visualisation of resultant datasets is not part of a typical Spark application.*



a. Data generated from IoT sensors can be subjected to either real-time analytics or batch analytics. So, accordingly they are stored in file systems, databases or send to Kafka, MQTT brokers etc.
b. The Big Data processing chiefly consists of 4 steps:
   1. Read datasets either from streams or databases.
   2. Visualize the data or transform the data.
   3. Visualize the transformed datasets.
   4. Save the resultant datasets to databases or send them back as streams.

Figure 5.7: Logical data-flow model of a typical Big Data application

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'* Basically creates a DataFrame

2. readStream() : *returns a 'DataStreamReader'* Basically 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 thesis. Some of the examples for this are listed in Table A.1 (SN 1 - 18). The *'DataStreamReader'* class provides methods to read streaming datasets from external environments. Some of the examples for this are listed in Table A.1 (SN 19 - 25).

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. Some of the examples for this are listed in Table A.1 (SN 26 - 30).

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. Examples for this are listed in Table A.1 (SN 42 - 69).

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. Examples for this are listed in Table A.1 (SN 70 - 78).

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 may be even a new data abstraction.* Such a re-usable object can be applied on other data frames that match the schema of the dataset using which the re-usable 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. Examples for this are listed in Table A.1 (SN 79 - 80). 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. Examples for this are listed in Table A.1 (SN 31 - 41).

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 5.4 and explained in Section 5.8.4.

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.

5.5 Conceptual Approach for Flow-based Spark Programming

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

(i) Modelling of graphical Spark flows (addresses: [PS2 → SC3]) Typically, a Spark application consists of three main parts, i.e. reading data from file systems or streaming sources, like IoT sensors, applying analytical transformations on those datasets and finally writing the results to either file systems or publishing them as real-time streams, as the case may be as depicted in Figure 5.7. One of the primary assumptions is end-users would typically follow the above high-level model 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) [6], 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 are not supported because the data abstractions output from each component are 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 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.

(ii) Suitable data abstraction. (addresses: [PS1 → SC1 & SC2]) For expressing a Spark program in a flow-based programming paradigm, the most suitable *data abstractions* for wiring together were selected (refer to the coloured tool boxes **TB2** & **TB3** in Section 5.3.2). 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 in a generic manner. 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.

From a high-level perspective, the process of programming Spark graphically consists of validating the flow to ensure that it yields a compilable and runnable Spark application. Figure 5.8 illustrates the key concepts for creating a Spark job involving machine learning algorithms or real-time analytics, by creating a unidirectional flow of connected components/graphical blocks.

Table 5.3 lists the design decisions taken in the formulation of the conceptual approach and how they solve the challenges **SC1 - SC3**.

Table 5.3: Design decisions

SC No.	Challenge	Solution	How it solves the challenge ?
SC1	**Design of a modular Spark Component**	1. Encapsulating a set of Spark APIs invoked in specific order to perform a data-analytic operation in the form of a modular component along with fine-tuning of the operation via user-supplied parameters. 2. Representing the Spark APIs in the form of modular components.	• Makes the graphical programming easy and intuitive where the user has to focus solely on specifying the data transformation logic, i.e. read, transform and eject the results of data analytics, in the form of wiring up of components, along with customisation of individual components via parameter passing and leaving all portions of the Spark driver program not related with data transformation explicitly to be generated by the system.
SC2	**Data Transformation Approach**	1. Selection of untyped APIs of Spark making use of *DataFrame*, *DStream* and *Streaming DataFrame* data abstractions.	• The untyped APIs detect syntax errors at compile time, provide columnar access to data and support easy to use APIs in a domain specific language (coloured tool box **TB2** & **TB3**). Manual inclusion of checks in the conceptual approach to detect analytical errors during the flow validation, e.g. *accessing and using a non-existent columns in a DataFrame*.

Continued on next page

Table 5.3 – continued from previous page

SC No.	Challenge	Solution	How it solves the challenge ?
SC3	Design of the translator model	1. Supporting auxiliary components to express a Spark program in flow-based programming paradigm. 2. Maintaining stand-alone method implementation of Spark APIs.	• The auxiliary components bridge the difference in execution model of Spark and mashup tools (Section 5.8.2). • The method invocation technique is same for all Spark components, hence it is generic. For supporting a new Spark API, the developer needs to add a new stand-alone method implementation for it, thereby making the approach easily extensible.

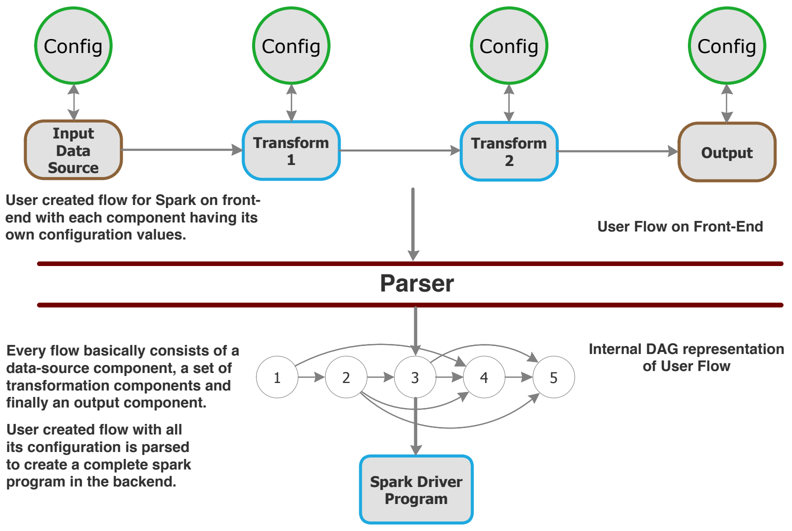

Config
Config
Config
Config
Input Data Source
Transform 1
Transform 2
Output
User created flow for Spark on front-end with each component having its own configuration values.
User Flow on Front-End
Parser
Every flow basically consists of a data-source component, a set of transformation components and finally an output component.
User created flow with all its configuration is parsed to create a complete spark program in the backend.
1
2
3
4
5
Internal DAG representation of User Flow
Spark Driver Program

Figure 5.8: High-level view of key concepts: designing of Spark jobs via graphical flows

5.6 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 application. A typical Spark application like any other Big Data processing application consists of three essential stages, i.e. reading data from file systems or streaming sources, actual data transformation, writing the results of data analytics back to file systems or streaming sources as shown in Figure 5.7. However, an important question arises here: *What kind of graphical components would support such an application model and how to develop such components?* Figure 5.9 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 basic execution component used in a mashup tool, for instance these correspond to an actor 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*).

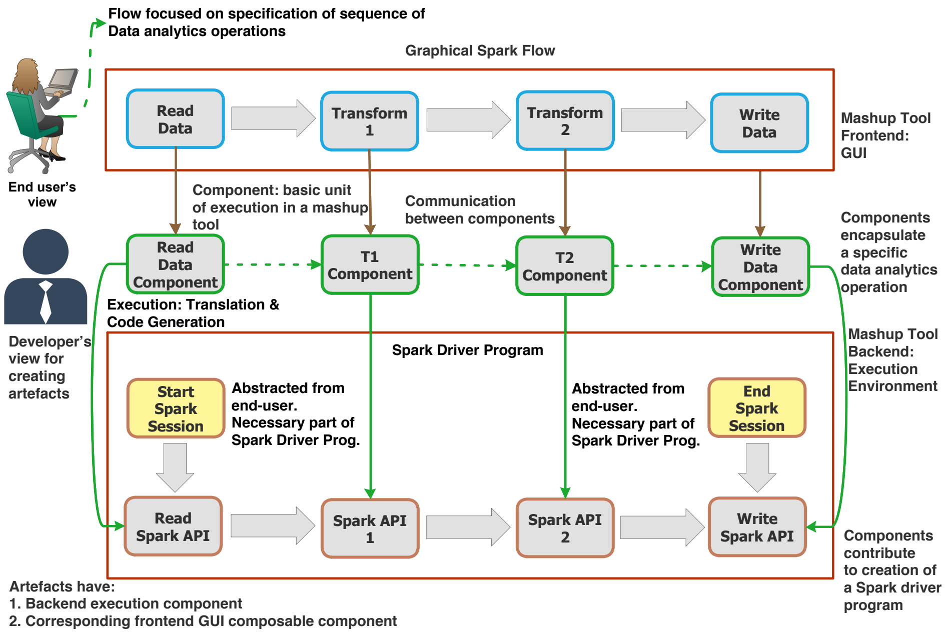

Flow focused on specification of sequence of Data analytics operations
Graphical Spark Flow
End user's view
Read Data
Transform 1
Transform 2
Write Data
Mashup Tool Frontend: GUI
Component: basic unit of execution in a mashup tool
Communication between components
Read Data Component
T1 Component
T2 Component
Write Data Component
Components encapsulate a specific data analytics operation
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

Figure 5.9: Developer's perspective for building Spark components for mashup tools

Every component takes a compatible data abstraction schema as input and produces a modified schema of the same data abstraction as discussed in Section 5.5. It is intuitive for the component developer to visualise the flow as a series of transformation of a specific Spark data abstraction schema. Inception of the flow begins by reading data/streaming data from external source, these components introduce the data abstraction schema to be used by later components in the flow. This is followed by components which accept this schema and apply transformation on it to produce an altered schema as output. A flow typically ends with a component which writes data to external storage or streaming sink which consumes the schema and marks the end of the flow. The approach can be broadly classified into three essential parts dealing specifically with: (i) Components, (ii) Flow validation and (iii) Application generation. All these parts will be explained below in separate sub-sections.

5.6.1 Components

A *'component'* is a basic unit of Spark flow composition. As discussed earlier, a component basically 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 from a streaming source like Kafka etc. Only declarative APIs are supported which take specific input and produce a specific output. The APIs operating on DataFrame, DStream and Streaming DataFrame data abstractions are supported which means the input to a component is an initial schema and the output is a transformed schema of the same type.

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 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.

5.6.1.1 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 Section 5.4. Accordingly, the high-level design decisions discussed above have been used to create different categories of components for supporting graphical flow-based Spark programming. Figure 5.10 shows the different kinds of components supported in aFlux for designing Spark jobs. One specific challenge is dealing with multiple incoming connections to a component in a mashup tool. Since in flow-based programming paradigm [121] of mashup tools, every component executes on receiving a message from its preceding component and the order of messages is not preserved. But, 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 a flow and their output is consumed by other components.

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 is 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 implementa-

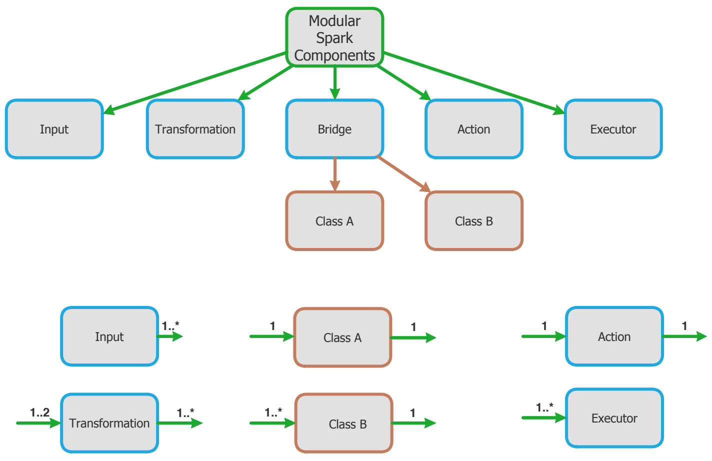


Figure 5.10: Classification of GUI components for supporting Spark in mashup tools

tion of Spark operation but are used to express the Spark sequenced operations (Section 5.8.2) in a flow-based programming paradigm. For example, they can impose an 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 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 Spark driver program. This has all the data required to generate a Spark application after the executor component has been triggered.

5.6.1.2 Key Attributes of a Component

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

(Data abstraction) colour denotes the data abstraction the component uses internally. It is used to differentiate between the types of data abstractions of the encapsulated APIs within the component actually work on. 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 (Section 5.6.1.3).

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

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.

Configuration Panel allows users to supply the necessary parameter for its optimal functioning.

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 5.11 shows the composition of a Spark flow using the modular components. From

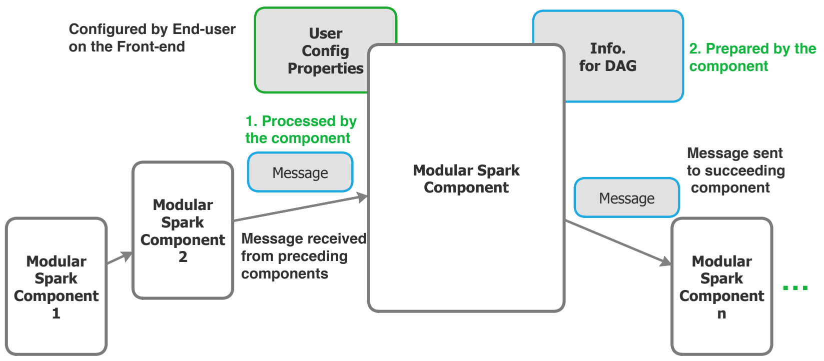

Configured by End-user on the Front-end
User Config Properties
Info. for DAG
2. Prepared by the component
1. Processed by the component
Message
Modular Spark Component
Message sent to succeeding component
Message
Modular Spark Component 2
Message received from preceding components
Modular Spark Component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

Figure 5.11: Composition of a Spark flow using the modular components

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 own specific information to be stored in the internal state. The positional hierarchy is also known as sequences in flow-based programming paradigm [121] which is to enforce correct sequence of execution of components in a flow. In this context, 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 is maintained and its list of permissible predecessors. 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.

5.6.1.3 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 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, with the exception of 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 [164]. A system is said to be composable when its different components can be assembled in various configurations to satisfy a user requirement [35]. The important criteria for components of a system to be composable are that they should be independent or self-contained and stateless [123]. A composable system is easier to validate because of its consistency to achieve a certain goal [123].

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 a 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 an unique internal name and an associated standalone method implementation for an analytic operation. Meta-data of this information are maintained on 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.

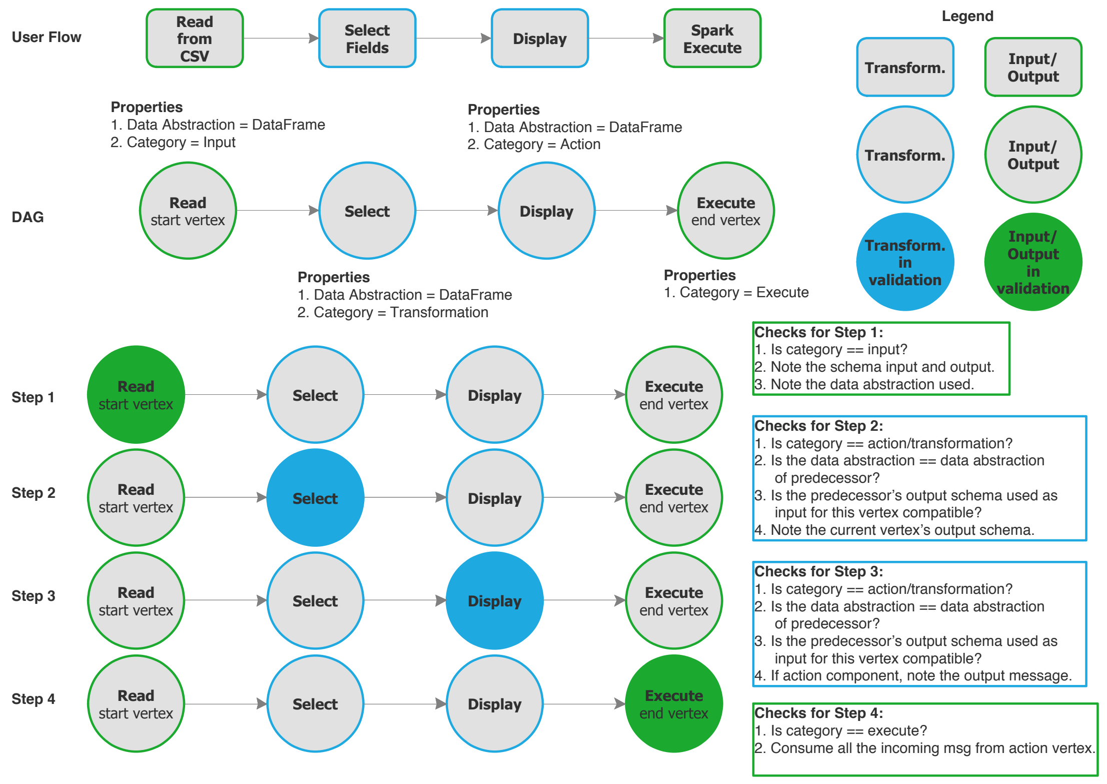
Then, it checks the vertex in the DAG and determines its category. If it is an input or transformation vertex, then, it uses the vertex's internal name to determine its associated method implementation. It calls the method via Java Reflection, passing the data abstraction schema and the vertex's user-supplied 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 simply 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. Based on this conceptual approach, Spark SQL, Spark ML, Spark Structured Streaming components have been prototyped in aFlux based on the DataFrame interface and Spark Streaming components based on the DStream interface.

5.6.2 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. 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 5.12 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.



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

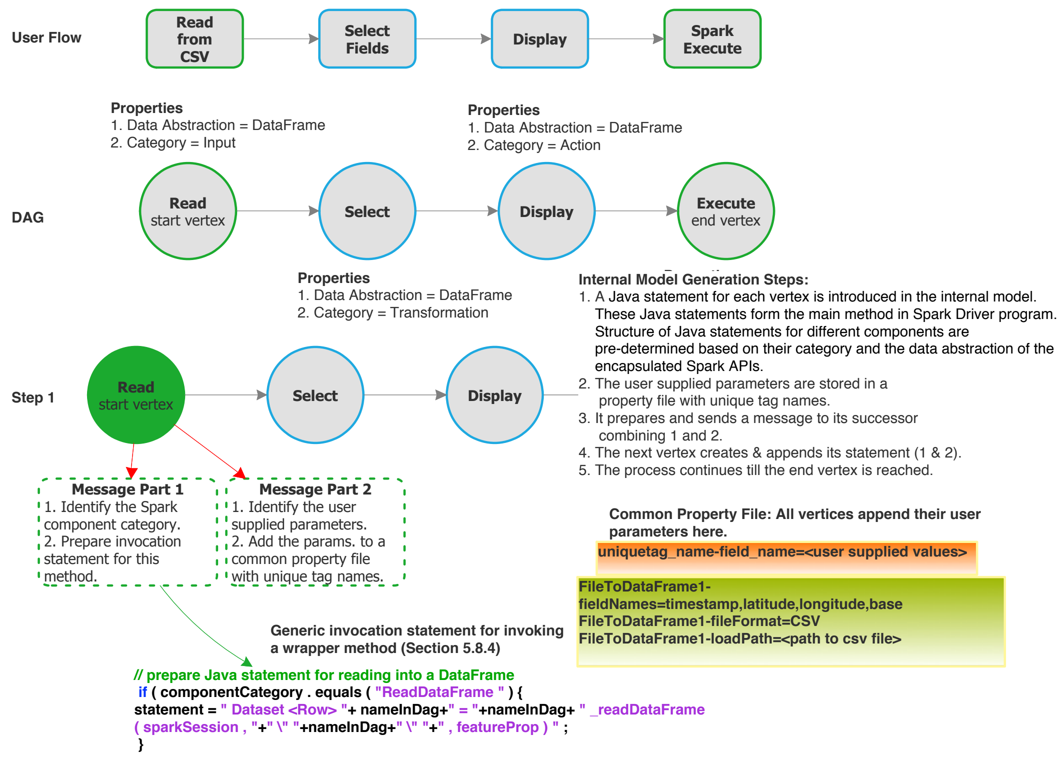
Checks for Step 1:
1. Is category == input?
2. Note the schema input and output.
3. Note the data abstraction used.

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.

Figure 5.12: Validation of a Spark flow: Steps



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.

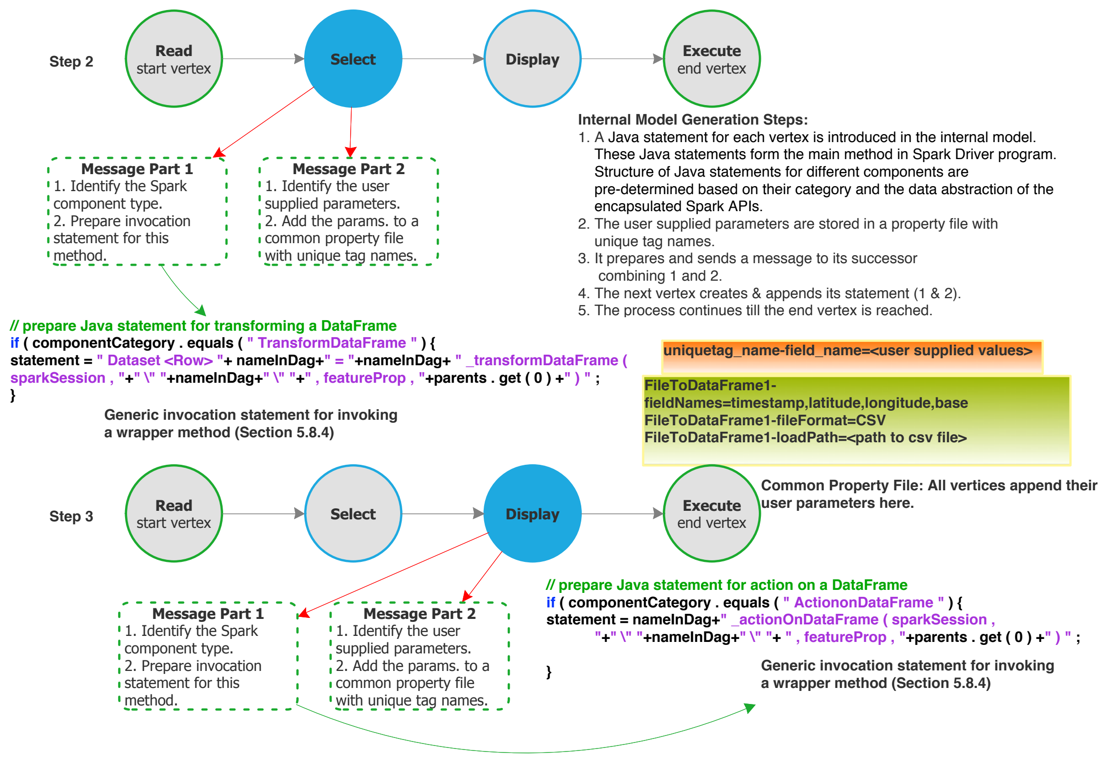
```
uniquetag_name-field_name=<user supplied values>
```

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

Generic invocation statement for invoking a wrapper method (Section 5.8.4)

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

Figure 5.13: Internal model representation of a Spark flow: Steps. Explanation in subsection 5.6.3. (contd. on Figure 5.14)


Step 2
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 (Section 5.8.4)
uniquetag_name-field_name=<user supplied values>
Common Property File: All vertices append their user parameters here.
Step 3
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 (Section 5.8.4)

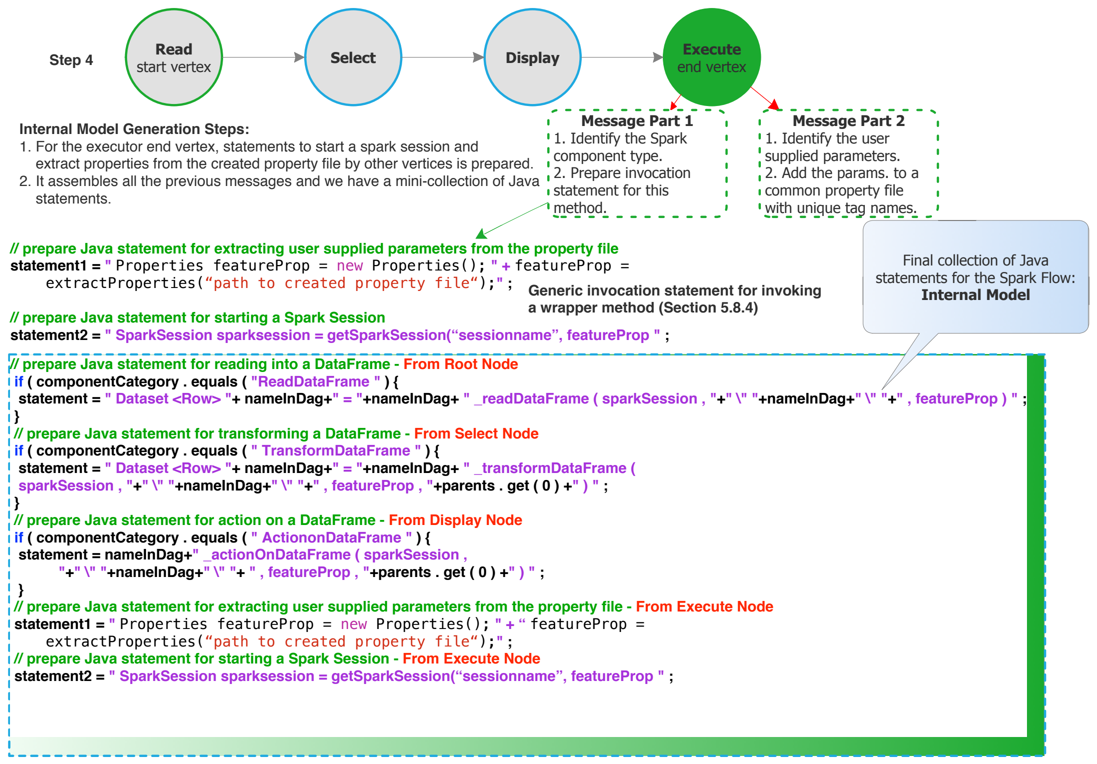
Internal Model Generation Steps:
1. A Java statement for each vertex is introduced in the internal model. These Java statements form the main method in Spark Driver program. Structure of Java statements for different components are pre-determined based on their category and the data abstraction of the 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.

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

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

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

}
```

Figure 5.14: Internal model representation of a Spark flow: Steps. Explanation in subsection 5.6.3. (contd. from Figure 5.13, contd. on Figure 5.15)


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 (Section 5.8.4)
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 " ;
```

Figure 5.15: Internal model representation of a Spark flow: Steps. Explanation in subsection 5.6.3. (contd. from Figure 5.14)

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.

5.6.3 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 run-time environment, as the components execute a DAG is regenerated. But this DAG not only captures the user flow but additional information is attached to every vertex during its creation, similar to intermediate code generation step in case of compilers [6].

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 [106, 104] are added for every vertex. These Java statements form the main methods in Spark Driver program. Structure of Java statements for different components are pre-determined based on the category of the component, i.e. whether they represent input, transformation, or action components (Section 5.4) 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 Secondly, the component checks the user-supplied parameters and creates a property file [165] 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. Figure 5.13, 5.14 and 5.15 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 Figure 5.15, 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.*

5.6.4 Generation of Spark Driver Program

After the preparation of the *internal model* of a Spark flow, it can be used to generate a complete Spark driver program.

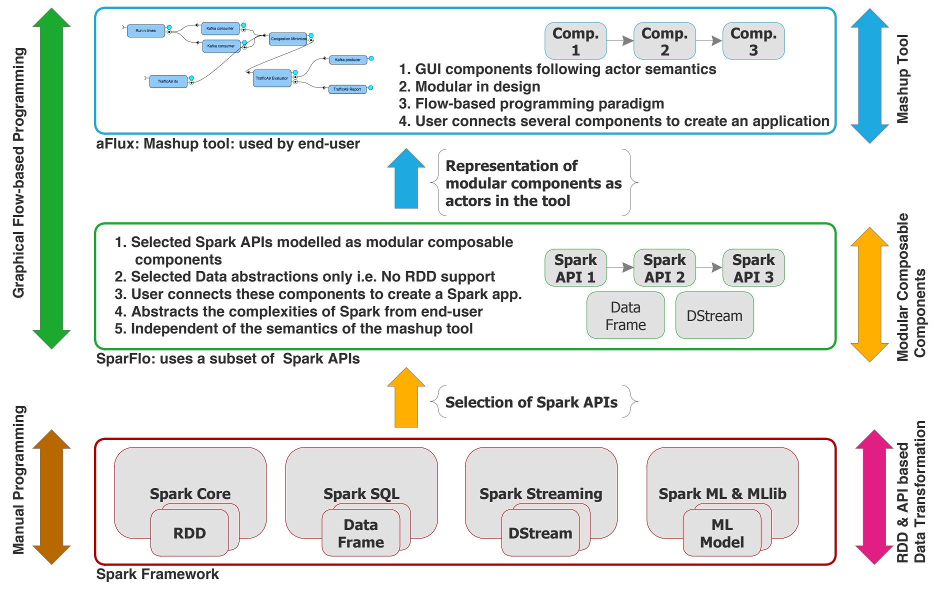
In the internal model, each vertex has contributed a Java statement and all these have been assembled by the last vertex of the DAG, i.e. which corresponds to the execute component of the user flow. These collection of statements become the statements within the main method of the Spark driver program. *API based code generator* is supplied with these prepared statements for producing a Spark application in Java programming language [146]. The internal details of how requisite Spark libraries are added to the final driver program and the generic implementation of Spark methods corresponding to each component are invoked using the Java statements from the internal model have been explained in Section 5.8.

5.7 SparFlo: A Subset of Spark APIs Bundled as Modular Components

Spark offers many different libraries in its ecosystem accessible either via 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 development of Spark applications via graphical flow-based programming more abstraction is necessary. With the approach described in the chapter, a subset of Spark APIs operating on DataFrame and DStream abstractions have been selected (coloured tool box **TB2**) which are compatible with the flow-based programming paradigm.

‘*SparFlo*’ is a library consisting of modular (Section 5.2) composable (Section 5.6.1.3) 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 a 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 Section 5.6.1.3. 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 (Section 5.8).

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 (Section 5.8.2). 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 in graphical tools based on flow-based programming paradigm.* Figure 5.16 illustrates the elements and characteristics of SparFlo.



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

Figure 5.16: SparFlo: A library of modular composable components using a subset of Spark APIs

5.7.1 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 Section 5.5, *iff*:

- The new Spark transformation is *accessible via an untyped API* (Section 5.3.1).
- It *must use either the DataFrame, DStream or Streaming DataFrame data abstraction* (coloured tool box **TB2**).

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 In order 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 the 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 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 Section 5.6. 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, its 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 5.4 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 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.

5.7.2 Applicability

The approach described is extensible to accommodate inclusion of newer Spark APIs and can be used to support flow-based Spark programming, subject to the following assumptions as discussed below:

Incompatible data abstraction or API type Spark functionalities accessible via declarative untyped APIs making use of either the DataFrame, DStream or Streaming DataFrame data abstractions are supported and can be represented as modular components for creating Spark flows. Typed APIs using Dataset are not supported as indicated in Figure 5.6. It is not possible to support data-transformations based on user-defined functions (UDFs) and RDDs as it goes beyond the dimensions of this thesis work (coloured tool boxes **TB1** & **TB2**).

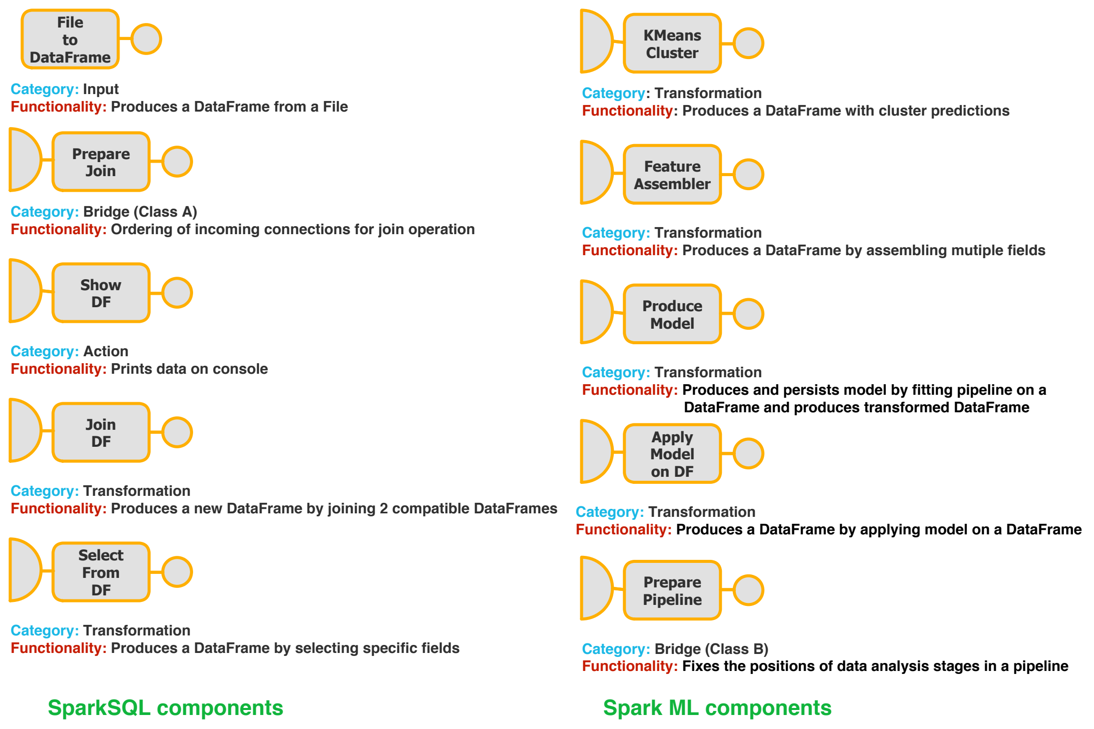
Restrictions in flow design Within the selection of Spark APIs and functionalities, there are restrictions on the kind of flow which can be composed to generate a compilable and runnable Spark driver program. Broadly, we have classified the a Spark flow into 3 kinds: (i) *unidirectional flows*: these flows flow in a single direction and have a start and an end point. If the components used in such a flow follow their positional hierarchy and other validation requirements, then such a flow is permitted in the system, (ii) *flows involving merges & distribution of dataset*: these flows typically include merging of datasets coming from different data sources and distributing of result sets from a transformation component to carry out different kind of analytics. Such flows are supported, *iff*, all merges and distributions pathways must be connected to the last component of the flow which is basically the *executor* component and (iii) *flows with feedback loops*: these flows typically have a connection to one of their previous components in the positional hierarchy of the flow. Such flows are not supported by the approach described here.

5.8 Implementation of the Conceptual Approach

In this section, we describe the various components from *SparFlo* implemented in order to realise flow-based Spark programming. Further, we describe the workings of the *'bridge'* components which help in expressing specific sequenced operations in Spark, like the Pipeline API for instance, in flow-based programming paradigms. Additionally, we zoom-in into the workings of the conceptual approach to generate a Spark driver program from a user created flow, i.e. the code generation aspects of the conceptual approach.

5.8.1 Components Implemented

To illustrate the workings of the approach and specific to the use-cases as discussed in Section 5.9, components belonging to the Spark SQL, Spark ML, Spark Streaming and Spark Structured Streaming have been implemented. Figures 5.17 and 5.18 list some of the implemented components for the user to create a Spark flow graphically. The figures also list their intended functionality as well as classify the components as per the categorisation of components discussed in Section 5.6 and Figure 5.10. The Spark SQL and Spark ML components use the DataFrame data abstractions of Spark while Spark Streaming uses DStream and Spark Structured Streaming relies on Streaming DataFrame respectively.



Category: Input
Functionality: Produces a DataFrame from a File

Category: Bridge (Class A)
Functionality: Ordering of incoming connections for join operation

Category: Action
Functionality: Prints data on console

Category: Transformation
Functionality: Produces a new DataFrame by joining 2 compatible DataFrames

Category: Transformation
Functionality: Produces a DataFrame by selecting specific fields

Category: Transformation
Functionality: Produces a DataFrame with cluster predictions

Category: Transformation
Functionality: Produces a DataFrame by assembling mutiple fields

Category: Transformation
Functionality: Produces and persists model by fitting pipeline on a DataFrame and produces transformed DataFrame

Category: Transformation
Functionality: Produces a DataFrame by applying model on a DataFrame

Category: Bridge (Class B)
Functionality: Fixes the positions of data analysis stages in a pipeline

Figure 5.17: Spark components for aFlux (contd. on Figure 5.18)

5.8.2 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. To be more specific, 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

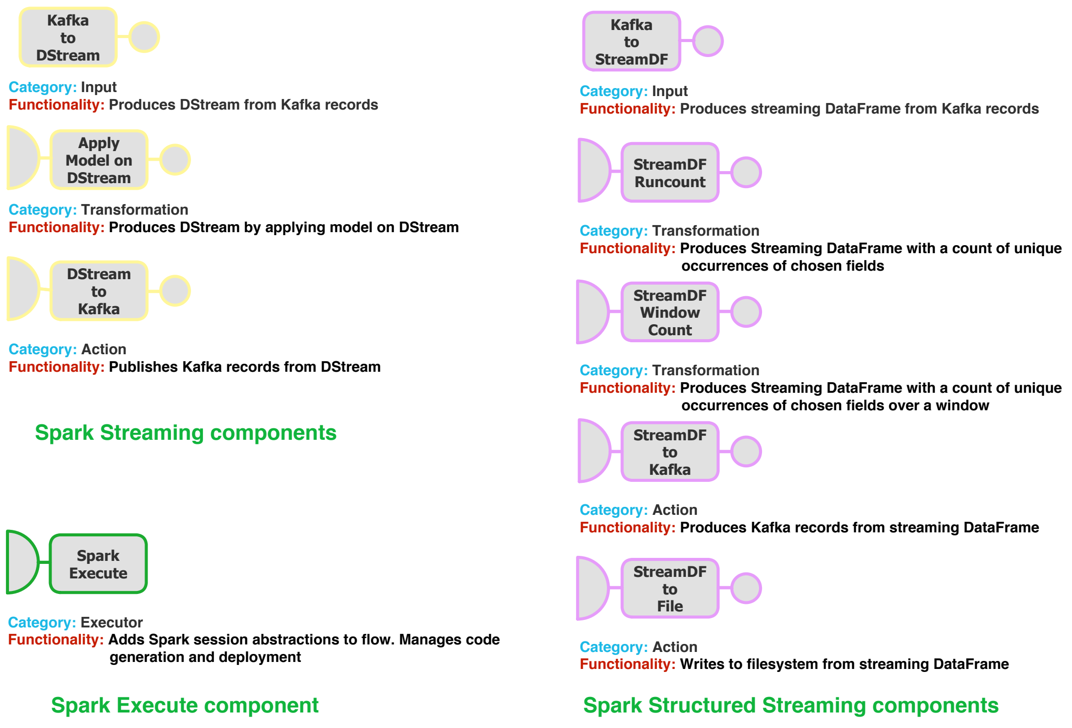

Kafka to DStream
Category: Input
Functionality: Produces DStream from Kafka records
Apply Model on DStream
Category: Transformation
Functionality: Produces DStream by applying model on DStream
DStream to Kafka
Category: Action
Functionality: Publishes Kafka records from DStream
Spark Streaming components
Spark Execute
Category: Executor
Functionality: Adds Spark session abstractions to flow. Manages code generation and deployment
Spark Execute component
Kafka to StreamDF
Category: Input
Functionality: Produces streaming DataFrame from Kafka records
StreamDF Runcount
Category: Transformation
Functionality: Produces Streaming DataFrame with a count of unique occurrences of chosen fields
StreamDF Window Count
Category: Transformation
Functionality: Produces Streaming DataFrame with a count of unique occurrences of chosen fields over a window
StreamDF to Kafka
Category: Action
Functionality: Produces Kafka records from streaming DataFrame
StreamDF to File
Category: Action
Functionality: Writes to filesystem from streaming DataFrame
Spark Structured Streaming components

Figure 5.18: Spark components for aFlux (contd. from Figure 5.17)

order of their arrival. Special handling is required in case of multiple incoming connections from components belonging to same component category which require ordering. Special components called *'bridge components'* are used to order the arrival of message 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 introduces two incoming connections with 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 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 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 5.19 illustrates the working of the two different kinds of bridge components.

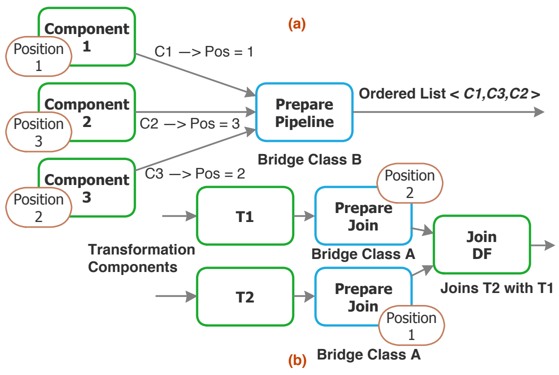


Figure 5.19: Bridge components: working details

5.8.3 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 Section 5.6 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 5.4. 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 its 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.

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

Table 5.4: Example: mapping of component category → Spark API type

5.8.4 Code Generation : Internal Model → Spark Driver Program

To explain how the *'internal model'* representation of a Spark flow created by the user is used to generate a complete Spark driver program, it is imperative to describe the internal mappings used in a component for such a purpose first. It has already been discussed that the components which the end-user uses to create a Spark flow on the front-end essentially abstract a set of Spark APIs to represent a specific data analytics operation. In other words, every component used by the end-user on the front-end corresponds to a generic implementation of the data analytic operation it represents making use of a specific set of Spark APIs internally. Table 5.5 lists an example list of the correspondence between different components to their encapsulated Spark APIs. For example, the component *'FileToDataFrame'* internally corresponds to a data analytic operation method which makes use of the *'DataFrameReader'* API of Spark SQL to read data from a file and create a DataFrame out of it. 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.

In order to make the code generation process as generic as possible, such generic method implementations making use of one or more Spark APIs are maintained. These are called as *'wrapper methods'*. Listing A.1 shows a wrapper method to read data from a file and create a DataFrame out of it. The *'FileToDataFrame'* component essentially corresponds to this wrapper method. Potential parameters to this wrapper method include file name, file path, file format etc. All such *'wrapper methods'* are bundled into a library called as *'Splux'*. Each component used by the user in a flow instantiates into an actor on deployment. Each actor 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. Figure 5.20 shows the invocation of a wrapper method in Splux library from the main method of the Spark driver program. Every component used in a flow produces an invocation statement for its wrapper method. These invocation statements are pre-determined based on the category of the component, i.e. whether they represent input, transformation, or action components (Section 5.4) and the data abstraction used, i.e. DataFrame, DStream or Streaming DataFrame, thereby forming the *'component class'* (Table 5.4). To reiterate, the wrapper method generic invocation statement for a component is specific to its *class*. These invocation statements form the main method in Spark Driver program.

The *wrapper methods* contained in *Splux* are *invoked via reflection.* From an implementation perspective, JavaPoet [94], an API based code generator is used to assemble the reflection statements to generate a Spark driver program. Listing A.2 lists the generated Spark driver program to read datasets from a file, to filter specific items and to display them.

SN	Component	Spark Library	Spark API(s)
1	FileToDataFrame	Spark SQL	DataFrameReader
2	SelectFromDF	Spark SQL	Select
3	JoinDF	Spark SQL	Join
4	PrepareJoin	-	-
5	ShowDF	Spark SQL	Show
6	DataFrameToFile	Spark SQL	DataFrameWriter
7	KMeansCluster	Spark ML	PipelineStage
8	FeatureAssembler	Spark ML	PipelineStage
9	PreparePipeline	-	-
10	ProduceModel	Spark ML	Pipeline, ModelWriter
11	ApplyModelOnDF	Spark ML	ModelReader
12	KafkaToDStream	Spark Streaming	KafkaReader
13	ApplyModelOnDStream	Spark Streaming, Spark ML	ForEachRDD, DataFrame, ModelReader
14	DStreamToKafka	Spark Streaming, Spark ML	ForEachRDD, DataFrame, ModelReader
15	KafkaToStreamDF	Spark Structured Streaming	StreamReader
16	FileToStreamDF	Spark Structured Streaming	StreamReader
17	StreamDFTokafka	Spark Structured Streaming	StreamWriter
18	StreamDFToFile	Spark Structured Streaming	StreamWriter
19	StreamDFToWindowCount	Spark Structured Streaming	Watermark,Window, GroupBy, Count
20	StreamDFToRunningCount	Spark Structured Streaming	GroupBy, Count
21	SparkExecute	All	Application Context

Table 5.5: Example: mapping of modular components → Spark API(s)

5.9 Evaluation via Examples

The evaluation scenario has been designed to capture the *modularity of the approach*, *code-abstraction from end-user*, *automatic handling of Spark session initialisation 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.

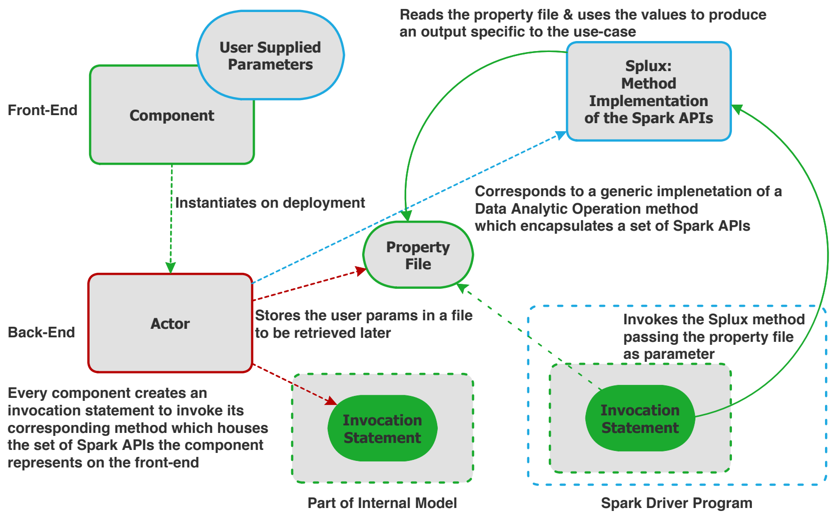

Reads the property file & uses the values to produce an output specific to the use-case
User Supplied Parameters
Front-End
Component
Splux: Method Implementation of the Spark APIs
Instantiates on deployment
Corresponds to a generic implenetation of a Data Analytic Operation method which encapsulates a set of Spark APIs
Property File
Back-End
Actor
Stores the user params in a file to be retrieved later
Invokes the Splux method passing the property file as parameter
Every component creates an invocation statement to invoke its corresponding method which houses the set of Spark APIs the component represents on the front-end
Invocation Statement
Invocation Statement
Part of Internal Model
Spark Driver Program

Figure 5.20: Invocation of generic Splux methods from the Spark driver program

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 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 model on real-time data and (iii) Spark Structured Streaming: for applying aggregations on real-time data.

5.9.1 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. All the code listings for this use case are in Appendix A.

5.9.1.1 Approach: Manual Programming via Java

The complete code for producing a machine learning model as discussed in this section via manual Java programming is shown in Listing A.3. 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. Line 2 in the code listing achieves this.

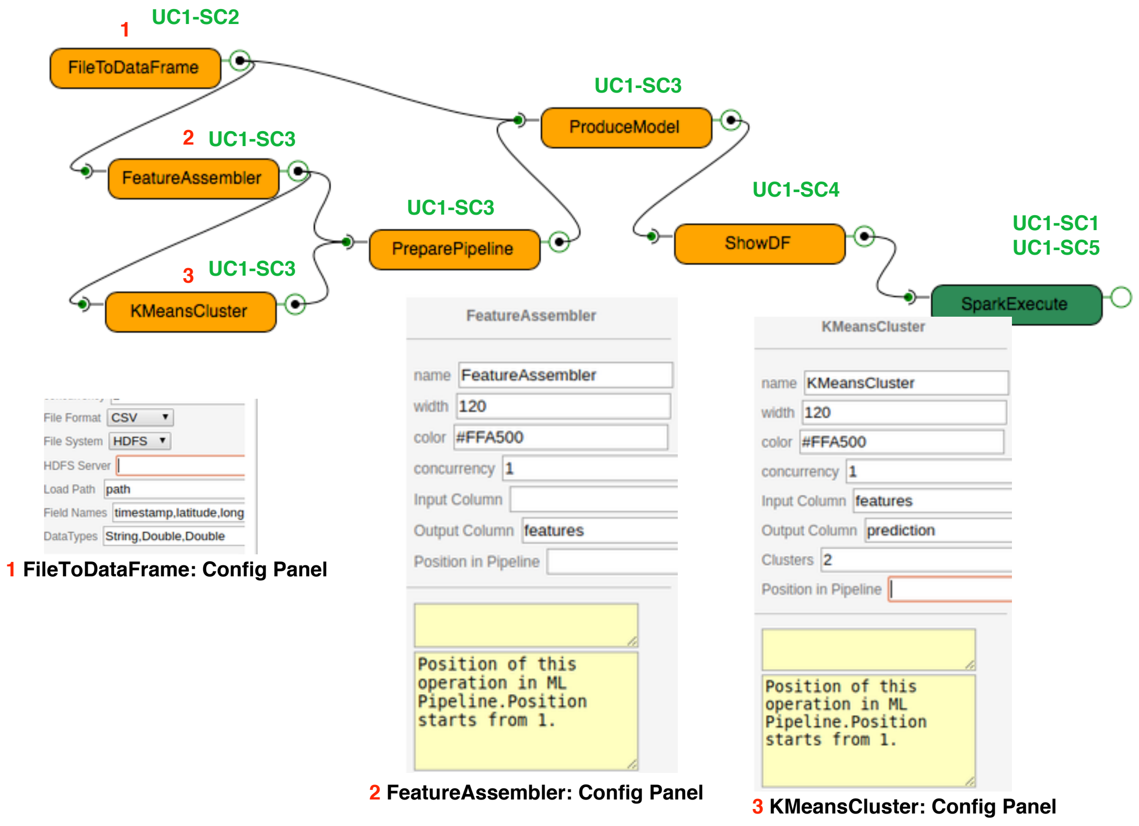
UC1-S2 Reading Datasets Data is read using Reader function of the SparkSession. DataFrame API views data as a table. Hence, schema must be supplied to it. Data in CSV format is read in this particular example. Line 4-16 in the code listing show how to read data into run-time environment using DataFrame reader API. A schema is defined for the dataset and supplied to the DataFrame reader which reads data into Spark run-time environment and produces a DataFrame from it.

UC1-S3 Data Transformations Here, 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 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. The lines 19-41 show the instantiation of the objects which are fed to the Pipeline API, which in turn is used to produce a model. Line 41 shows the production of a model fitted on the input data. PipelineModel writer API has been used to persist the model on file system for later application.

UC1-S4 Invocation of Actions Next thing is to display the transformed data. So 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 as show in line 44. 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 as show in line 46.

5.9.1.2 Approach: Graphical Programming via aFlux



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.

Figure 5.21: Spark flow in aFlux for producing machine learning model

The approach via aFlux enables the user to create the same Spark program by connecting graphical components together, leading to auto-generation of the Spark driver program. Figure 5.21 illustrates an aFlux flow for producing 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 Section 5.4, (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 a 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 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 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 configuration panel of both 'FeatureAssembler' and 'KMeansCluster', their respective position in pipeline must be specified. Figure 5.21 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 creation and termination of the application context.

Figures 5.24, 5.25 and 5.26 show the validation of the flow created for use case 1. Figures 5.27, 5.28, 5.29, 5.30 and 5.31 illustrate the steps of the internal model generation. The complete Spark driver program generated from the user flow is shown in Listing A.6 and its generated property file storing the user supplied parameters is shown in Listing A.7.

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 autogeneration of Spark session initialisation code, lines 26-32 in Listing A.6, which is vital for 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 provide easy customisation via user supplied parameters from the component's configuration panel as shown in Figure 5.21.

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 Section 5.6.1.3, the flow is validated for correctness, i.e. if such a flow would generated 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 obvious 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.

5.9.2 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. All the figures and code listings for this use case are in Appendix A.

5.9.2.1 Approach: Manual Programming via Java

The complete code for producing applying the machine learning model on streaming data as discussed in this section via manual Java programming is shown in Listing A.4. 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 duration of micro-batch as one of its input.

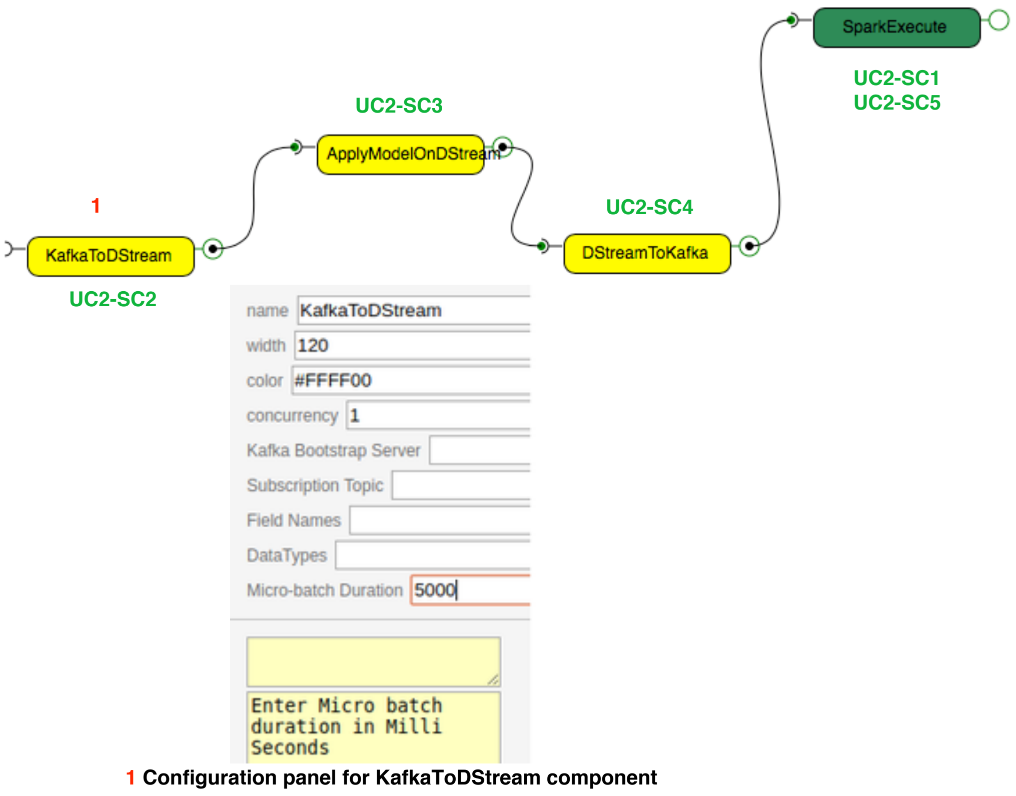
UC2-S2 Reading Datasets After the creating of a streaming application context, data is read from a compatible input source like Kafka. Kafka records published with topic 'trip-data' are read in the form of $< key, value >$ pairs into JavaPairDstream data abstraction. A map operation has been called to drop key from every record as show from line 5 to 24.

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 as shown in line 45.

UC2-S4 Invocation of Actions Action step involves pushing data out of Spark run-time environment, i.e. push back results to Kafka. Spark Streaming does not support built in Kafka listener. *'ForEachPartition'* method can be used to push data out of each partition. Lines 44 to 55 show how Kafka records are produced and published with topic *'enrichedData'*.

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.

5.9.2.2 Approach: Graphical Programming via aFlux



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

1 Configuration panel for KafkaToDStream component

Figure 5.22: Spark flow in aFlux for applying model to streaming data

Figure 5.22 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 encapsulates input category of Spark APIs as classified in Section 5.4, (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 is in correspondence to 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 in order 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 creation and termination of the application context.

Figure A.1 shows the validation of the flow created for use case 2. Figures A.2, A.3 and A.4 illustrate the steps of the internal model generation. The complete Spark driver program generated from the user flow is shown in Listing A.8 and its generated property file is shown in Listing A.9.

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, this components transparently reads the DStream data from the previous component, converts it into RDD and then invokes the saved ML model on it as shown in Listing 5.5. After applying the ML model, the resultant-set is again converted back from RDD to DStream and send as 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 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 ML model, user-defined functions are necessary to convert it back to DStream data abstraction. This process cannot be automated as the fields of 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.

```java
JavaDStream < String > msgDataStream = input.map(new Function < Tuple2 < String
    , String > , String > () {
 @Override
 public String call(Tuple2 < String , String > tuple2) {
  return tuple2._2();
 }
});

//Convert DStream to RDD
String[] fieldNames = properties.getProperty(transformDFPipeline[1] + "-
    fieldNames").split(",");
String[] dataTypes = properties.getProperty(transformDFPipeline[1] + "-
    dataTypes").split(",");
msgDataStream.foreachRDD(new VoidFunction < JavaRDD < String >> () {

 @Override
 public void call(JavaRDD < String > rdd) {
  JavaRDD < Row > rowRDD = rdd.map((Function < String , Row > ) record -> {
   String[] attributes = record.split(",");
   Object[] newArgs = new Object[attributes.length];
   for (int i = 0; i < attributes.length; i++) {
    newArgs[i] = mapToDataType(attributes[i], dataTypes[i]);
   }
   return RowFactory.create(newArgs);
  });

  List < StructField > fields = new ArrayList < > ();
  for (int i = 0; i < fieldNames.length; i++) {
   DataType dt = getDataType(dataTypes[i]);
   StructField field = DataTypes.createStructField(fieldNames[i], dt, true);
   fields.add(field);
  }
  StructType csvschema = DataTypes.createStructType(fields);


  // Apply Model created using DataFrame

  String modelPath = properties.getProperty(transformDFPipeline[2] + "-loadPath
    ");

  // Get Spark 2.0 session
  // Pipeline model object will be  serialized and sent from the driver to the
    worker
  PipelineModel savedModel = PipelineModel.read().load(modelPath);
  SparkSession spark = JavaSparkSessionSingleton.getInstance(rdd.context().
    getConf());
  Dataset < Row > msgDataFrame = spark.createDataFrame(rowRDD, csvschema);
  msgDataFrame = savedModel.transform(msgDataFrame);


  //Convert from RDD to DStream and send to Kafka
  String bootstrapServers = properties.getProperty(transformDFPipeline[3] + "-
    bootStrapServers");
  String topicToPublish = properties.getProperty(transformDFPipeline[3] + "-
    topicToPublish");


  JavaRDD < Row > toRDD = msgDataFrame.toJavaRDD();

  toRDD.foreach(rowrdd -> {
   Properties kafkaProp = new Properties();
   kafkaProp.put("bootstrap.servers", bootstrapServers);
   kafkaProp.put("key.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
   kafkaProp.put("value.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
   KafkaProducer < String,
   String > producer = new KafkaProducer < String,
   String > (kafkaProp);
   String data = rowrdd.get(0).toString();
   //Row to String
   for (int i = 1; i <= rowrdd.length() - 1; i++) {
    data += ",";
    data += rowrdd.get(i).toString();

   }
   producer.send(new ProducerRecord < String, String > (topicToPublish, data));
   producer.close();
  });

 }
});
```

Listing 5.5: Auto-conversion between different data abstractions

5.9.3 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. All the figures and code listings for this use case are in Appendix A.

5.9.3.1 Approach: Manual Programming via Java

The complete code for performing streaming operations on data read from Kafka as discussed in this section via manual Java programming is shown in Listing A.5. 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 and maps them to the schema supplied as indicated in line 4 of the application. Lines 9 to 17 show how Kafka records are read and mapped to a DataFrame abstraction where entire value part of the record is treated as a string. Lines 18 to 25 show how the value represented as a string is mapped to the schema provided earlier. 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. Lines 27 to 30 deal with watermarking to handle late data, grouping of columns, counting them and selection of results satisfying the condition.

UC3-S4 Invocation of Actions Spark Structured Streaming associates a streaming query with DataFrames. Only those transformation path 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. Lines 35 to 44 list a streaming query which publishes data back to Kafka.

UC3-S5 Terminating Application Context In the last step, *'awaitTermination()'* method is invoked on the streaming query as shown in line 46 so that it runs indefinitely.

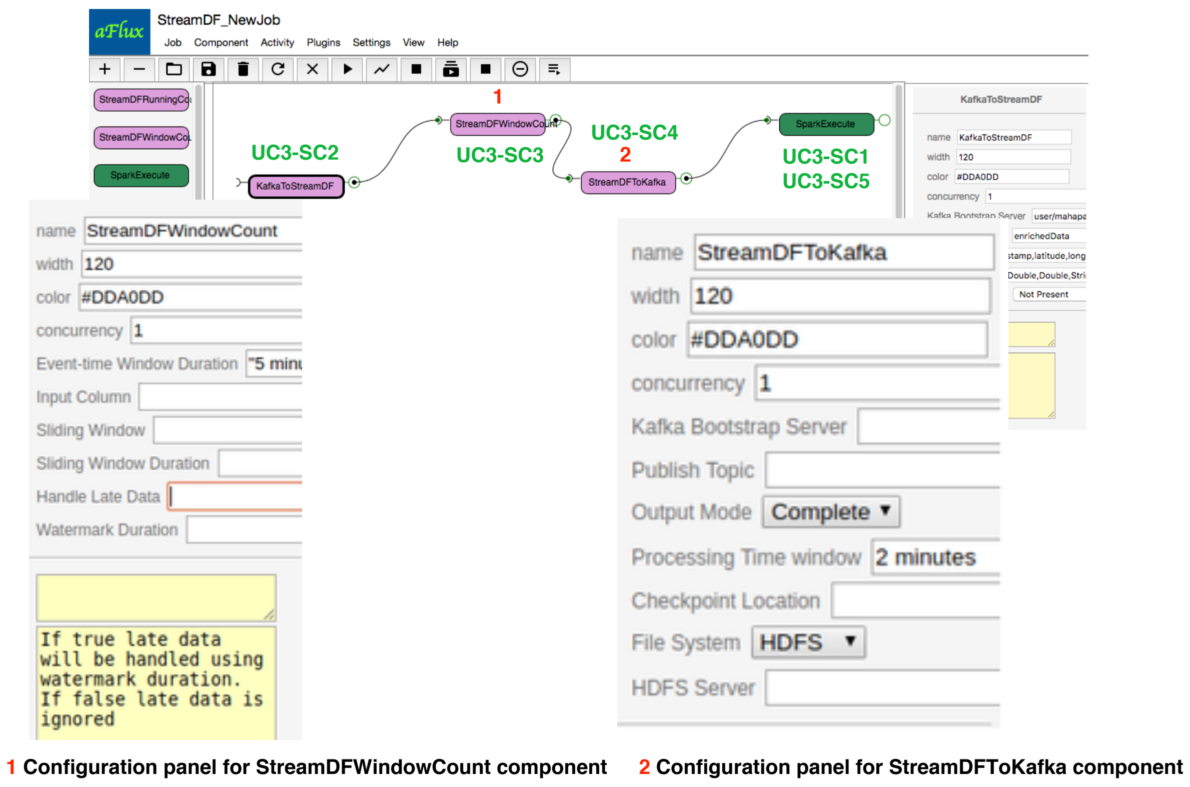
5.9.3.2 Approach: Graphical Programming via aFlux

Figure 5.23 illustrates an aFlux flow for performing streaming aggregations on real-time datasets. The components used are: (i) the 'KafkaToStreamDF' 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 Section 5.4, (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 is in correspondence to 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 a 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



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

Figure 5.23: Spark flow in aFlux for streaming aggregations

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 creation and termination of the application context.

The action component used in all the three use cases of graphical Spark programming have no output port functionality but are connected to the 'Spark Execute' component to mark the end of the flow. Figure A.5 shows the validation of the flow created for use case 3. Figures A.6, A.7 and A.8 illustrate the steps of the internal model generation. The complete Spark driver program generated from the user flow is shown in Listing A.10 and its generated property file is shown in Listing A.11.

Chapter 7 compares and discusses the flow-based Spark programming concepts with existing solutions offering similar high-level programming for Spark.

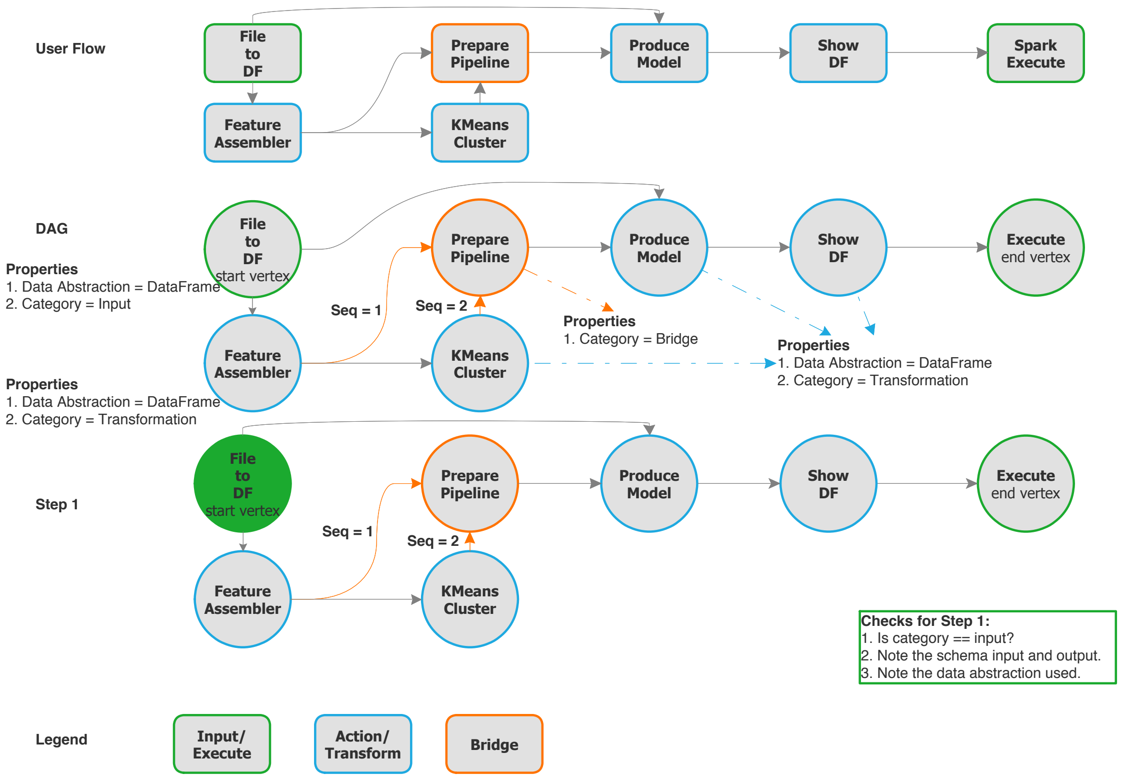


Figure 5.24: Validation of use case 1 Spark flow: producing a machine learning model: Steps. (contd. on Figure 5.25)

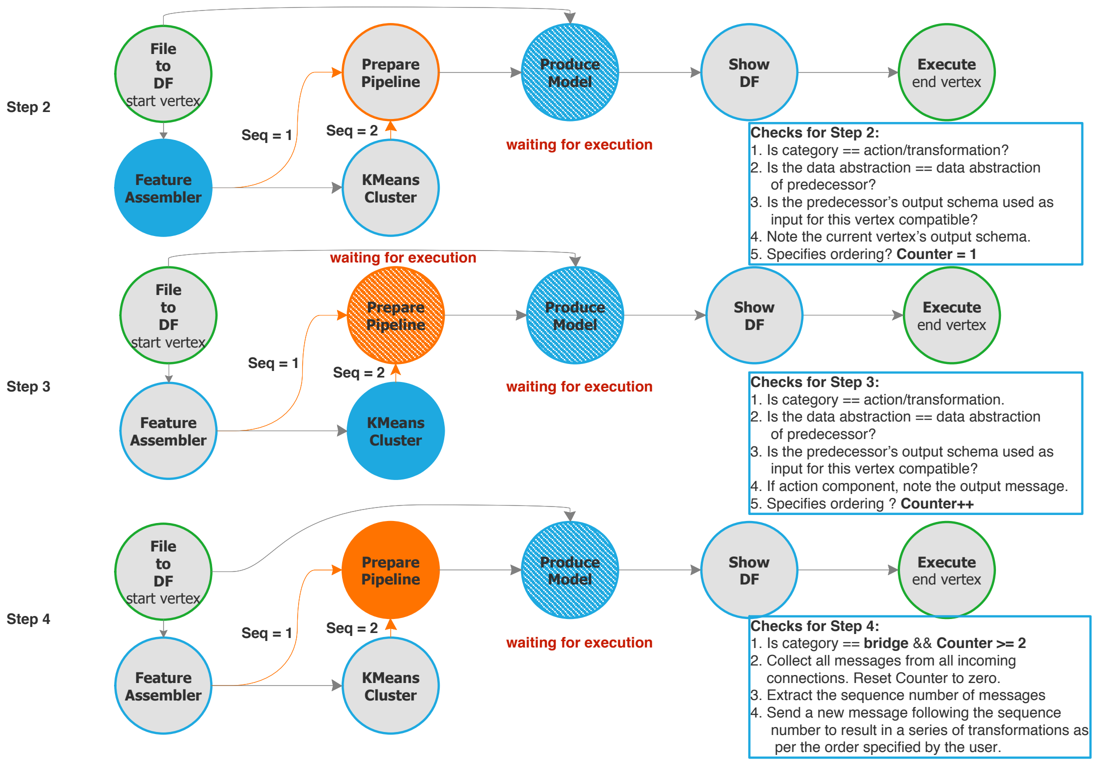
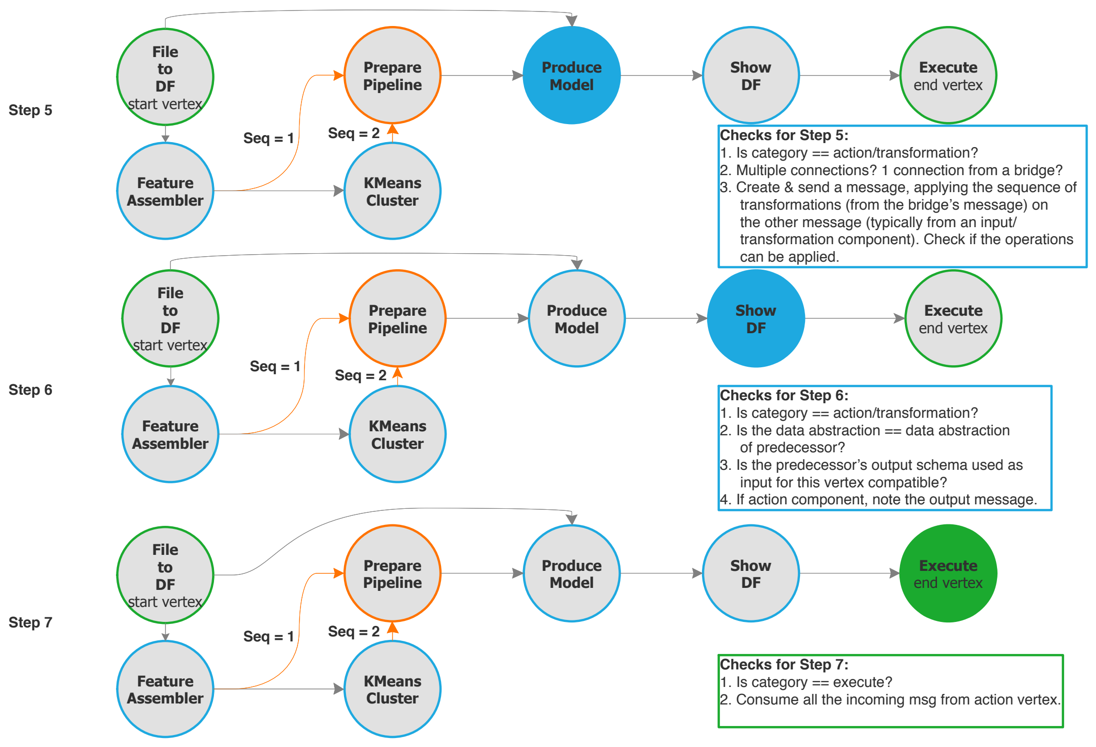

Step 2
File to DF start vertex
Feature Assembler
Prepare Pipeline
KMeans Cluster
Produce Model
Show DF
Execute end vertex
Seq = 1
Seq = 2
waiting for execution
Checks for Step 2:
1. Is category == action/transformation?
2. Is the data abstraction == data abstraction of predecessor?
3. Is the predecessor's output schema used as input for this vertex compatible?
4. Note the current vertex's output schema.
5. Specifies ordering? Counter = 1
Step 3
waiting for execution
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.
5. Specifies ordering ? Counter++
Step 4
Checks for Step 4:
1. Is category == bridge && Counter >= 2
2. Collect all messages from all incoming connections. Reset Counter to zero.
3. Extract the sequence number of messages
4. Send a new message following the sequence number to result in a series of transformations as per the order specified by the user.

Figure 5.25: Validation of use case 1 Spark flow: producing a machine learning model: Steps. (contd. from Figure 5.24, contd. on Figure 5.26)

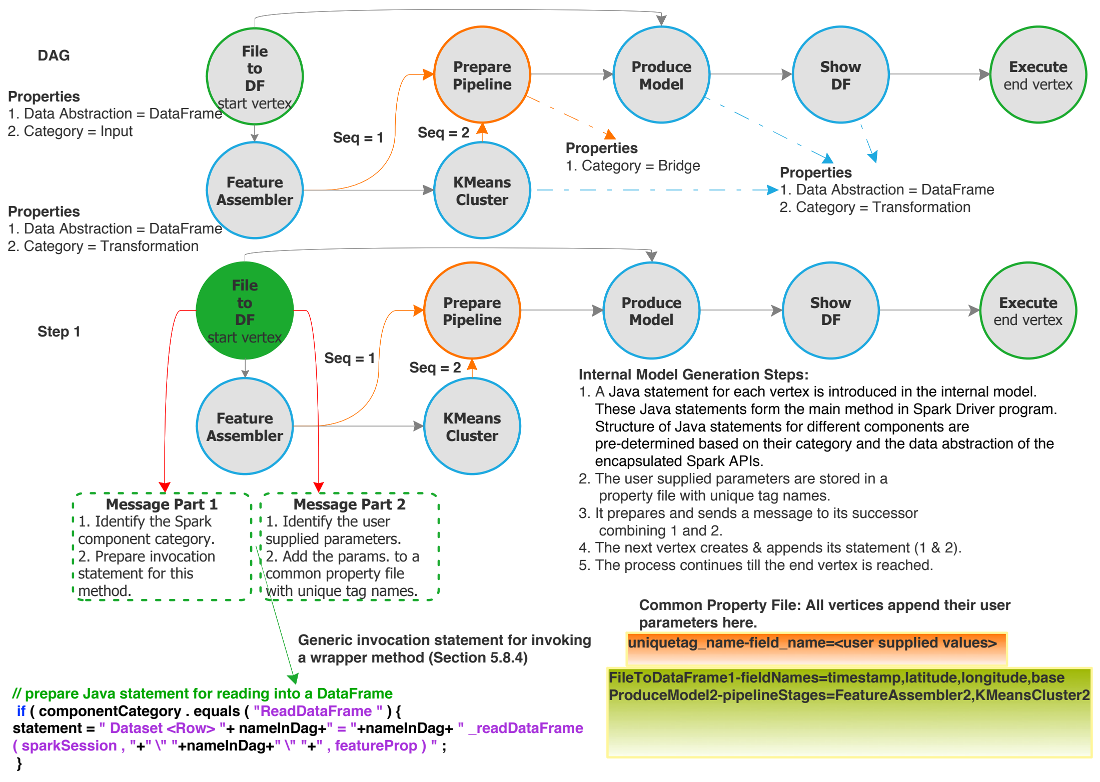


Checks for Step 5:
1. Is category == action/transformation?
2. Multiple connections? 1 connection from a bridge?
3. Create & send a message, applying the sequence of transformations (from the bridge's message) on the other message (typically from an input/ transformation component). Check if the operations can be applied.

Checks for Step 6:
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 7:
1. Is category == execute?
2. Consume all the incoming msg from action vertex.

Figure 5.26: Validation of use case 1 Spark flow: producing a machine learning model: Steps. (contd. from Figure 5.25)



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

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

Properties
1. Category = Bridge

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

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.

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.

Generic invocation statement for invoking a wrapper method (Section 5.8.4)

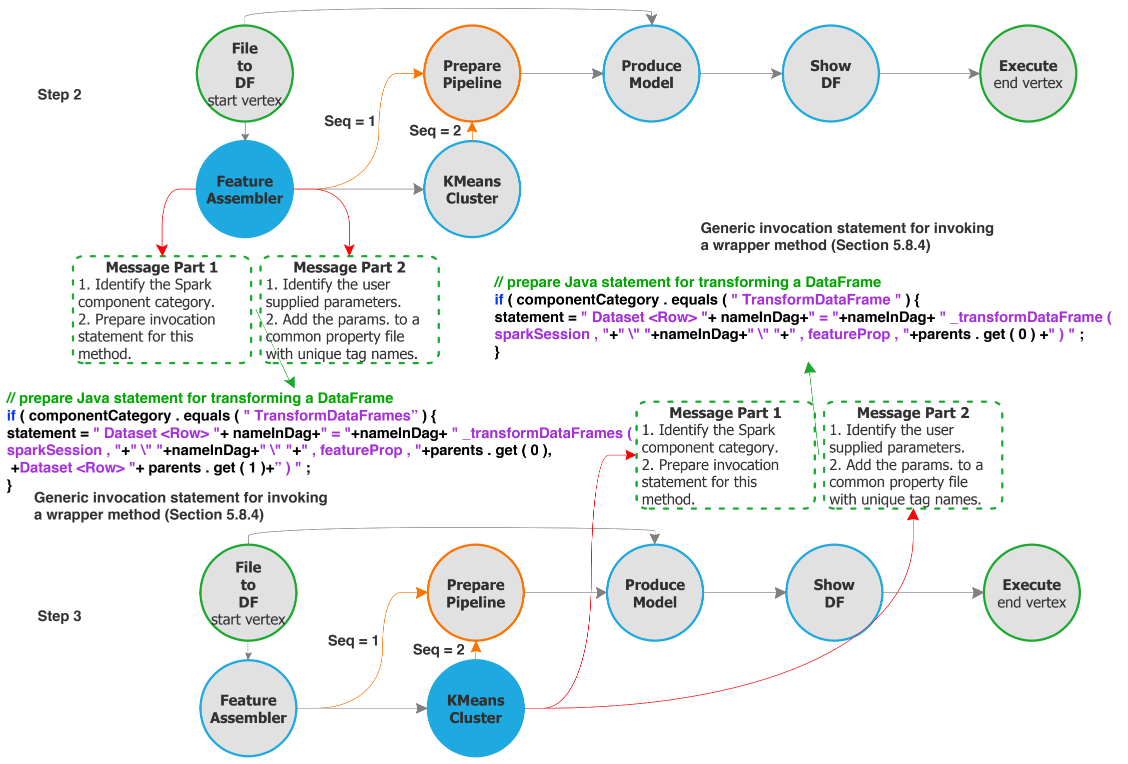
```
// prepare Java statement for reading into a DataFrame
 if ( componentCategory . equals ( "ReadDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _readDataFrame
( sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp ) " ;
 }
```

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

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

```
FileToDataFrame1-fieldNames=timestamp,latitude,longitude,base
ProduceModel2-pipelineStages=FeatureAssembler2,KMeansCluster2
```

Figure 5.27: Internal model representation of use case 1 Spark flow: Steps. (contd. on Figure 5.28)



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.

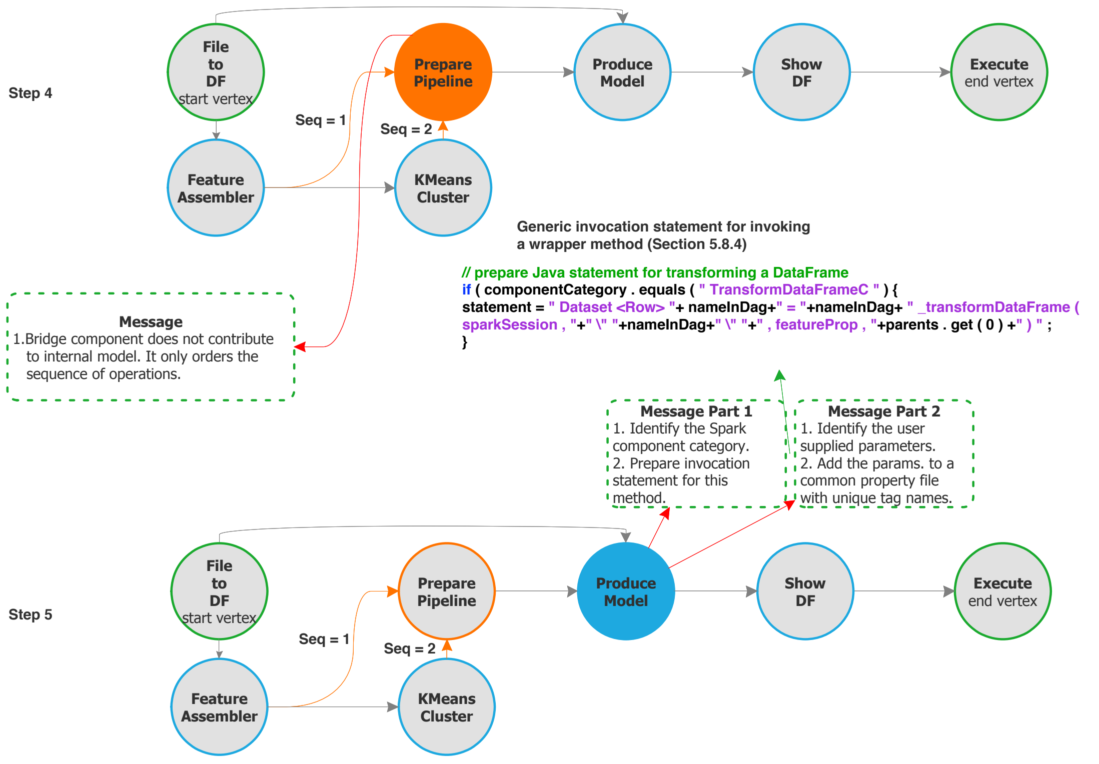
```
// prepare Java statement for transforming a DataFrame
if ( componentCategory . equals ( " TransformDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrame (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
```

```
// prepare Java statement for transforming a DataFrame
if ( componentCategory . equals ( " TransformDataFrames” ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrames (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ),
 +Dataset <Row> "+ parents . get ( 1 )+” ) " ;
}
```

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.

Figure 5.28: Internal model representation of use case 1 Spark flow: Steps. (contd. from Figure 5.27, contd. on Figure 5.29)



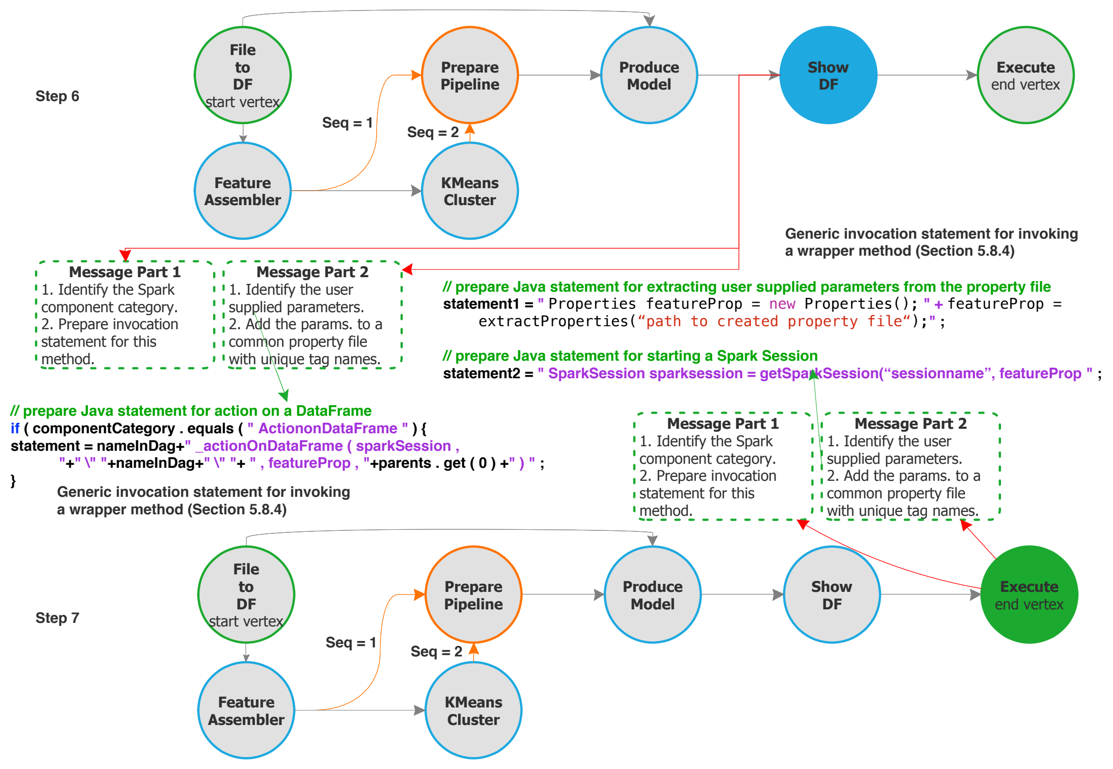
Message
1.Bridge component does not contribute to internal model. It only orders the sequence of operations.

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

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.

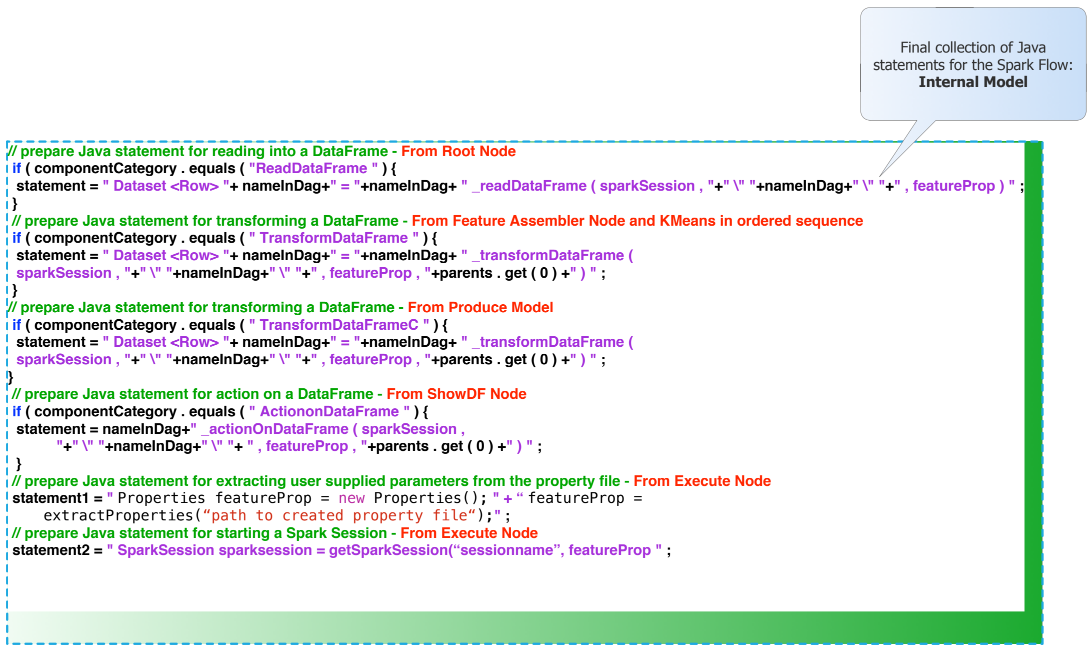
Figure 5.29: Internal model representation of use case 1 Spark flow: Steps. (contd. from Figure 5.28, contd. on Figure 5.30)


Step 6
File to DF start vertex
Feature Assembler
Seq = 1
Prepare Pipeline
Seq = 2
KMeans Cluster
Produce Model
Show DF
Execute end vertex
Generic invocation statement for invoking a wrapper method (Section 5.8.4)
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.
Generic invocation statement for invoking a wrapper method (Section 5.8.4)
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.
Step 7
File to DF start vertex
Feature Assembler
Seq = 1
Prepare Pipeline
Seq = 2
KMeans Cluster
Produce Model
Show DF
Execute end vertex

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

Figure 5.30: Internal model representation of use case 1 Spark flow: Steps. (contd. from Figure 5.29, contd. on Figure 5.31)


Final collection of Java statements for the Spark Flow: Internal Model

```
// 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 Feature Assembler Node and KMeans in ordered sequence
if ( componentCategory . equals ( " TransformDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrame (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
// prepare Java statement for transforming a DataFrame - From Produce Model
if ( componentCategory . equals ( " TransformDataFrameC " ) {
statement = " Dataset <Row> "+ nameInDag+ " = "+nameInDag+ " _transformDataFrame (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
// prepare Java statement for action on a DataFrame - From ShowDF 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 " ;
```

Figure 5.31: Internal model representation of use case 1 Spark flow: Steps. (contd. from Figure 5.30)

6 Graphical Flow-based Flink Programming

"Controlling complexity is the essence of computer programming."

— Brian Kernigan

IoT data typically comes in the form of data streams that often need to be processed under latency requirements to obtain insights in a timely fashion. Examples include traffic monitoring and control in a smart city; traffic data from different sources (e.g. cars, induction loop detectors, cameras) need to be combined in order to take traffic control decisions (e.g. setting speed limits, opening extra lanes in highways). The more sensors and capabilities, the more data streams require processing. Specialised stream-processing platforms, such as Apache Flink, Spark Streaming and Kafka Streams, are being used to address the challenge of processing vast amounts of data (also called Big Data), that come in as streams, in a timely, cost-efficient and trustworthy manner.

The problem with these stream platforms is that they are difficult to both set-up and write applications for. The current practice relies on human expertise and the skills of data engineers and analysts, who can deploy Big Data stream platforms in clusters, manage their life-cycle and write data analytics applications in general-purpose high-level languages such as Java, Scala and Python. Although many platforms, including Flink and Spark, provide SQL-like programming interfaces to simplify data manipulation and analysis, the barrier is still high for non-programmers.

To counter this challenge, we use the graphical programming approach of Spark discussed in Chapter 5 to support Flink programming via flow-based programming paradigm. To provide a technical underpinning for our proposal and evaluate its feasibility, we have validated the approach in aFlux to support the specification of streaming data pipelines for Flink, one of the most popular Big Data stream-processing platforms. The main challenge is the presence of diverse programming abstractions and APIs to develop Flink applications which are difficult to model in flow-based programming paradigm. In this chapter, we address the following :

1. A thorough analysis of the Flink framework to select the most suitable programming abstractions and APIs for flow-based Flink programming, i.e. *not supporting APIs requiring user defined data transformation functions or supporting code snippets during flow creation to interact with target framework internals.* Modelling the selected APIs as modular components.

2. Extending the generic approach to parse a graphical Spark flow discussed in Chap-

ter 5 to support flow-based Flink programming, i.e. parse a flow created using modular Flink components with support for early-stage validation and automatic code generation of Flink driver program.

6.1 Structure

This chapter is structured in the following way:

- Section 6.2 discusses the various programming abstractions supported by Flink to develop applications with and also explores the APIs supported in Flink. Here we outline the selected programming abstraction, data abstraction and APIs for the thesis work.
- Section 6.3 highlights the design decisions taken to support graphical flow-based programming of Flink.
- Section 6.4 describes the conceptual approach.
- Section 6.5 discusses 'FlinkFlo', a library consisting of modular composable components. The components bundle a set of Flink APIs which are executed in a specific order to perform one data analytic operation. This uses only a subset of Flink APIs which are compatible with the flow-based programming paradigm.
- Section 6.6 deals with the prototyping of the conceptual approach in aFlux.
- Section 6.7 discusses the evaluation of the approach.

The chapter concludes by enumerating the research contributions made to the thesis work. Appendix B is attached to this chapter and contains some code listings discussed here.

6.2 Flink Programming and Data Abstractions

Programming Abstractions The general introduction to Flink Ecosystem has been covered in Section 2.7. In this section, we briefly describe the APIs of Flink used to develop applications. Flink offers four level of *'programming abstractions'* to develop stream or batch applications as listed below:

1. low-level building blocks using streams, state and time.
2. core APIs based on DataSet/DataStream API.
3. declarative SQL type, i.e. Table API.
4. high-level SQL based operations.

The characteristics, features and advantages of all these levels of *programming abstractions* have been discussed in Section 2.7.

Structure of a Flink program The basic building blocks of a Flink program consists of streams and transformations. A stream is defined as an infinite flow of data while a transformation is an operation which takes one or more streams as input and produces one or more streams as output. Listing 6.1 shows a basic Flink program to calculate the frequency of words in a text collection. The program has two steps:

1. First, the texts are split into individual words.

2. Second, the words are grouped and counted.

When such a Flink program is executed, it is mapped into a streaming dataflow consisting of streams and transformation operators as shown in Figure 6.1. Each such dataflow starts with one or more source operators and ends with one or more sink operators. The dataflows typically resemble directed acyclic graphs (DAGs). Generally, there is a one-to-one correspondence between transformations in the program and the operators in the dataflow (vertices in the DAG).

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// Source
DataSet<String> text = env.readTextFile("/path/to/file");

// Transformations
DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in pairs (2-tuples) containing: (word,1)
        text.flatMap(new Tokenizer())
        // group by the tuple field "0" and sum up tuple field "1"
        .groupBy(0)
        .sum(1);

// Sink
counts.writeAsCsv(outputPath, "\n", " ");
```

Listing 6.1: Structure of a Flink program

Flink programs are typically parallel and distributed. During execution, a stream has one or more stream partitions and each operator has one or more operator subtasks. The operator subtasks are completely independent of each other and execute in separate threads or even separate machines. The number of operator subtask is the parallelism of a particular operator. It is interesting to state here that Flink executes batch programs as a special case of streaming programs where the streams are bounded. Hence, all the streaming concepts discussed above are equally applicable to batch programs developed in Flink. The windows supported in Flink and the concepts of time used in Flink have already been discussed in Section 2.7.

Data Abstractions Flink has two kinds of data-abstractions to represent and manipulate data. They are:

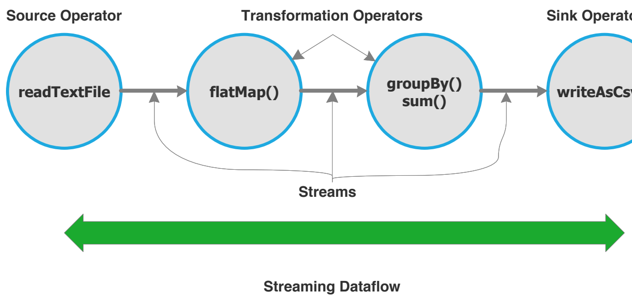


Figure 6.1: Mapping of a Flink program to a streaming dataflow during execution which resembles a typical DAG, as in [16]

DataSet It is an immutable collection of finite set of data. It is used to create batch programs in Flink and can be created and transformed by DataSet APIs.

DataStream It is an immutable collection of infinite set of data. It is used to create streaming programs in Flink and can be created and transformed by DataStream APIs.

Flink programs are regular programs that implement transformations on distributed collections created from reading files, Kafka topics etc. Results are made available via sinks. Depending on the data-source, i.e. batch or stream either DataSet or DataStream APIs from the core API level of programming abstraction can be used to write the transformations typically following the sequence of reading datasets, applying transformations and saving the results of the transformations. The Flink programs are executed lazily, i.e. the transformations in the main method of the Flink program are stored in the program plan and materialised when they are explicitly triggered. Section 7.1.2 discusses the lazy evaluation of Flink in detail and compares it with the lazy evaluation of Spark.

Selection of Programming & Data Abstractions From the available programming abstractions of Flink,programming via core APIs operating on Data DataSet and DataStream data abstractions, have been selected for the thesis work. The conceptual approach described in Section 6.4 supports only the transformations accessible via core APIs and using the aforementioned data abstractions. Any transformation making use of low-level stateful stream processing of Flink or programming via the SQL/Table API is not supported. It is easy to represent core APIs based on DataSet and DataStream APIs as modular components that can be wired together in a flow-based programming paradigm. Finally, the different input parameters required by an API can be specified by the user from the front-end. Core APIs based on these aforementioned data abstractions prevent the usage of low-level building blocks of Flink programming and provide data transformation for each operation, i.e. user need not write custom functions from scratch for data transformation.

6.3 Design Decisions

In order to support Flink pipelines in mashup tools, we needed to decide on the (i) required abstraction level, (ii) formulation of modular components from Flink APIs and (iii) a way to support validation of graphical flows to yield a compilable and runnable Flink program.

Accordingly, from the different abstraction levels, we decided to select the core API abstraction levels for supporting Flink pipelines in graphical mashup tools, as these APIs are easy to represent in a flow-based programming model. They prevent the need for user-defined functions to bring about data transformation and provide predictable input and output types for each operation—the tool can then focus on validating the associated schema changes. Moreover, it is easy to represent DataStream and DataSet APIs as graphical components that can be wired together. Finally, the different input parameters required by an API can be specified by the user from the front-end.

We follow the lazy execution model while composing a Flink pipeline graphically, i.e. when a user connects different components, we do not automatically generate Flink code but instead take a note of the structure and capture it via a DAG, simultaneously checking for structural validity of the flow. When the flow is marked as complete, the runnable Flink code is generated. Lastly, we impose structural restrictions on the graphical flow which can be composed by the user, i.e. it must begin with a data-source component, followed by a set of transformation components and finally ending with a data-sink component, in accordance with the anatomy of a Flink program. Only when the flow is marked complete does the actual code generation begins.

6.4 Conceptual Approach

The conceptual approach for designing flow-based Flink programs consists of: (i) A model to enable the graphical creation of programs for stream analytics, in other words, to automatically translate items specified via a GUI to runnable source code, known as the *Translation & Code Generation Model*, and (ii) a model to continuously assess the end-user flow composition for structural validity and provide feedback to ensure that the final graphical flow yields a compilable source code, known as the *Validation Model*. Figure 6.2 gives a high-level overview of the conceptual approach used to achieve such a purpose. This conceptual approach is based on the design decisions discussed in Section 6.3.

Since the main idea is to support flow-based Flink programming for Stream analytics, we restrict the scope of the translator to the DataStream APIs from the core-level API abstractions. Nevertheless, the DataSet APIs used for batch analytics are fully compatible with the approach as both the data abstractions of Flink are fundamentally the same.

In accordance to the anatomy of a Flink program, we have built *'SmartSantander Data'* as the data-source component, an *'Output Result'* supporting writing operation to Kafka, CSV and plain text as data-sink component. Map, filter and window operations are the supported transformation components. Accordingly, we built the *'GPS Filter'* component to specify filter operations, the *'select'* component to support map operations and a *'Window'* as well as *'WindowOperation'* to specify windows on data streams. We also support the Flink CEP library via the following components: *'CEP Begin'*, *'CEP End'*, *'CEP Add condition'* and *'CEP New condition'*. The CEP library is used to detect patterns in data streams. We also have two additional components, namely *'Begin Job'* and *'End Job'*, to mark the start and end of a Flink pipeline. The translator & code generation model have been designed to work within this scope of selection. We define all potential positional rules of these components, i.e. which component can before a specific component in the flow, in a flow and the validation model works within this scope.

The conceptual approach described for flow-based Flink programming is exactly the same as the Spark approach described in Section 5.5 and Section 5.6 of Chapter 5. The only difference is that the modular components here represent a set of Flink APIs and work on data abstractions specific to Flink.

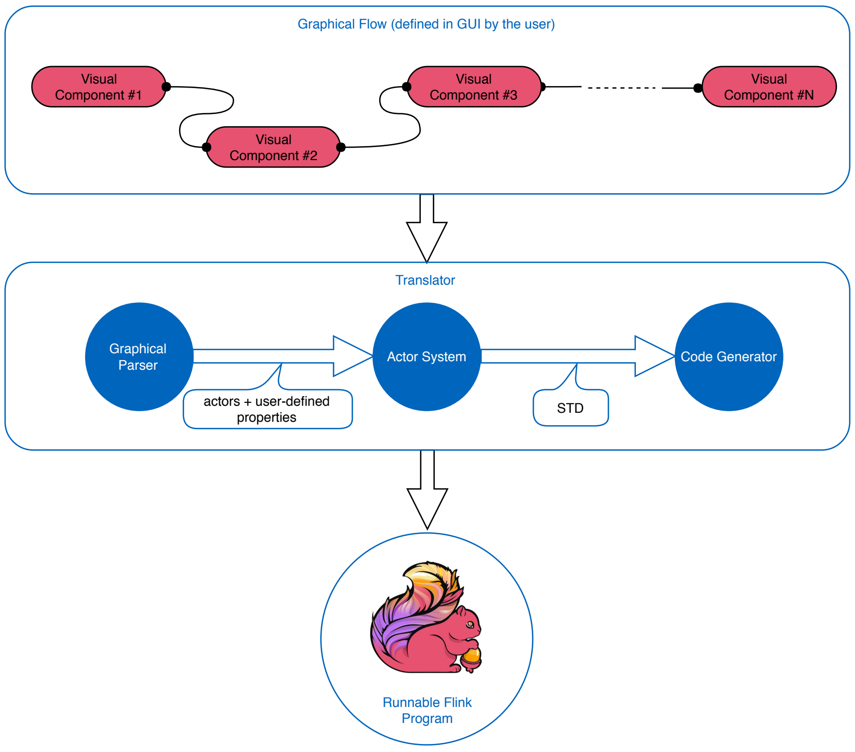


Figure 6.2: Conceptual approach for translation and code generation

6.4.1 Translation & Code Generation

The aim of the translation & code generation model is to provide a way to translate a graphical flow defined by the end user into a native Flink program. This model behaves as follows: (i) First, the end user defines a flow by connecting a set of *graphical components* in a flow-like structure. It represents a certain Flink functionality (i.e. abstraction of a set of Flink APIs invoked in a certain order to perform one data analytic operation) and has a set of properties that the user may configure according to their needs. (ii) Then, a *translator* acquires the aggregated information of the user-defined flow, which contains (a) the set of graphical components that compose the flow, (b) the way in which they are connected and (c) the properties that users have configured for each component.

The *translator* has three basic components: a *graphical parser*, an *actor system* and a *code generator*. It takes as input the aggregated information of the user-defined graphical flow (i.e. graphical components, the flow structure and the user-defined properties) and its output is a packaged and runnable Flink job. The graphical parser takes the aforementioned aggregated information and processes it, creating an internal model and instantiates the set of actors corresponding to the flow. The actor system is the execution environment of actors, which contains the business logic of the translator. Actors are taken from the output of the graphical parser. The actor model abstraction makes each actor independent, and the only way to interact with the rest is by means of exchanging messages. Actors communicate using a data structure that has been explicitly defined for making the translation, using a tree-like structure that makes appending new nodes extremely easy. In this model, the data structure is referred to as STDS (Specific Tree-Like Data Structure). As previously stated, each actor corresponds to a specific Flink functionality and, in turn, to the standalone implementation method of that specific functionality. It adds a generic method-invocation statement as a message response to the next connected actor. The method-invocation statement also passes the user parameters and the output from its preceding node as input to the standalone implementation method of Flink-functionality APIs. The next actor receives this message and appends its corresponding method-invocation statement and so forth which is the same process used in case of a Spark, i.e. internal model generation in Section 5.6.3 of Chapter 5.

6.4.2 Validation

The translation model allows the translation of graphical flows into source code. However, some graphical flows may result in source code that either cannot be compiled or yields runtime errors. We have provided support on tool-level for handling the type of errors that occur because of data dependencies in a flow, during its specification. If one of the data dependency rules is violated when the user connects or disconnects a component in a flow, visual feedback is provided, which helps avoid problems early on. Such compositional rules must be specified for every modular Flink component, according to the following pattern:

Structure of a compositional rule

$$\underbrace{\text{Component A}}_{\substack{\text{main}\\ \text{visual component}}} \; \underbrace{\left|\begin{matrix}\text{should}\\ \text{must}\end{matrix}\right|}_{\text{isMandatory}} \; \text{come} \; \underbrace{\text{(immediately)}}_{\text{isConsecutive}} \; \underbrace{\left|\begin{matrix}\text{before}\\ \text{after}\end{matrix}\right|}_{\text{isPrecedent}} \; \underbrace{\text{Component B}}_{\substack{\text{argument}\\ \text{visual component}}}$$

For example, the following rules can be specified:

- *'Window' component must come immediately after 'Select' component*
- *'End Job' component should come after 'Load data' component*

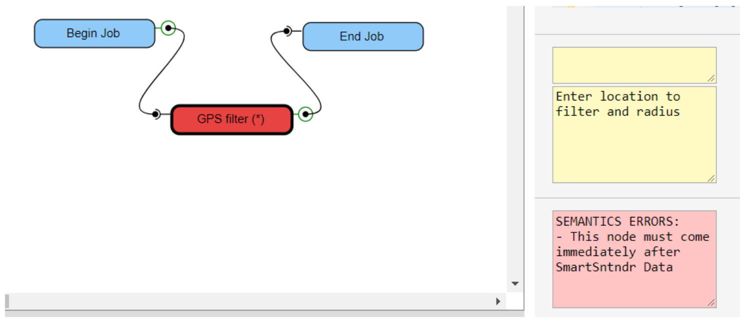
Each compositional rule is defined between two components having a certain number of flags, i.e. 'component A' and 'component B' where 'component A' is the main component of the compositional rule and 'component B' is given as parameter. The *'isMandatory'* flag when set to true means that 'Component A MUST (...) Component B' and when set to false means 'Component A SHOULD (...) Component B'. The *'isPrecedent'* flag indicates which component should come before the other. When set to true, it implies a condition of precedence, i.e. 'Component A (...) AFTER Component B'. On the contrary, when set to false it implies a condition of non-precedence, i.e. 'Component A (...) BEFORE Component B'. The *'isConsecutive'* flag governs the strict positional hierarchy of the components. When set to true it implies a strict contiguity, i.e. 'Component A (...) IMMEDIATELY (...) Component B'. On the contrary, when set to false, the contiguity is non-strict, i.e. 'Component A (...) Component B'.

On the front-end, when a user connects two components, it is considered a state-change. With every state-change, the entire flow is captured from the front-end and subjected to the validation process. Basically, the flow is captured in the form of a tree; the next step is to check whether the components are compatible to accept the input received from their preceding components, whether two immediate connections are legal and whether the tested component's positional rules permit it to be used after its immediate predecessor. Algorithm 1 summarises the flow validation steps. During the check, if an error is found with any one component of the flow, the user is alerted with the appropriate reasons and the component is highlighted as shown in Figure 6.3.

Algorithm 1: Validation of Flink flows

```
foreach flow to be validated do
    order the list of element as they appear in the flow;
    foreach element in the orderedList do
        get the set of compositional rules for it;
        instantiate a new result;
        foreach rule in compositional rules do
            foreach element in the orderedList do
                if rule is not met then
                    result.add(rule);
                end
            end
        end
        if result is empty then
            clear error information from element;
        else
            add error information to element;
        end
    end
end
```

Finally, after completion of the flow-validation, the code generator takes the STDs as



Enter location to filter and radius

SEMANTICS ERRORS:
- This node must come immediately after SmartSntndr Data

Figure 6.3: Flink flow validation: user feedback

input. It has internal mapping to translate parametrised statements (generic invocation statements to invoke the standalone implementation of Flink APIs) into real Flink source code statements. This entity combines the parametrised statement with this mapping and the user-defined properties, and then generates the final source code. The compiling process also takes place here. The code generator output is a packaged, running Flink job that can be deployed in an instance of Flink. The approach is very similar to the approach used for programming Spark from mashup tools (Section 5.6.4).

6.5 FlinkFlo: A Subset of Flink APIs Bundled as Modular Components

Flink offers many different libraries in its ecosystem accessible either via low-level stateful stream processing approach, programming via Table APIs, data transformation via SQL or invocation via core APIs working on different data abstractions like DataSet (for batch analytics) and DataStream (for stream analytics). The manual development of Flink application involves interaction with these elements. To support development of Flink applications from graphical flow-based programming paradigm abstraction is essential. With the approach described in the chapter, a subset from the Flink ecosystem making use of the core APIs programming abstraction operating on *DataSet* and *DataStream* data abstractions have been selected which are compatible with the flow-based programming paradigm.

'*FlinkFlo*' is a library consisting of modular (Section 5.2) composable (Section 5.6.1.3) components. The components in *FlinkFlo* bundle a set of Flink APIs from the selected subset of Flink 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 a 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 Section 5.6.1.3. The components of FlinkFlo can then be expressed as compositional units in a flow-based programming graphical tool, for instance they have been expressed as actors in aFlux. Figure 6.4 illustrates the elements and characteristics of FlinkFlo.

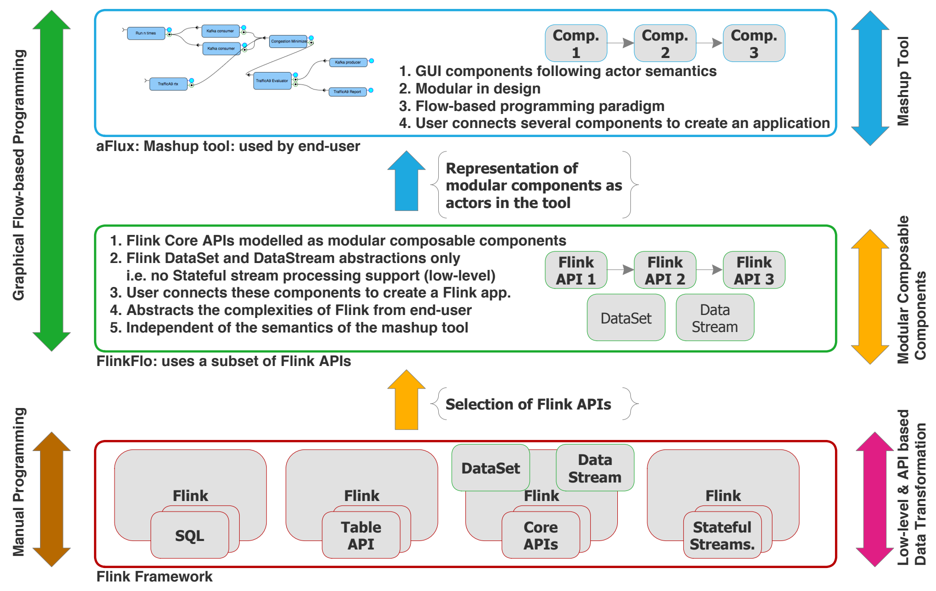

Graphical Flow-based Programming
Comp. 1
Comp. 2
Comp. 3
1. GUI components following actor semantics
2. Modular in design
3. Flow-based programming paradigm
4. User connects several components to create an application
aFlux: Mashup tool: used by end-user
Mashup Tool
Representation of modular components as actors in the tool
1. Flink Core APIs modelled as modular composable components
2. Flink DataSet and DataStream abstractions only i.e. no Stateful stream processing support (low-level)
3. User connects these components to create a Flink app.
4. Abstracts the complexities of Flink from end-user
5. Independent of the semantics of the mashup tool
Flink API 1
Flink API 2
Flink API 3
DataSet
Data Stream
FlinkFlo: uses a subset of Flink APIs
Modular Composable Components
Selection of Flink APIs
Manual Programming
Flink SQL
Flink Table API
DataSet
Data Stream
Flink Core APIs
Flink Stateful Streams.
Flink Framework
Low-level & API based Data Transformation

Figure 6.4: FlinkFlo: A library of modular composable components using a subset of Flink APIs

6.5.1 Extensibility: Inclusion of new/forthcoming Flink APIs

A new/forthcoming Flink API can be modelled as a FlinkFlo component to be used in a flow-based tool like aFlux and auto-generate a runnable Flink program using the approach described in Section 6.4, *iff*:

- The new Flink transformation is *accessible via core API programming abstraction* (Section 6.2).
- It *must use either the DataSet or the DataStream data abstraction*.

For creating a modular component and its prototyping in a mashup tool entails similar APIs analysis, development of wrapper methods and component development by the developer following steps explained in Section 5.7.1.

6.6 Implementation

In this section, we describe the various components prototyped specific to the evaluation use cases (discussed in Section 6.7) in order to realise flow-based Flink programming.

6.6.1 The SmartSantander Data Connector for Flink

Figure 2.11 illustrates the typical structure of a Flink flow which begins by reading datasets from a data source, followed by a series of transformations and ends with a data sink. By default, Flink supports the following kinds of data sources:

File-based data sources typically monitor a file or a specific directory and load from it into the Flink environment for processing.

Socket-based data sources connect to a hostname via a specific port and read datasets into the Flink environment for processing.

Collection-based data sources read datasets from Java Collection classes.

The built-in data sources are quite simple and do not suffice for real world use-cases. Hence, Flink provides an option to extend the basic data sources to create more complicated data sources to read from other third party systems/platforms. For instance, developers have built data sources to read data directly from Twitter, a Kafka broker system or even platforms like Wikipedia [166]. Hence, to fit the use-cases as explained in Section 6.7, the first objective was to develop a Flink connector to read data from the SmartSantander APIs [135] and create a data stream out of it for analytics. The sensors deployed in the city of Santander post their data in as soon as it is available to a back-end. The live data can be accessed via REST APIs provided by the back-end system.

The entry point of Flink connectors to the Flink engine is the 'SourceFunction' class. Any class that extends class one can be used as a data source when creating Flink jobs. Hence, a new class extending'SourceFunction' was created to continuously stream data from the SmartSantander REST APIs so that other Flink transformation APIs can be applied. From all the available datasets, three main data collections have been considered for the implementation, i.e. the traffic dataset, environment dataset and air quality dataset. A dataset typically in JSON format after being fetched via a REST API is deserialized, duplicates are removed, (if any) and is available as continuous streams within the Flink run-time environment for analytics.

6.6.2 Flink Components for aFlux

To illustrate the workings of the approach and specific to the use-cases as discussed in Section 6.7 components supporting Flink streaming have been implemented [119] as shown in Figure 6.5. All the components are actors internally and typically follow the same structural definitions and patterns as the components for Spark discussed in Section 5.6.1 of Chapter 5. Typically, the components fall in four categories and correspond to the anatomy of a Flink program as enumerated in Section 2.7.2.

The approach used to model a Flink pipeline relies on three aspects, i.e. load data from data source, transform data and finally publish the result via a data sink. This is also the preliminary form of semantic validation, i.e. deciding if the positional hierarchy of a component is allowed or not. The user-flow is parsed and expressed as an abstract syntax

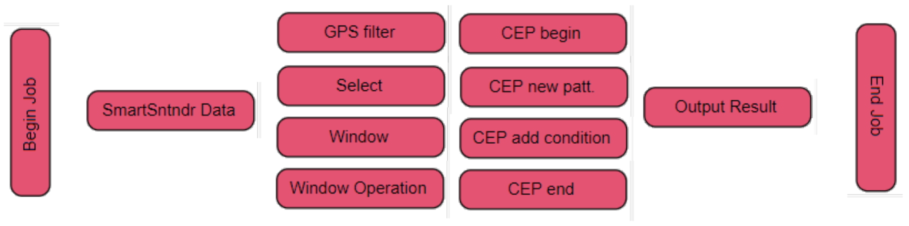


Figure 6.5: Flink components for aFlux

tree which is passed as an input to the code generator. Each node in the tree maps to a standalone implementation of the Flink Core APIs. The code generator generates code for sequences like, opening and closing a Flink session, and for the nodes in the abstract syntax tree it wires the standalone implementation of the APIs, while passing the user parameters and the output from the preceding node as input. The result is a runnable Flink program, compiled, packaged and deployed on a cluster.

6.7 Evaluation

The implemented approach has been evaluated for its ease in graphically creating Flink jobs from aFlux by abstracting the code-generation from the end-user and automatic Flink driver program generation. For evaluation purposes, we have used live data from the city of Santander, Spain, which is offered as open data behind public APIs [135].

Scenario In this smart city use-case, the user is an analyst of Santander City Hall, who need not have programming skills. The user only needs to know how to use aFlux from the end-user perspective (e.g. drag and drop mashup components) and have some very basic knowledge of what Flink can do from a functionality point of view rather than from a developer point of view. For example, the city hall analyst should know that changes in the city are measured in events and events can be processed in groups called windows. The user does not need to know any details about how to create a window in the Flink Java or Scala API, or the fact that generics need to be used when defining the window type of window. Two broad use-cases have been identified for evaluation purposes:

6.7.1 Use-Case 1: Real Time Data Analytics

The process of analysing stream data as it is generated involves gathering data from different sources of the city via APIs and processing them. The goal of this use-case is to gain insights about the city, that help decision makers take the appropriate calls. For instance, if the air quality is below the desired levels, the city hall would probably wonder whether it is a good decision to restrict traffic in the city. To prepare a report, an analyst could create a program to emit the air quality and traffic charge of a certain area and see if they are related (hence it is a good decision to restrict traffic) or not (hence restricting traffic will have no impact on the air quality). For stream analytics, a scenario was designed to correlate between temperature and air pollution in one area of the city. Additionally to check, if the conditions of the area in observation in comparison to other areas of the city affect this relation. For this four Flink flows in aFlux are created. Two flows are needed to analyse temperature data and two are needed for air quality (e.g. the level of carbon monoxide). Two flows are required for each dataset because one will include a 'GPS filter' to restrict the data to one particular area of the city (Figure 6.6) and the other flow would read data for the whole city (Figure 6.7).

```java
DataStream<Double> levelOfOzone = filteredAirQuality.map(new MapFunction<
    AirQualityObservation, Double>() {
  @Override
  public Double map(AirQualityObservation airQualityObservation) throws
    Exception {
    return Double.valueOf(airQualityObservation.getLevelOfOzone());
  }
});
```

Listing 6.2: Generated Java code to select multiple datasets for comparison

What has been evaluated? We describe certain aspects which become easier with graphical programming of Flink below:

Code abstraction Figure 6.6 and Figure 6.7 show how the analyst can easily get input from streaming sources by using a graphical data-source component, i.e. the 'SmartSntndr Data'. Adding a new source of data is as simple as changing a property in the 'SmartSntndr Data' component and aFlux auto-generates the Java code for it as shown in Listing 6.2.

Window types Tumbling windows were used in Figure 6.6 and Figure 6.7, but processing the data in a different type of window (e.g. using sliding windows) is as easy as changing the properties of the 'Window' mashup component (Figure 6.8) and aFlux generates the code as listed in Listing 6.3. In Java, the user would need to know that a sliding window takes an extra parameter, and that the window slide needs to be specified using Flink's Time class, in which a different method is invoked depending on the time units that they desire.

Ease of data filtering If the analyst were to filter datasets based on the area location that is being analysed, doing it from aFlux is changing properties of the 'GPS Filter' component and choosing the desired radius and aFlux would auto-generate the code to read the data from the 'SmartSntndr Data' component and filter it before passing to the next components in the flow as indicated in Listing B.1 (Appendix B). Filtering datasets via manual programming can be difficult as compared to the GUI-based selection and code auto-generation approach.

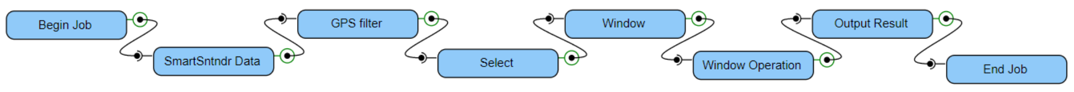


Figure 6.6: Stream analytics: Flink flow making use of GPS filter

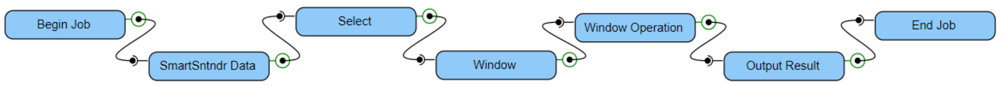
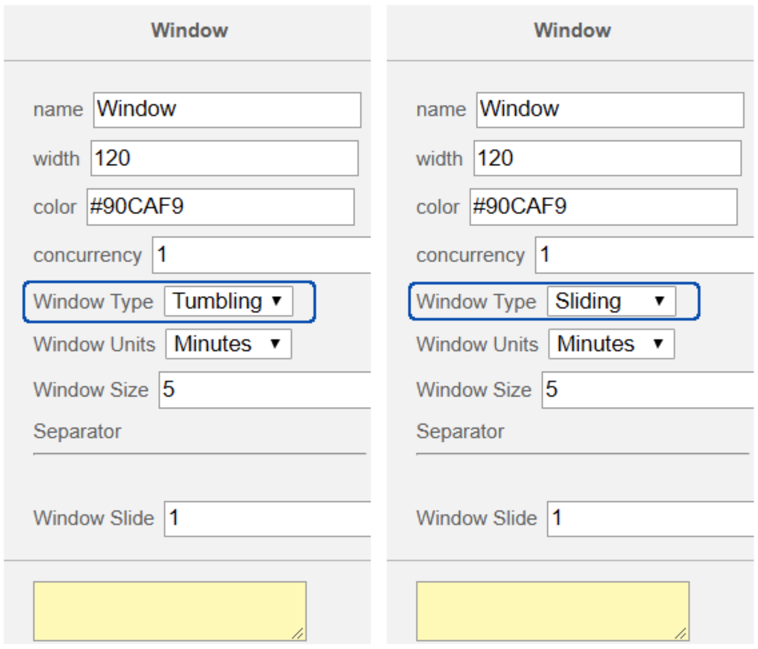


Figure 6.7: Stream analytics: Flink flow without GPS filter



Window	
name	Window
width	120
color	#90CAF9
concurrency	1
Window Type	Tumbling ▼
Window Units	Minutes ▼
Window Size	5
Separator	
Window Slide	1

Window	
name	Window
width	120
color	#90CAF9
concurrency	1
Window Type	Sliding ▼
Window Units	Minutes ▼
Window Size	5
Separator	
Window Slide	1

Figure 6.8: Tumbling vs sliding windows in aFlux

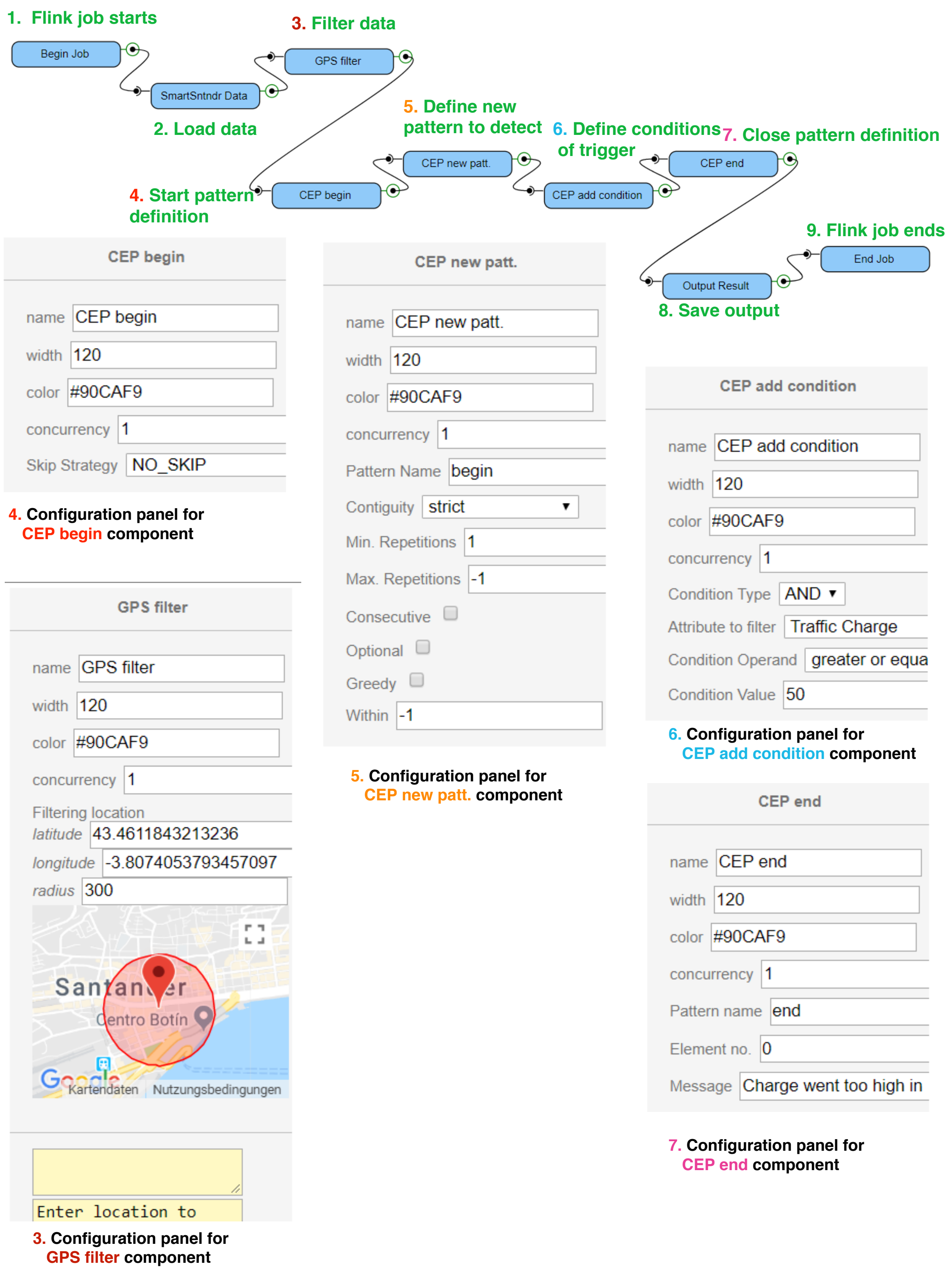
```
AllWindowedStream<Double, TimeWindow> tumblingWindowedTemperature = temperature
    .windowAll(TumblingEventTimeWindows.of(Time.minutes(5)));

AllWindowedStream<Double, TimeWindow> slidingWindowedTemperature = temperature.
    windowAll(SlidingEventTimeWindows.of(Time.minutes(5), Time.minutes(1)));
```

Listing 6.3: Auto-generated codes for tumbling and sliding sindows from aFlux

6.7.2 Use-Case 2: Detection of Complex Patterns

Detecting a pattern of events does not focus to gain insights from the data, but to have the system notify the City Hall whenever a specific pattern is detected. The analyst would only have to define the pattern to search and the type of notification to be sent in case of a match — and aFlux would take care of the details. For instance, the analyst could define a pattern to detect a progressive increment in the traffic concentration in a certain area of the city to indicate a probable accident situation. If this happens, the system would sent a notification to send a police car. Flink has the Complex Event Processing (CEP) [17] library which is used to detect patterns on an infinite data stream. In this case, the analyst makes use of GUI components in aFlux to specify patterns and trigger alert when such an event takes place. The idea is to design a flow so that it detects unusual traffic volume in a given area and whenever the volume of traffic increases by more that 50% of its normal value then an alert is triggered. It is also designed to have a second event detection if the traffic increases over 60% within 10 minutes from the previous alert and a third event to detect traffic volume amounting to 75% within the next 10 minutes from the previous alert. Figure 6.9 shows the flow developed in aFlux for this scenario. It basically makes use of the‘CEP new patt.’ and the ‘CEP add condition’ components to define patterns to detect on streams. Repeated addition of these components one after another allows the user to specify more than one pattern to detect. The automatic code generation is done by the graphical programming approach. Listing B.2 shows the code auto-generated for the CEP library of Flink.

Chapter 7 compares and discusses the flow-based Flink programming concepts with existing solutions offering similar high-level programming for Flink.



CEP begin

name	CEP begin
width	120
color	#90CAF9
concurrency	1
Skip Strategy	NO_SKIP

4. Configuration panel for CEP begin component

GPS filter

name	GPS filter
width	120
color	#90CAF9
concurrency	1
Filtering location	
latitude	43.4611843213236
longitude	-3.8074053793457097
radius	300

Enter location to

3. Configuration panel for GPS filter component

CEP new patt.

name	CEP new patt.
width	120
color	#90CAF9
concurrency	1
Pattern Name	begin
Contiguity	strict ▾
Min. Repetitions	1
Max. Repetitions	-1
Consecutive	☐
Optional	☐
Greedy	☐
Within	-1

5. Configuration panel for CEP new patt. component

CEP add condition

name	CEP add condition
width	120
color	#90CAF9
concurrency	1
Condition Type	AND ▾
Attribute to filter	Traffic Charge
Condition Operand	greater or equa
Condition Value	50

6. Configuration panel for CEP add condition component

CEP end

name	CEP end
width	120
color	#90CAF9
concurrency	1
Pattern name	end
Element no.	0
Message	Charge went too high in

7. Configuration panel for CEP end component

Figure 6.9: Flink flow in aFlux to detect events in streams

7 Discussion & Conclusion

"Computer Science is no more about computers than astronomy is about telescopes."

— Edsger W. Dijkstra

This chapter first compares the flow-based Big Data programming concepts developed in this thesis with other related works. Next, the conceptual approach used to support high-level programming of Big Data applications is discussed from perspectives of extensibility to other Big Data target frameworks and discusses the possibility of formulating a unified approach for programming Spark and Flink, i.e. making the approach target Big Data framework-agnostic. In this context, the abstraction-level supported over Spark and Flink thereby influencing the concepts within the approach which are framework specific as well as concepts which are framework independent are clearly outlined. Furthermore, subjective opinions and insights from the derivatives of the investigation done as part of the thesis work have been discussed. Finally, the chapter concludes by summarising the thesis contributions.

7.1 Discussion

The conceptual approach used for supporting high-level programming of Big Data applications for Spark and Flink in flow-based programming paradigm has been discussed from two different perspectives in this section.

7.1.1 Comparison with Existing Solutions

In this section, we compare the graphical flow-based programming concepts for Big Data with existing similar solutions. In Section 2.12, we have classified existing solutions into two categories. The category 2 solutions deal with tools which support high-level programming for Big Data applications. These tools are Lemonade, QryGraph, Nussknacker, Apache Zeppelin, Apache NiFi, Apache Beam and Microsoft Azure. Additionally, we have also considered QM-IConf from category 1 as it generates native Big Data program for Apache Storm.

To reiterate, the conceptual approach has two aspects to it: (i) Analysing the target frameworks and selecting data abstractions, APIs which are compatible with graphical

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	✓	✓	✓
Thesis work prototyped in aFlux	Flow-based GUI tool	Spark and Flink. **Extensible**	✓	✓	✓

Table 7.1: Comparison of high-level Big Data programming with existing solutions

flow-based programming paradigm, i.e. *not supporting APIs requiring user defined data transformation functions or supporting code-snippets during flow creation to interact with target framework internals.* Modelling the selected APIs as modular components. (ii) Representing the modular components in flow-based programming tools. Devising a generic approach to parse a graphical flow created using these components with support for early-stage validation and automatic code generation of Big Data programs for the specific target framework.

Table 7.1 summarises the comparison of the conceptual approach developed in the thesis with existing solution offering similar abstraction over Big Data application development. In particular, we compare and contrast with respect to 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 7.1 lists this criterion.* All tools except Apache Beam and Apache Zeppelin provide a graphical interface to design an application. Beam is a high-level unified programming model which has its own 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, Flink, Apex, MapReduce, IBM Streams etc. [15]. 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 developed in the thesis, 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 7.1 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 really a Big Data programming tool. It is used to design data-flow pipelines via flow-based programming paradigm. Nevertheless, it has special 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. But in order 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 a number of different execution environments with minimal changes [15]. 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. But when a user specifically needs 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 the thesis, prototyped in aFlux, currently supports Spark and Flink and is extensible to other target Big Data frameworks. For discussion on extensibility of the approach, please refer to Section 7.1.2.

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 7.1 lists this criterion.* All of the existing tools including the conceptual approach developed in the thesis provide 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 a number of Big Data frameworks while simultaneously introducing its own level of usage complexity. Similarly, Zeppelin does not provide a 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 7.1 lists this criterion.* Tools like Lemonade, QryGraph, Nussknacker, QM-IConf and the conceptual approach developed in the thesis 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 7.1 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 developed in the thesis generate target Big Data program from the user flow.

7.1.2 Extensibility of the Conceptual Approach to Additional Target Frameworks

From a user's perspective, designing a high-level flow to accomplish similar tasks via Spark or Flink the flow structure remains almost identical making use of comparable components. For instance, to create a program to calculate frequency of words in a text collection the flow is identical and uses the same number of components as well as the same order in which the components are arranged as shown in Figure 7.1. Nevertheless, the underlying APIs used in Spark and Flink to achieve the same functionality differ significantly.

Flow validation: Big Data framework-agnostic The flow validation techniques applied at the tool level to ensure that the flow created by the user will generate a compilable and runnable Big Data program is framework-agnostic. In short, we validate:

- The flow begins with an input component, followed by a set of transformation components and ends with an action/output component.
- All the components use the same data abstraction.
- The components arranged in sequence are compatible with the input coming from their predecessors.

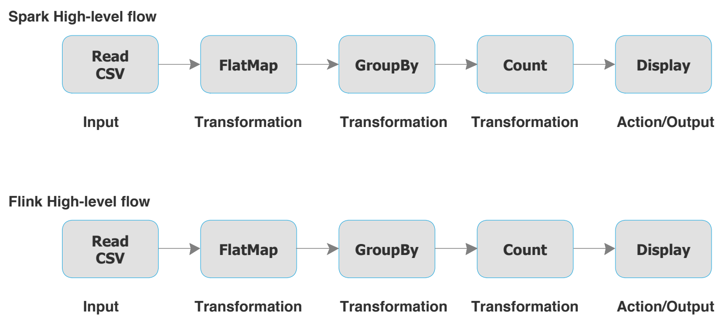


Figure 7.1: High-level Big Data flow to count frequency of words in text collection

Code generation: Big Data framework specific The translation and code generation is target framework specific as the flow abstracts necessary components of the final Big Data program like starting a Flink/Spark session and terminating it. All the code generated by the components in the user flow are inside this skeleton. The components typically represent a stand-alone method implementation of target framework specific APIs.

```
//Spark Code
Dataset<String> textFile = spark.read()
              .textFile("/path/to/file")
              .as(Encoders.STRING());
Dataset<String> words = textFile.flatMap(s -> {
  return  Arrays.asList(s.toLowerCase().split(" ")).iterator();
  }, Encoders.STRING()).filter(s -> !s.isEmpty());

Dataset<Row> counts = words.toDF("word").groupBy(col("word")).count();
counts.show();

//Flink Code

DataSet<String> text = env.readTextFile("/path/to/file");

DataSet<Tuple2<String, Integer>> counts =
         text.flatMap(new Tokenizer())
         .groupBy(0)
         .sum(1);
counts.writeAsCsv(outputPath, "\n", " ");
```

Listing 7.1: Generated code for Spark and Flink

Spark and Flink use different methods to deliver similar results. For instance, the ‘count’ transformation component used in both the flows in Figure 7.1 internally represent different methods of Spark (count()) and Flink (sum()). Additionally, the internal model generation from the user flow after passing the validation uses target framework specific invocation statements to invoke the standalone method implementation of APIs. The codes generated for Spark and Flink for the flows depicted in Figure 7.1 are shown in Listing 7.1.

Abstraction of target framework execution environment internals The user-flow and the resultant code generation process for both Spark and Flink hide from the user the internal workings of the execution environment of the target frameworks. For example, both Spark and Flink use lazy evaluation [103] to minimise computation overheads. But there is difference in the way the lazy evaluation is handled in Spark and Flink.

In Spark, execution does not start until an action is applied on a RDD. This means that all Spark transformations are handled lazily, i.e. when a transformation is called it is not really executed but only computed when its result is required which also implies that some datasets may not be read if their output is not used. Consider, an example where a log file of 1 GB stored in HDFS blocks needs to be processed for error entries. and fetch the first entry. The logical step is to create a RDD from the log file followed by creation of another RDD which contains only the filtered error entries, from which the first entry needs to be extracted. In case of eager evaluation, Spark would have processed all blocks of the log file to create a RDD containing all error messages even when only the first entry was required while in lazy evaluation it stops processing the different blocks of the log file whenever the first entry is found. In short, Spark records the transformations applied on datasets in the form a DAG and does absolutely nothing but when asked for result, it first optimises to do the minimum needed to deliver the result. By optimisation, it is mean that when an action is encountered Spark checks the DAG and finds out what operations are needed and clubs them together into a single MapReduce pass which minimises the number of map and reduce pass which a developer needs to take care of manually when programming directly using the MapReduce programming model. When the main method of a Spark program starts execution, *Spark waits for the first action encounter on an RDD to start the actual execution*.

Flink handles the lazy evaluation bit differently, i.e. when the main method is executed, neither the datasets are loaded nor the transformations are computed. Instead, an execution plan is created containing the sequence of operations to be computed. These operations are *executed when the execution is explicitly triggered by an execute() call on the execution environment*. This is distinct from the Spark lazy evaluation as Spark starts execution when it first encounters an action statement while Flink ignores even action statements but starts when it encounters execute(), the last statement of a Flink program. This waiting till the end of the program allows Flink optimiser to do a lot more optimisations of operations compared to its counterpart. As part of our investigation, we did not find any work comparing the degree of laziness supported by both the target execution environments.

The user programming these frameworks from a high-level is shielded from these difference which may produce slight difference in result.

Extending to additional target frameworks As discussed in Section 7.1.1, the conceptual approach has two aspects to it. The first step is to analyse a different Big Data framework and model APIs as modular components. The second aspect is to support parsing of a high-level graphical flow created from such components and generate the target Big Data program. The second step has aspects which are framework-agnostic like flow creation rules, flow validation and support for parametrisation of components which are already compatible. The code generation process, though, has aspects which are target framework specific, e.g. generating code-snippets to initialise and terminate a session within the execution engine of a Big Data framework. These aspects need to handled for each supported framework which also includes components containing stand-alone method implementation of target framework APIs which are invoked during the code-generation process. This is in contrast to the code-generation techniques used in existing solutions like Lemonade, QM-IConf, Nussknacker, QryGraph where the high-level programming approach is tightly-coupled to the target Big Data framework. *The graphical programming approach have already been extended to support additional target frameworks including Pig and Hive.*

Unified programming for Spark and Flink: feasibility From the discussion in Section 7.1.2 it might appear that it would be feasible to formulate a unified graphical query language and using the conceptual approach to generate Spark and Flink program for a same user flow. In reality, there are fundamental differences where Spark and Flink differ in terms of underlying architecture, data processing style and data abstractions offered. Spark grew out from the older Hadoop ecosystem in order to write easily optimised MapReduce programs and is lambda architecture compatible. Flink on the other hand is a framework and distributed processing engine for performing stateful computations over unbounded and bounded data streams built around the kappa architecture. The Kappa architecture, a simplification of the lambda architecture, relies on data streams instead of using persisted datasets and has the following key concepts:

1. Everything is a stream.
2. Data sources are immutable.
3. Support for replay, i.e. reprocessing of all data which has already been processed.

```
//Spark code
Dataset<Row> salesSum = spark.read()
                             .option("mode", "DROPMALFORMED")
                             .option("header", "true")
                             .option("inferSchema", "true")
                             .csv(file)
                             .groupBy("id", "date")
                             .sum("amount")
                             .groupBy("date");

WindowSpec wSpec = Window.partitionBy("date")
                         .orderBy(salesSum.col("sum(amount)").desc());

// window Operation, for gathering the "id" column in the output
salesSum.withColumn("rank", dense_rank().over(wSpec))
                                        .select("id", "sum(amount)", "date")
                                        .where("rank == 1")
                                        .show();
//Flink code
env.readCsvFile(file)
   .ignoreInvalidLines()
   .types(Integer.class, Integer.class, String.class)
   .groupBy(0,2)
   .sum(1)
   .groupBy(2)
```

```
    .maxBy(1)
    .print();
```

Listing 7.2: Difference in equivalent code in Spark and Flink

The key difference is in the underlying computational concepts. Spark has a batch concept and uses micro-batches (data in buckets) to support streaming while in Flink is a stream processing (data is streamed as it arrives, i.e. event-based) and treats batch processing as a special case of stream processing.

Due to fundamental differences in computational models, generic code generation for both target frameworks in not feasible for all cases. Additionally, there are difference in methods needed to do basic data processing and both frameworks use different approaches to achieve result. For example, consider a use case where a program needs to compute the most frequently sold product from a CSV file containing three fields product id, date and amount. Spark does not have support for maxBy() method of Flink and it has to be done via other approaches as shown in code Listing 7.2. Similarly Spark has reduceByKey() which when ported to Flink necessitates the usage of a pair method of groupBy() and sum(). A simple missing equivalent method involves lot of manual code writing and this might be feasible for specific use cases but difficult to generalise.

Additionally, Flink iterates over each dataset based on event, filters, groups, applies transformation and sends for output while in Spark events are already grouped by time hence directly proceeds to application of filtering, transformation and finally send to output. Due to inherent difference in way stream processing is handled, a unified high-level language is not feasible without affecting generalisability of the approach when applied to the whole frameworks.

7.2 Conclusion

Data analytics is gaining prominence and its even more relevant in the context of IoT where the generated datasets needs to be analysed to make a business decision. Handling large volume of data and also processing data streams require cluster based data science methods like Big Data analytics. 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. Hence, experts find it difficult to select a particular framework for their needs. The ecosystem has no support for end-user programming which restricts it widespread application among domain-experts who are not programmers. In this context, the main idea of the thesis was to provide domain-experts with flow-based graphical tools for high-level programming of Big Data applications.

IoT mashup tools based on the graphical flow-based programming paradigm has been successfully used to lower the development effort in the context of IoT applications. The thesis used the existing concept of graphical flow-based programming paradigm of mashup tools to enable high-level programming for Big Data applications.

Succinctly, the contributions made in this thesis are:

Flow-based Programming for Data Analytics Improved graphical flow-based programming tool concepts based on actor model with support for concurrent execution semantics to support in-flow data analytics has been proposed and prototyped (Chapter 4). The integrated stream processing capabilities along with support for parametrising the controlling factors of stream processing abstracts the various methods of stream processing and enables non-experts to prototype adjustable stream analytics jobs. A flow-based programming model with concurrent execution semantics is found to be suitable for modelling a wide range of Big Data applications. *The improvements over the current state-of-the-art existing solutions is discussed in Section 4.5.*

Development of modular component libraries of Big Data frameworks The Big Data frameworks like Spark and Flink have their own set of APIs, data abstractions and different libraries providing different functionalities. Programmers have to interact with these APIs and data abstractions to develop Big Data applications. The first step in supporting high-level graphical programming for Big Data applications was to analyse the target frameworks, select APIs and data abstractions that are compatible and easy to model in flow-based programming paradigm, i.e. *not supporting APIs requiring user defined data transformation functions or supporting code-snippets during flow creation to interact with target framework internals.*

Spark

1. An analysis of the different data abstractions of Spark helped to select the most suitable ones for use in a graphical flow-based programming paradigm. From the available data abstractions of Spark, DStream and DataFrame (including Streaming DataFrame), have been selected for the thesis work. The conceptual approach described in Section 5.5 supports only the transformations accessible via untyped, i.e. DataFrame APIs (Figure 5.6) and using the aforementioned data abstractions. Any transformation making use of RDD based approach involving user defined data transformation is not supported.

2. An analysis of the Spark framework was done which helped to formulate a unified classification of APIs present in different Spark libraries (Section 5.4). This is essential *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.*

3. *SparFlo* is a library created consisting of modular composable 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 (Section 5.7). These modular components are composable when composed confirming to the flow compositional rules discussed in Section 5.6.1.3. 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 (Section 5.8).

Flink

1. An analysis of the Flink framework was done to select the most suit-

able programming and data abstractions for use in a graphical flow-based programming paradigm. From the available programming abstractions of Flink,programming via core APIs operating on Data DataSet and DataStream data abstractions, have been selected for the thesis work (Section 6.2). The conceptual approach described in Section 6.4 supports only the transformations accessible via core APIs and using the aforementioned data abstractions. Any transformation making use of low-level stateful stream processing of Flink or programming via the SQL/Table API is not supported.

2. *FlinkFlo* is a library created consisting of modular composable components. The components in *FlinkFlo* bundle a set of Flink APIs from the selected subset of Flink APIs which are executed in a specific order to perform one data analytic operation (Section 6.5). The components of *FlinkFlo* can be used in any implementing tool to enable high-level programming of Flink applications.

High-level graphical programming for Big Data applications A generic approach for Spark (Section 5.6) and Flink programming (Section 6.4) via graphical flows, validation of graphical flows and auto-generation of Spark and Flink program created using *SparFlo* and *FlinkFlo* components has been developed. *The improvements over the current state-of-the-art existing solutions providing similar high-level programming for Big Data applications is discussed in Section 7.1.1.*

The graphical programming concepts for Spark and Flink have been prototyped and evaluated in separate use cases to demonstrate the ease of use, code-abstraction and automatic data interface conversion, which are the keys in lowering the complexity and its ensued learning curve involved in the development of Big Data applications.

A Appendix for Spark

A.1 Spark Code Listings

```
public  static  Dataset<Row> process(SparkSession spark,
                    String key,
                    Properties userInput){

 // initialise an empty data frame
Dataset<Row> inputDataSet = spark.emptyDataFrame();

// read from Properties file
String loadPath = userInput.getProperty(key+"-loadPath");
String inputFormat = userInput.getProperty(key+"-fileFormat");
String[] fieldNames = userInput.getProperty(key+"-fieldNames").split(",");

// prepare schema from user input
String[] dataTypes =  userInput.getProperty(key+"-dataTypes").split(",");
List<StructField> fields = new ArrayList<>();
for( int i= 0; i <fieldNames.length;i++) {
DataType dt = getDataType(dataTypes[i]);
StructField field = DataTypes.createStructField(fieldNames[i], dt, true);
fields.add(field);
}
StructType inputSchema = DataTypes.createStructType(fields);

inputDataSet = spark.read()
                      .format("csv")
                      .option("header", "false")
                      .option("mode", "DROPMALFORMED")
                      .option("inferSchema","false")
                      .schema(inputSchema)
                      .load(loadPath);

return inputDataSet;
}
```

Listing A.1: Splux: wrapper method (standalone method implementation of Spark APIs) to read data from a file and create a DataFrame out of it

```
//main method in Spark Driver program
public static void main(String[] args) throws Exception {

Properties featureProp = new Properties();
featureProp = extractProperties(PROPERTIES_PATH);

SparkSession sparkSession = getSparkSession("SparkSession4",featureProp);

Dataset<Row> FileToDataFrame1 = FileToDataFrame1_readDataFrame
(sparkSession,"FileToDataFrame1",featureProp);

ShowDF3_actionOnDataFrame(sparkSession,"ShowDF3",featureProp,FileToDataFrame1);

sparkSession.stop();

}
//Method to invoke Splux API using reflection
public static Properties extractProperties(String path) throws Exception {
Object objForAppStage = Class.forName("org.sparkexample.ExtractProperties").
    newInstance();
Method appStageMethod = objForAppStage.getClass().getMethod("process",String.
    class);
Properties properties = (Properties)appStageMethod
                .invoke(objForAppStage,path);
return properties;
}
//Method to invoke Splux API using reflection
public static SparkSession getSparkSession(String key, Properties featureProp)
    throws Exception {
Object objForAppStage = Class.forName("org.sparkexample.GenerateSparkSession").
    newInstance();
Method appStageMethod = objForAppStage.getClass().getMethod("process",String.
    class,Properties.class);
SparkSession sparkSession = (SparkSession)appStageMethod.invoke(objForAppStage,
    key,featureProp);
return sparkSession;
}
//Method to invoke Splux API using reflection
public static Dataset<Row> FileToDataFrame1_readDataFrame(SparkSession
    sparkSession, String key,
Properties featureProp) throws Exception {
Object objForAppStage = Class.forName("org.sparkexample.FileToDataFrame").
    newInstance();
Method appStageMethod = objForAppStage.getClass().getMethod("process",
    SparkSession.class,String.class,Properties.class);
Dataset<Row> inputData =
(Dataset<Row>)appStageMethod.invoke(objForAppStage,sparkSession,key,featureProp
    );
return inputData;
}
//Method to invoke Splux API component using reflection
public static void ShowDF3_actionOnDataFrame(SparkSession sparkSession, String
    key,Properties featureProp, Dataset<Row> inputDS) throws Exception {
Object objForAppStage = Class.forName("org.sparkexample.ShowDF").newInstance();
Method appStageMethod = objForAppStage.getClass().getMethod("process",
    SparkSession.class,String.class,Properties.class,Object.class);
appStageMethod.invoke(objForAppStage,sparkSession,key,featureProp,inputDS);
}
```

Listing A.2: Sample Spark driver program generated by aFlux

```
//initialize spark session - Step 1 (UC1-S1)
SparkSession spark = SparkSession.builder().master("local[2]").appName("App").
    getOrCreate();

//prepare schema - Begin Step 2 (UC1-S2)
StructType csvschema = DataTypes.createStructType(new StructField[]{
DataTypes.createStructField("timestamp", DataTypes.StringType, true),
DataTypes.createStructField("latitude", DataTypes.DoubleType, true),
DataTypes.createStructField("longitude", DataTypes.DoubleType, true),});

//read data from external file system
Dataset<Row> inputDataSet = spark.read()
                           .format("csv")
                           .option("header", "false")
                           .option("mode", "DROPMALFORMED")
                           .option("inferSchema","false")
                           .schema(csvschema)
                           .load("trip-data.csv"); // - End Step 2

//select features - Begin Step 3 (UC1-S3)
String[] inputCols = {"latitude","longitude"};

// Feature Extraction: Produce a coloums containing features
PipelineStage myAssembler = new VectorAssembler()
               .setInputCols(inputCols)
                           .setOutputCol("features");

//ML Algorithm: apply KMeans algorithm on features
PipelineStage kmeans = new KMeans()
                       .setK(8)
                       .setFeaturesCol("features")
                       .setPredictionCol("Prediction");

//Add stages into a pipeline
PipelineStage[] stage = {myAssembler,kmeans};
Pipeline myPipeline = new Pipeline().setStages(stage);

// Model fitting
PipelineModel model = myPipeline.fit(inputDataSet);

//persist model into external file system
model.write().overwrite().save("models/kmeans"); //- End Step 3

//apply model on test data and print to console - Step4 (UC1-S4)
model.transform(inputDataSet).show();

//stop spark session - Step 5 (UC1-S5)
spark.stop();
```

Listing A.3: Manual programming: batch processing to produce ML model

```
//initialize spark streaming context with micro-batch duration of 1000 ms -
    Begin Step 1 (UC2-S1)
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaStreamingContext ssc = new JavaStreamingContext(conf, new Duration(1000));
    // End - Step 1

//Configure to read from Kafka - Begin Step 2 (UC2-S2)
Map<String, Object> kafkaParams = configurationParamsForKafkaListener();
Collection<String> topics = Arrays.asList(topicArray[0]);

//Produce DStream from Kafka Record
JavaInputDStream<ConsumerRecord<String, String>> stream =
KafkaUtils.createDirectStream(ssc,
LocationStrategies.PreferConsistent(),
ConsumerStrategies.<String,                    String>Subscribe(topics,
    kafkaParams));

JavaPairDStream<String, String> input = stream.mapToPair(
                              record ->
                              new Tuple2<(record.key(),record.value()));

JavaDStream<String> inputStream =input.map(
new Function<Tuple2<String,String>,String>(){
public String call(
Tuple2<String, String> tuple2){
return tuple2._2();}
});//- End Step 2

//Convert DStream to RDD to DataFrame to apply saved ML Model - //Begin Step 3
    (UC2-S3)
inputStream.foreachRDD(new VoidFunction<JavaRDD<String>>() {
// Pipeline model object will be  serialized and sent from the driver to the
    worker
PipelineModel savedModel =  PipelineModel.read().load("models/kmeans");
@Override
public void call(JavaRDD<String> rdd) {
JavaRDD<Row> rowRDD = rdd.map(
  (Function<String, Row>) record -> {
  String[] attributes = record.split(",");
  return RowFactory.create(attributes[0].trim(),
  Double.parseDouble(attributes[1].trim()),
  Double.parseDouble(attributes[2].trim()),
  attributes[3].trim());
});

StructType csvschema =
DataTypes.createStructType(new StructField[]{
DataTypes.createStructField("timestamp", DataTypes.StringType, true),
DataTypes.createStructField("latitude", DataTypes.DoubleType, true),
DataTypes.createStructField("longitude", DataTypes.DoubleType, true)});
Dataset<Row> msgDataFrame = spark.createDataFrame(rowRDD, csvschema);

//Apply model
msgDataFrame =  savedModel.transform(msgDataFrame);//-End Step 3

//Publish to kafka - Begin Step 4 (UC2-S4)
JavaRDD<Row> toRDD = msgDataFrame.toJavaRDD();          toRDD.foreach(rowrdd->{
Properties kafkaProp = new Properties();
kafkaProp.put("bootstrap.servers", bootStrapServre);
kafkaProp.put("key.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
kafkaProp.put("value.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
KafkaProducer<String, String> producer = new     KafkaProducer<String, String>(
    kafkaProp);
String data = rowrdd.get(0).toString();
//Row to String
for(int i=1;i <= rowrdd.length()-1;i++) {
  data += ",";
  data += rowrdd.get(i).toString();

}
producer.send(new ProducerRecord<String, String>("enrichedData",data));
producer.close();});
}
} // - End Step 4
//Start Streaming Session and wait till termination - Step 5 (UC2-S5)
ssc.start();
ssc.awaitTermination();
```

Listing A.4: Manual programming: stream processing and applying a ML model

```
//create spark session - Step 1 (UC3-S1)
SparkSession spark = SparkSession.builder().master("local[2]").appName("
    SparkApp").getOrCreate();

//produce schema - Begin Step 2 (UC3-S2)
StructType csvschema =
DataTypes.createStructType(new StructField[]{
DataTypes.createStructField("timestamp", DataTypes.StringType, true),DataTypes.
    createStructField("latitude", DataTypes.StringType, true),DataTypes.
    createStructField("longitude", DataTypes.StringType, true),});

//read from Kafka
Dataset<Row> rawData = spark.readStream()
.format("kafka")
.option("kafka.bootstrap.servers",BOOTSTRAP_SERVERS)
.option("subscribe","enrichedCarData").load()
.selectExpr("CAST(value AS STRING) as message")
.select(functions.from_json(functions.col("message"),csvschema).as("json"))
.select("json.*");

//Cast into schema
Dataset<Row> castData = rawData.
withColumn("timestamp",
functions.unix_timestamp(functions.col("timestamp"),"dd/mm/yyyy hh:mm:ss.SSS").
    cast(DataTypes.TimestampType))
.withColumn("latitude", functions.col("latitude").cast(DataTypes.DoubleType))
.withColumn("longitude", functions.col("longitude").cast(DataTypes.DoubleType))
.withColumn("eventTime", functions.current_timestamp())); // -End Step 2

//Windowed count with late-data hadling - Begin Step 3 (UC3-S3)
Dataset<Row> windowedCounts = castData
.withWatermark("eventTime", "3 minutes")
.groupBy(functions.window(functions.col("eventTime"), "2 minutes", "2 minutes")
    ,functions.col("junctionID"))
.count().withColumn("carcount", functions.col("count").cast(DataTypes.
    StringType))
.withColumn("value",functions.col("carcount").cast(DataTypes.StringType))
.select("value");
// End of Step 3

//Publish to Kafka - Begin Step 4 (UC3-S4)
StreamingQuery streamQ = windowedCounts
.writeStream()
.format("kafka")
.option("topic", "update")
.option("failOnDataLoss", "false")
.option("kafka.bootstrap.servers", bootstrapServers)
.trigger(Trigger.ProcessingTime("300 seconds"))
.option("checkpointLocation",checkpointPath)
.start();
// Step 5 (UC3-S5)
streamQ.awaitTermination();
```

Listing A.5: Manual programming: stream processing to apply aggregations

```
package org.sparkexample;

import java.lang.Exception;
import java.lang.String;
import java.lang.reflect.Method;
import java.util.Properties;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public class App {
  public static void main(String[] args) throws Exception {
    Properties featureProp = new Properties();
    featureProp = extractProperties("hdfs://vm-10-155-208-115.cloud.mwn.de
   :8020/user/tanmaya/DemoFolder/Application1");
    SparkSession sparkSession = getSparkSession("SparkSession3",featureProp);
    Dataset<Row> FileToDataFrame1 = FileToDataFrame1_readDataFrame(sparkSession
   ,"FileToDataFrame1",featureProp);
    Dataset<Row> ProduceModel2 = ProduceModel2_transformDataFrameC(sparkSession
   ,"ProduceModel2",featureProp,FileToDataFrame1);
  }

  public static Properties extractProperties(String path) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   ExtractProperties").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class);
    Properties properties = (Properties)appStageMethod.invoke(objForAppStage,
   path);
    return properties;
  }

  public static SparkSession getSparkSession(String key, Properties featureProp
   ) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   GenerateSparkSession").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class,Properties.class);
    SparkSession sparkSession = (SparkSession)appStageMethod.invoke(
   objForAppStage,key,featureProp);
    return sparkSession;
  }

  public static Dataset<Row> FileToDataFrame1_readDataFrame(SparkSession
   sparkSession, String key,
      Properties featureProp) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.FileToDataFrame"
   ).newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   SparkSession.class,String.class,Properties.class);
    Dataset<Row> inputData = (Dataset<Row>)appStageMethod.invoke(objForAppStage
   ,sparkSession,key,featureProp);
    return inputData;
  }

  public static Dataset<Row> ProduceModel2_transformDataFrameC(SparkSession
   sparkSession, String key,
      Properties featureProp, Dataset<Row> inputDS) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.ProduceModel").
   newInstance();
```

```
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
    SparkSession.class,String.class,Properties.class,Object.class);
    Dataset<Row> inputData = (Dataset<Row>)appStageMethod.invoke(objForAppStage
    ,sparkSession,key,featureProp,inputDS);
    return inputData;
  }
}
```

Listing A.6: aFlux generated Spark code for use case 1

```
#Sat Jul 07 17:21:03 CEST 2018
FeatureAssembler2-inputCol=latitude,longitude
KMeansCluster2-outputCol=prediction
SparkSession3-sparkConfPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/spark-conf.properties
FeatureAssembler2-outputCol=features
FileToDataFrame1-fieldNames=timestamp,latitude,longitude,base
ProduceModel2-pipelineStages=FeatureAssembler2,KMeansCluster2
FileToDataFrame1-dataTypes=String,Double,Double,String
KMeansCluster2-numClusters=2
FileToDataFrame1-fileFormat=CSV
FileToDataFrame1-loadPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/InputFiles/uber-raw-data-apr14.csv
ProduceModel2-savePath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/Models/kmeans
KMeansCluster2-inputCol=features
```

Listing A.7: Generated property file for use case 1

```java
package org.sparkexample;

import java.lang.Exception;
import java.lang.String;
import java.lang.reflect.Method;
import java.util.Properties;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;

public class App {
  public static void main(String[] args) throws Exception {
    Properties featureProp = new Properties();
    featureProp = extractProperties("hdfs://vm-10-155-208-115.cloud.mwn.de
   :8020/user/tanmaya/DemoFolder/Application2");
    SparkSession sparkSession = getSparkSession("SparkSession4",featureProp);
    JavaStreamingContext jsc = getJavaStreamingContextMethod(sparkSession,
   featureProp);;
    JavaDStream<String> KafkaToDStream1 = KafkaToDStream1_readDStream(jsc,"
   KafkaToDStream1",featureProp);
    ApplyModelOnDStream2_transformDStream(sparkSession,"ApplyModelOnDStream2",
   featureProp,KafkaToDStream1);
    jsc.start();
    jsc.awaitTermination();
  }

  public static Properties extractProperties(String path) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   ExtractProperties").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class);
    Properties properties = (Properties)appStageMethod.invoke(objForAppStage,
   path);
    return properties;
  }

  public static SparkSession getSparkSession(String key, Properties featureProp
  ) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   GenerateSparkSession").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class,Properties.class);
    SparkSession sparkSession = (SparkSession)appStageMethod.invoke(
   objForAppStage,key,featureProp);
    return sparkSession;
  }

  public static JavaStreamingContext getJavaStreamingContextMethod(SparkSession
    sparkSession,
      Properties featureProp) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   GetJavaStreamingContext").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   SparkSession.class,Properties.class);
    JavaStreamingContext ssc = (JavaStreamingContext)appStageMethod.invoke(
   objForAppStage,sparkSession,featureProp);
    return ssc;
  }
  public static JavaDStream<String> KafkaToDStream1_readDStream(
    JavaStreamingContext ssc,
       String key, Properties featureProp) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.KafkaToDStream")
    .newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
    JavaStreamingContext.class,String.class,Properties.class);
    JavaDStream<String> inputData = (JavaDStream<String>)appStageMethod.invoke(
    objForAppStage,ssc,key,featureProp);
    return inputData;
  }

  public static void ApplyModelOnDStream2_transformDStream(SparkSession
    sparkSession, String key,
       Properties featureProp, JavaDStream<String> inputDStream) throws
    Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
    ApplyModelOnDStream").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
    JavaStreamingContext.class,String.class,Properties.class,Object.class);
    appStageMethod.invoke(objForAppStage,sparkSession,key,featureProp,
    inputDStream);
  }
}
```

Listing A.8: aFlux generated Spark code for use case 2

```
#Sat Jul 07 18:32:54 CEST 2018
microBatchDuration=5000
RDDToDF3-dataTypes=String,Double,Double,String
KafkaToDStream1-bootStrapServers=191.168.22.91\:9092
KafkaToDStream1-fieldNames=timestamp,latitude,longitude,base
RDDToDF3-fieldNames=timestamp,latitude,longitude,base
DStreamToKafka3-bootStrapServer=191.168.22.91\:9092
SparkSession4-sparkConfPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/spark-conf.properties
KafkaToDStream1-dataTypes=String,Double,Double,String
KafkaToDStream1-subscribeToTopic=iotData
DStreamToKafka3-topicToPublish=enrichedData
ApplyModelOnDF3-loadPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/Application2/Kmeans
ApplyModelOnDStream2-transformPipeline=RDDToDF3,ApplyModelOnDF3,DStreamToKafka3
```

Listing A.9: Generated property file for use case 2

```
package org.sparkexample;

import java.lang.Exception;
import java.lang.String;
import java.lang.reflect.Method;
import java.util.Properties;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public class App {
  public static void main(String[] args) throws Exception {
    Properties featureProp = new Properties();
    featureProp = extractProperties("hdfs://vm-10-155-208-115.cloud.mwn.de
   :8020/user/tanmaya/DemoFolder/Application1");
    SparkSession sparkSession = getSparkSession("SparkSession3",featureProp);
    Dataset<Row> FileToDataFrame1 = FileToDataFrame1_readDataFrame(sparkSession
   ,"FileToDataFrame1",featureProp);
    Dataset<Row> ProduceModel2 = ProduceModel2_transformDataFrame(sparkSession,
   "ProduceModel2",featureProp,FileToDataFrame1);
  }

  public static Properties extractProperties(String path) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   ExtractProperties").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class);
    Properties properties = (Properties)appStageMethod.invoke(objForAppStage,
   path);
    return properties;
  }

  public static SparkSession getSparkSession(String key, Properties featureProp
   ) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.
   GenerateSparkSession").newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   String.class,Properties.class);
    SparkSession sparkSession = (SparkSession)appStageMethod.invoke(
   objForAppStage,key,featureProp);
    return sparkSession;
  }

  public static Dataset<Row> FileToDataFrame1_readDataFrame(SparkSession
   sparkSession, String key,
      Properties featureProp) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.FileToDataFrame"
   ).newInstance();
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
   SparkSession.class,String.class,Properties.class);
    Dataset<Row> inputData = (Dataset<Row>)appStageMethod.invoke(objForAppStage
   ,sparkSession,key,featureProp);
    return inputData;
  }

  public static Dataset<Row> ProduceModel2_transformDataFrame(SparkSession
   sparkSession, String key,
      Properties featureProp, Dataset<Row> inputDS) throws Exception {
    Object objForAppStage = Class.forName("org.spark_for_aflux.ProduceModel").
   newInstance();
```

```
    Method appStageMethod = objForAppStage.getClass().getMethod("process",
    SparkSession.class,String.class,Properties.class,Object.class);
    Dataset<Row> inputData = (Dataset<Row>)appStageMethod.invoke(objForAppStage
    ,sparkSession,key,featureProp,inputDS);
    return inputData;
  }
}
```

Listing A.10: aFlux generated Spark code for use case 3

```
#Sat Jul 07 17:21:03 CEST 2018
FeatureAssembler2-inputCol=latitude,longitude
KMeansCluster2-outputCol=prediction
SparkSession3-sparkConfPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/spark-conf.properties
FeatureAssembler2-outputCol=features
FileToDataFrame1-fieldNames=timestamp,latitude,longitude,base
ProduceModel2-pipelineStages=FeatureAssembler2,KMeansCluster2
FileToDataFrame1-dataTypes=String,Double,Double,String
KMeansCluster2-numClusters=2
FileToDataFrame1-fileFormat=CSV
FileToDataFrame1-loadPath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/InputFiles/uber-raw-data-apr14.csv
ProduceModel2-savePath=hdfs\://vm-10-155-208-115.cloud.mwn.de\:8020/user/
    tanmaya/DemoFolder/Models/kmeans
KMeansCluster2-inputCol=features
```

Listing A.11: Generated property file for use case 3

A.2 Analysis of Spark APIs

Table A.1: Analysis of Spark APIs

SN	Spark API	Input	Output	Spark Library	API Type	Classification
1.	csv()	Dataset<String> csvDataset	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads an Dataset[String] storing CSV rows and returns the result as a DataFrame.						
2.	csv()	String... paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads CSV files and returns the result as a DataFrame.						
3.	csv()	String path	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads a CSV file and returns the result as a DataFrame.						
4.	json()	Dataset<String> jsonDataset	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads a Dataset[String] storing JSON objects and returns the result as a DataFrame.						
5.	json()	String... paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads JSON files and returns the results as a DataFrame.						
6.	json()	String path	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads a JSON file and returns the results as a DataFrame.						
7.	orc()	scala.collection.Seq<String> paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads ORC files from the collection and returns the result as a DataFrame.						
8.	orc()	String... paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads ORC files and returns the result as a DataFrame.						
9.	orc()	String path	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads an ORC file and returns the result as a DataFrame.						
10.	parquet()	scala.collection.Seq<String> paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads Parquet files from the collection, returning the result as a DataFrame.						
11.	parquet()	String... paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads Parquet files, returning the result as a DataFrame.						

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
12.	parquet()	String path	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads Parquet files, returning the result as a DataFrame.						
13.	text()	scala.collection.Seq<String> paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads text files from the collection and returns a DataFrame.						
14.	text()	String... paths	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads text files and returns a DataFrame.						
15.	text()	String path	Dataset<Row>	Spark SQL	Transformation	**Input**
Functionality: Loads text files and returns a DataFrame.						
16.	textFile()	scala.collection.Seq<String> paths	Dataset<String>	Spark SQL	Transformation	**Input**
Functionality: Loads text files and returns a Dataset of String.						
17.	textFile()	String... paths	Dataset<String>	Spark SQL	Transformation	**Input**
Functionality: Loads text files and returns a Dataset of String.						
18.	textFile()	String path	Dataset<String>	Spark SQL	Transformation	**Input**
Functionality: Loads text files and returns a Dataset of String.						
19.	csv()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads a CSV file stream and returns the result as a DataFrame.						
20.	json()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads a JSON file stream and returns the results as a DataFrame.						
21.	load()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads input in as a DataFrame, for data streams that read from some path.						
22.	orc()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
Functionality: Loads a ORC file stream, returning the result as a DataFrame.						
23.	parquet()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads a Parquet file stream, returning the result as a DataFrame.						
24.	text()	String path	Dataset<Row>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads text files and returns a DataFrame						
25.	textFile()	String path	Dataset<String>	Spark Structured Streaming	Transformation	**Input**
Functionality: Loads text file(s) and returns a Dataset of String.						
26.	socketTextStream()	String hostname, int port, StorageLevel storageLevel	ReceiverInput DStream <String>	Spark Streaming	Transformation	**Input**
Functionality: Creates an input stream from TCP source hostname:port.						
27.	receiverStream()	Receiver<T> receiver, scala.reflect.ClassTag<T> evidence$1	<T> ReceiverInput DStream<T>	Spark Streaming	Transformation	**Input**
Functionality: Create an input stream with any arbitrary user implemented receiver.						
28.	binary RecordsStream()	String directory, int recordLength	DStream<byte[]>	Spark Streaming	Transformation	**Input**
Functionality: Create an input stream that monitors a Hadoop-compatible filesystem for new files and reads them as flat binary files						

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
29.	fileStream()	String directory, scala.reflect.ClassTag<K> evidence$4, scala.reflect.ClassTag<V> evidence$5, scala.reflect.ClassTag<F> evidence$6	<K,V,F extends org.apache.hadoop .mapreduce .InputFormat<K,V» InputDStream <scala.Tuple2<K,V»	Spark Streaming	Transformation	**Input**
Functionality: Create an input stream that monitors a Hadoop-compatible filesystem for new files and reads them as flat binary files						
30.	rawSocketStream()	String hostname, int port, StorageLevel storageLevel, scala.reflect.ClassTag<T> evidence$3	<T> ReceiverInput DStream<T>	Spark Streaming	Transformation	**Input**
Functionality: Create an input stream from network source hostname:port, where data is received as serialized blocks						
31.	csv()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame in CSV format at the specified path.						
32.	json()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame in JSON format at the specified path.						
33.	orc()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame in ORC format at the specified path.						
33.	parquet()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame in Parquet format at the specified path.						
34.	save()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame at the specified path.						
35.	text()	String path	void	Spark SQL	Action	**Action**
Functionality: Saves the content of the DataFrame in a text file at the specified path.						
36.	start()	String path	StreamingQuery	Spark SQL	Action	**Action**
Functionality: Starts the execution of the streaming query & continually output results to the given path as new data arrives.						

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
37.	write()	-	DataFrame Writer<T>	Spark SQL	Action	**Action**
Functionality: Interface for saving the content of the non-streaming Dataset out into external storage						
38.	writeStream()	-	DataStream Writer<T>	Spark Structured Streaming	Action	**Action**
Functionality: Interface for saving the content of the streaming Dataset out into external storage						
39.	show()	-	-	Spark SQL	Action	**Action**
Functionality: Displays the top 20 rows of Dataset in a tabular form.						
40.	saveAsTextFiles()	String prefix, String suffix	void	Spark Streaming	Action	**Action**
Functionality: Save each RDD in this DStream as at text file, using string representation of elements.						
41.	saveAsObjectFiles()	String prefix, String suffix	void	Spark Streaming	Action	**Action**
Functionality: Save each RDD in this DStream as a Sequence file of serialized objects.						
42.	distinct()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset that contains only the unique rows from this Dataset.						
43.	drop()	Column col	Dataset<Row>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset with a column dropped.						
44.	dropDuplicates()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset that contains only the unique rows from this Dataset.						
45.	localCheckpoint()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Eagerly locally checkpoints a Dataset and return the new Dataset.						
46.	sort()	Column... sortExprs	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset sorted by the given expressions.						
47.	toDF()	-	Dataset<Row>	Spark SQL	Transformation	**Type A**
Functionality: Converts this strongly typed collection of data to generic Dataframe.						
48.	limit()	int n	Dataset<T>	Spark SQL	Transformation	**Type A**

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
Functionality: Returns a new Dataset by taking the first n rows.						
49.	as()	String alias	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset with an alias set.						
50.	cache()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Persist this Dataset with the default storage level.						
51.	coalesce()	int numPartitions	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset that has exactly numPartitions partitions.						
52.	describe()	String... cols	Dataset<Row>	Spark SQL	Transformation	**Type A**
Functionality: Computes basic statistics for numeric and string columns						
53.	filter()	Column condition	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Filters rows using the given condition.						
54.	orderBy()	Column... sortExprs	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset sorted by the given expressions.						
55.	persist()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Persist this Dataset with the default storage level.						
56.	repartition()	Column... partitionExprs	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset partitioned by the given partitioning expressions.						
57.	select()	Column... cols	Dataset<Row>	Spark SQL	Transformation	**Type A**
Functionality: Selects a set of column based expressions.						
58.	sort Within Partitions()	Column... sortExprs	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Returns a new Dataset with each partition sorted by the given expressions.						
59.	summary()	String... statistics	Dataset<Row>	Spark SQL	Transformation	**Type A**
Functionality: Computes specified statistics for numeric and string columns.						
60.	unpersist()	-	Dataset<T>	Spark SQL	Transformation	**Type A**
Functionality: Mark the Dataset as non-persistent.						
61.	where()	String conditionExpr	Dataset<T>	Spark SQL	Transformation	**Type A**

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
Functionality: Filters rows using the given SQL expression.						
62.	cache()	-	JavaDStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Persist RDDs of this DStream with the default storage level						
63.	checkpoint()	Duration interval	static DStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Enable periodic checkpointing of RDDs of this DStream.						
64.	count()	-	static JavaD-Stream<Long>	Spark Streaming	Transformation	**Type A**
Functionality: Return a new DStream in which each RDD has a single element generated by counting each RDD of this DStream.						
65.	filter()	Function<T,Boolean> f	JavaDStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Return a new DStream containing only the elements that satisfy a predicate.						
66.	window()	Duration windowDuration	JavaDStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Return a new DStream in which each RDD contains all the elements in seen in a sliding window of time over this DStream.						
67.	repartition()	int numPartitions	JavaDStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Return a new DStream with an increased or decreased level of parallelism.						
68.	persist()	-	JavaDStream<T>	Spark Streaming	Transformation	**Type A**
Functionality: Persist RDDs of this DStream with the default storage level.						
69.	glom()	-	static JavaDStream <java.util.List<T>>	Spark Streaming	Transformation	**Type A**
Functionality: Return a new DStream in which each RDD is generated by applying glom() to each RDD of this DStream.						
70.	except()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing rows in this Dataset but not in another Dataset.						
71.	unionByName()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing union of rows in this Dataset and another Dataset.						
72.	union()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing union of rows in this Dataset and another Dataset.						
73.	join()	Dataset<?> right	Dataset<Row>	Spark SQL	Transformation	**Type B**

Continued on next page

Table A.1 – continued from previous page

SN	Spark API	Input	Output	Spark Library	API Type	Classification
Functionality: Join with another DataFrame.						
74.	crossJoin()	Dataset<?> right	Datset<Row>	Spark SQL	Transformation	**Type B**
Functionality: Explicit cartesian join with another DataFrame.						
75.	exceptAll()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing rows in this Dataset but not in another Dataset.						
76.	intersect()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing rows only in both this Dataset and another Dataset.						
77.	intersectAll()	Dataset<T> other	Dataset<T>	Spark SQL	Transformation	**Type B**
Functionality: Returns a new Dataset containing rows only in both this Dataset and another Dataset while preserving the duplicates.						
78.	union()	DStream<T> that	DStream<T>	Spark Streaming	Transformation	**Type B**
Functionality:						
79.	join()	Dataset<?> right, Column joinExprs	Dataset<Row>	Spark SQL	Transformation	**Type C**
Functionality: Inner join with another DataFrame, using the given join expression.						
80.	joinWith()	Dataset<U> other, Column condition	<U> Dataset <scala.Tuple2 <T,U> >	Spark SQL	Transformation	**Type C**
Functionality:Using inner equi-join to join this Dataset returning a Tuple2 for each pair where condition is true.						
81.	transform WithToPair()	JavaPairDStream<K2,V2> other, Function3<R, JavaPairRDD<K2,V2>, Time, JavaPairRDD<K3,V3> > transformFunc	Static <K2,V2, K3,V3> JavaPairDStream <K3,V3>	Spark Streaming	Transformation	**Type D**

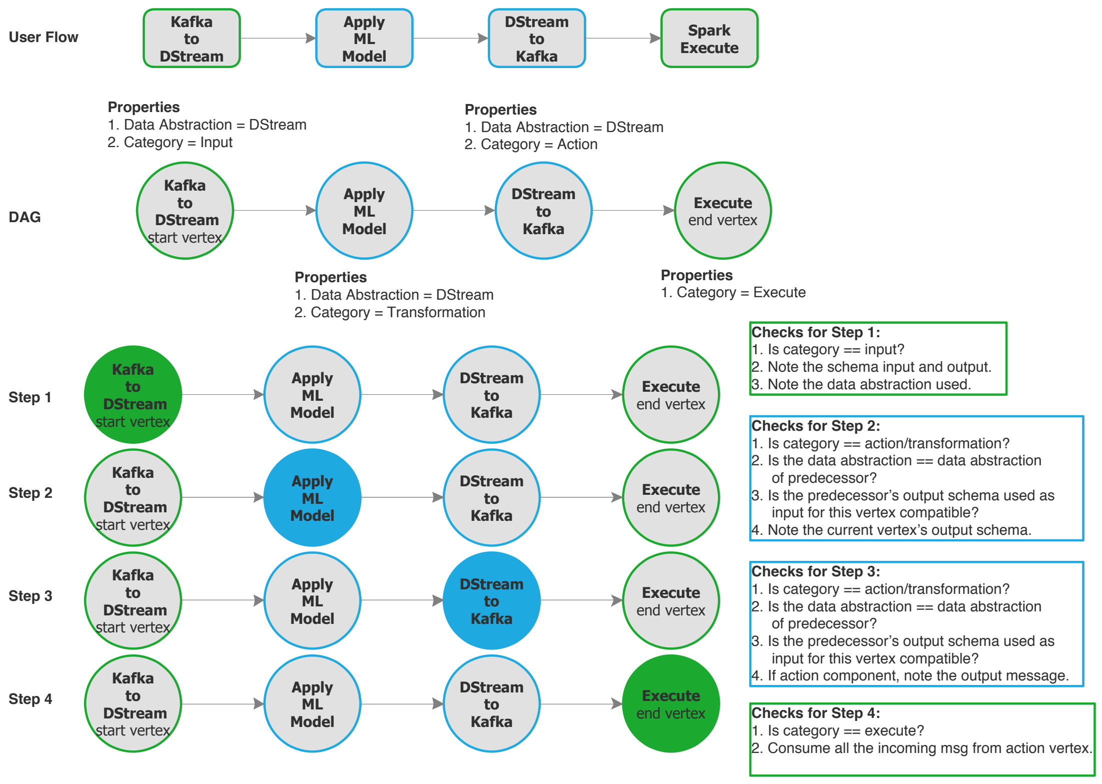
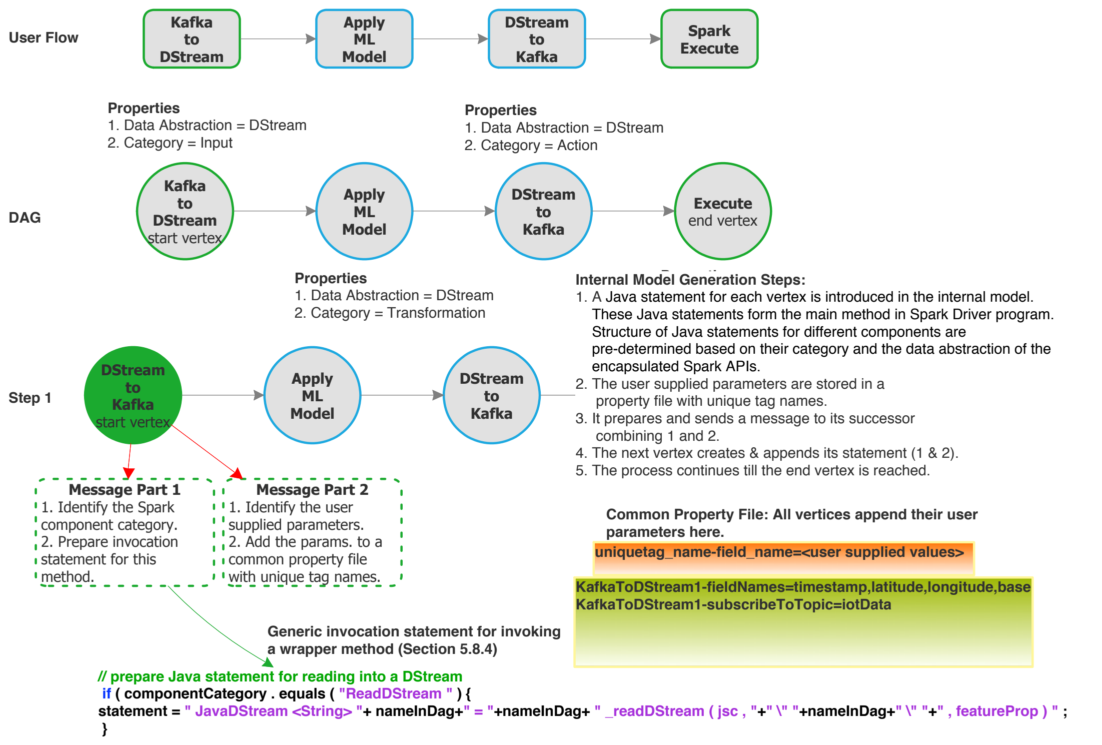

User Flow
Kafka to DStream
Apply ML Model
DStream to Kafka
Spark Execute
Properties
1. Data Abstraction = DStream
2. Category = Input
Properties
1. Data Abstraction = DStream
2. Category = Action
DAG
Kafka to DStream start vertex
Apply ML Model
DStream to Kafka
Execute end vertex
Properties
1. Data Abstraction = DStream
2. Category = Transformation
Properties
1. Category = Execute
Step 1
Step 2
Step 3
Step 4
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.

Figure A.1: Validation of use case 2 Spark flow: applying model to Streaming data: Steps



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.

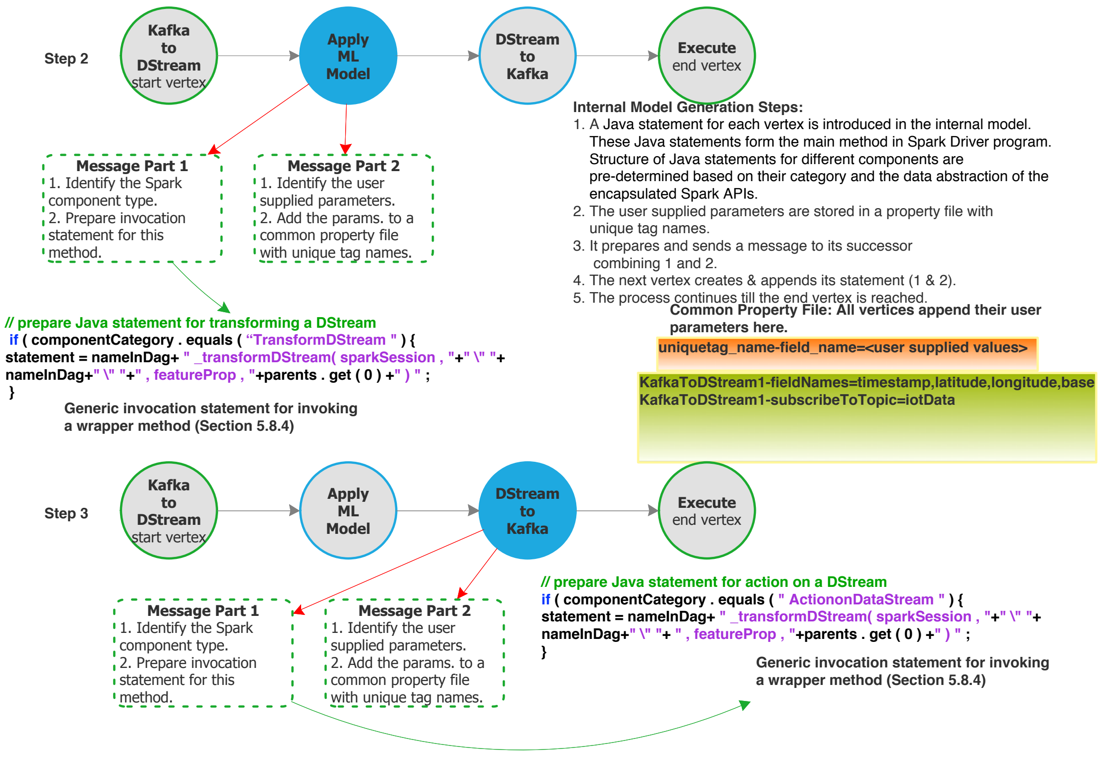
Common Property File: All vertices append their user parameters here.

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

```
KafkaToDStream1-fieldNames=timestamp,latitude,longitude,base
KafkaToDStream1-subscribeToTopic=iotData
```

```
// prepare Java statement for reading into a DStream
 if ( componentCategory . equals ( "ReadDStream " ) {
statement = " JavaDStream <String> "+ nameInDag+" = "+nameInDag+ " _readDStream ( jsc , "+" \" "+nameInDag+" \" "+" , featureProp ) " ;
 }
```

Figure A.2: Internal model representation of use case 2 Spark flow: Steps. (contd. on Figure A.3)


Step 2
Kafka to DStream start vertex
Apply ML Model
DStream to Kafka
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.
Step 3
Kafka to DStream start vertex
Apply ML Model
DStream to Kafka
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.

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.

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

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

```
KafkaToDStream1-fieldNames=timestamp,latitude,longitude,base
KafkaToDStream1-subscribeToTopic=iotData
```

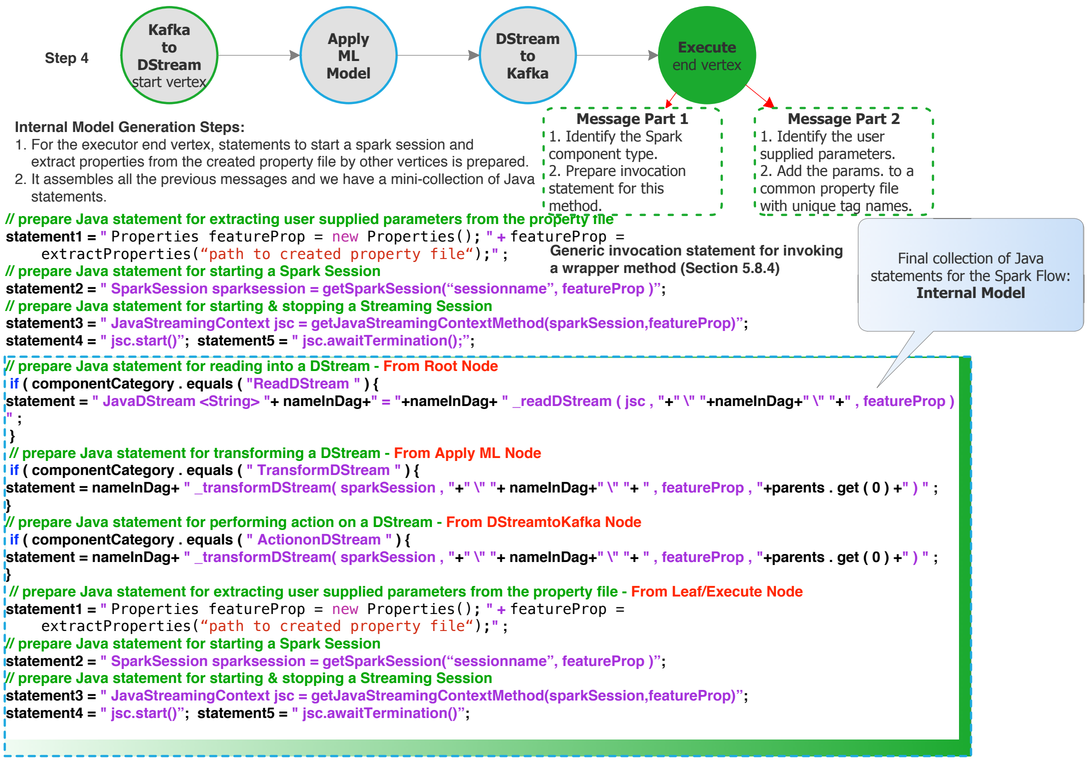
```
// prepare Java statement for transforming a DStream
 if ( componentCategory . equals ( “TransformDStream " ) {
statement = nameInDag+ " _transformDStream( sparkSession , "+" \" "+
nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
 }
```

Generic invocation statement for invoking a wrapper method (Section 5.8.4)

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

Generic invocation statement for invoking a wrapper method (Section 5.8.4)

Figure A.3: Internal model representation of use case 2 Spark flow: Steps. (contd. from Figure A.2, contd. on Figure A.4)


Step 4
Kafka to DStream start vertex
Apply ML Model
DStream to Kafka
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.
Final collection of Java statements for the Spark Flow: Internal Model

Internal Model Generation Steps:

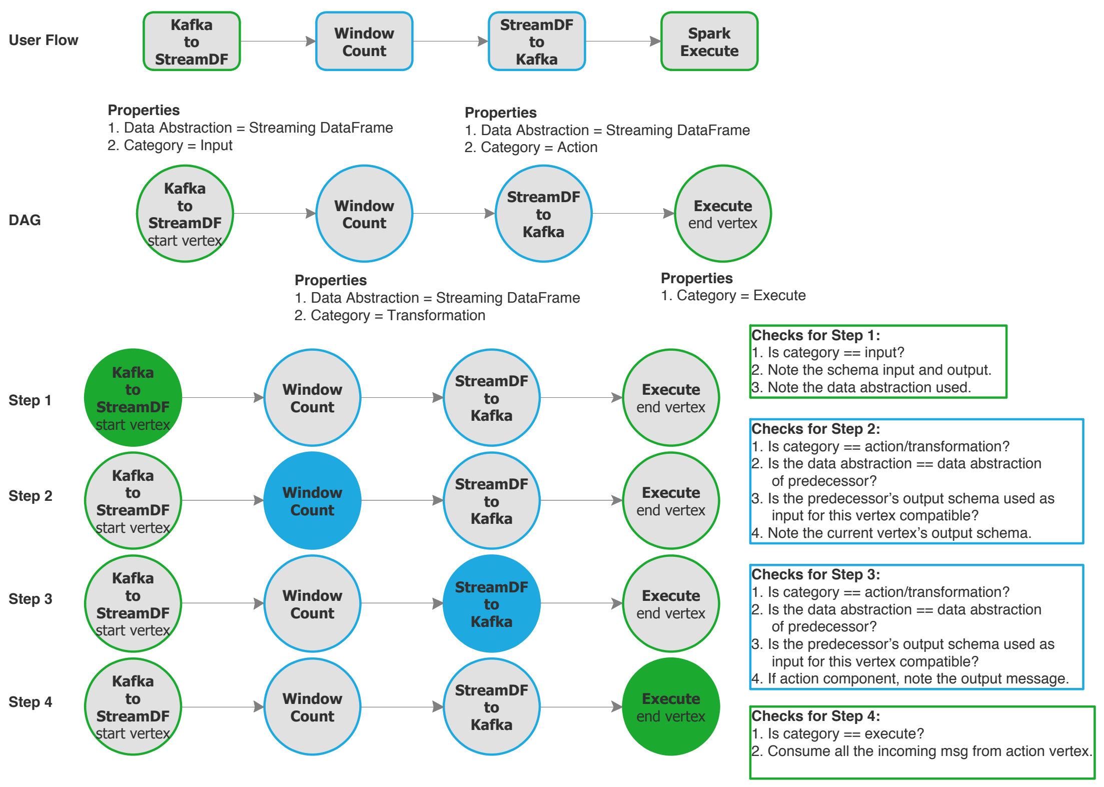
1. For the executor end vertex, statements to start a spark session and extract properties from the created property file by other vertices is prepared.
2. It assembles all the previous messages and we have a mini-collection of Java statements.

```
// 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 starting & stopping a Streaming Session
statement3 = " JavaStreamingContext jsc = getJavaStreamingContextMethod(sparkSession,featureProp)”;
statement4 = " jsc.start()”;  statement5 = " jsc.awaitTermination();”;
```

Generic invocation statement for invoking a wrapper method (Section 5.8.4)

```
// prepare Java statement for reading into a DStream - From Root Node
 if ( componentCategory . equals ( "ReadDStream " ) {
statement = " JavaDStream <String> "+ nameInDag+" = "+nameInDag+ " _readDStream ( jsc , "+" \" "+nameInDag+" \" "+" , featureProp )
" ;
 }
 // prepare Java statement for transforming a DStream - From Apply ML Node
 if ( componentCategory . equals ( " TransformDStream " ) {
statement = nameInDag+ " _transformDStream( sparkSession , "+" \" "+ nameInDag+" \" "+ " , featureProp , "+parents . get ( 0 ) +" ) " ;
}
// prepare Java statement for performing action on a DStream - From DStreamtoKafka Node
 if ( componentCategory . equals ( " ActiononDStream " ) {
statement = nameInDag+ " _transformDStream( sparkSession , "+" \" "+ nameInDag+" \" "+ " , featureProp , "+parents . get ( 0 ) +" ) " ;
}
 // prepare Java statement for extracting user supplied parameters from the property file - From Leaf/Execute Node
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 starting & stopping a Streaming Session
statement3 = " JavaStreamingContext jsc = getJavaStreamingContextMethod(sparkSession,featureProp)”;
statement4 = " jsc.start()”;  statement5 = " jsc.awaitTermination()”;
```

Figure A.4: Internal model representation of use case 2 Spark flow: Steps. (contd. from Figure A.3)



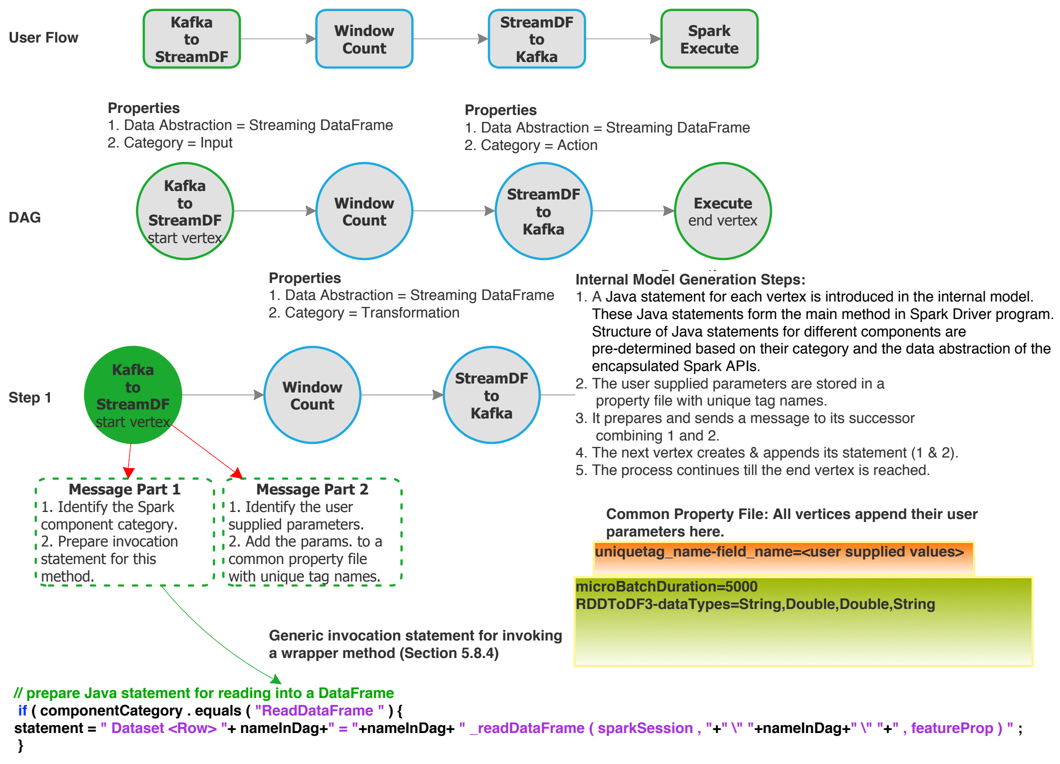
Checks for Step 1:
1. Is category == input?
2. Note the schema input and output.
3. Note the data abstraction used.

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.

Figure A.5: Validation of use case 3 Spark flow: performing streaming aggregations: Steps


User Flow
Kafka to StreamDF
Window Count
StreamDF to Kafka
Spark Execute
Properties
1. Data Abstraction = Streaming DataFrame
2. Category = Input
Properties
1. Data Abstraction = Streaming DataFrame
2. Category = Action
DAG
Kafka to StreamDF start vertex
Window Count
StreamDF to Kafka
Execute end vertex
Properties
1. Data Abstraction = Streaming DataFrame
2. Category = Transformation
Step 1
Generic invocation statement for invoking a wrapper method (Section 5.8.4)

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.

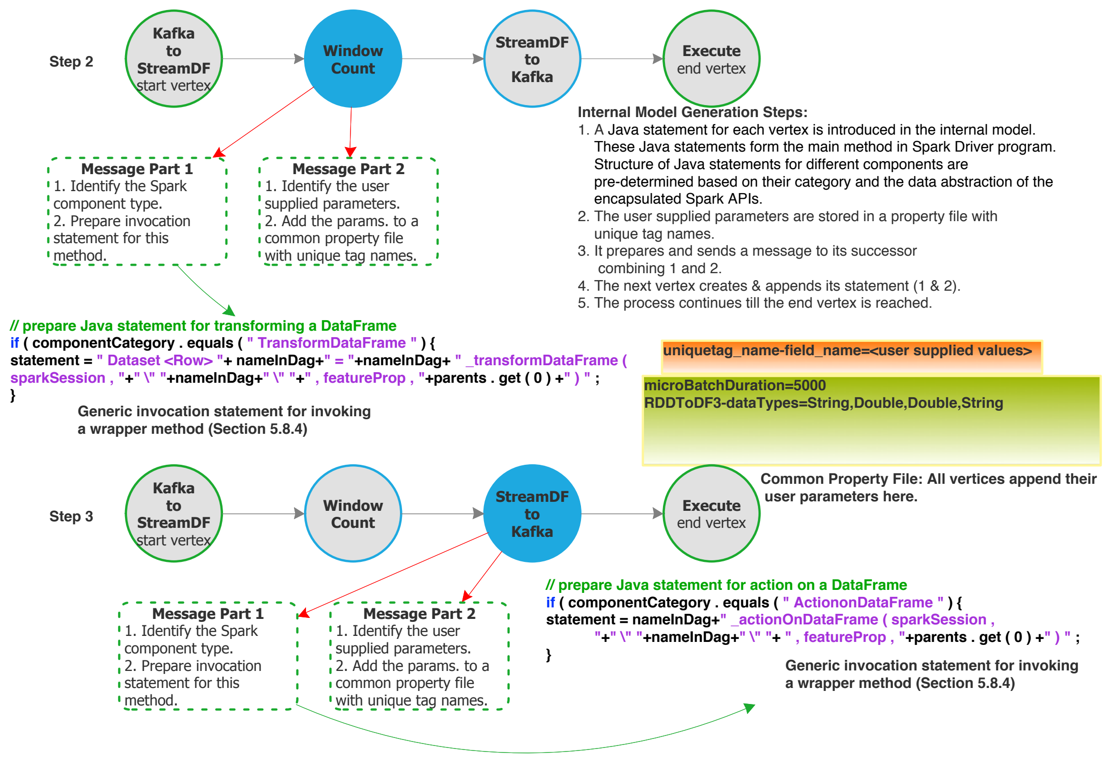
Common Property File: All vertices append their user parameters here.

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

```
microBatchDuration=5000
RDDToDF3-dataTypes=String,Double,Double,String
```

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

Figure A.6: Internal model representation of use case 3 Spark flow: Steps. (contd. on Figure A.7)


Step 2
Kafka to StreamDF start vertex
Window Count
StreamDF to Kafka
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 (Section 5.8.4)
Common Property File: All vertices append their user parameters here.
Step 3
Kafka to StreamDF start vertex
Window Count
StreamDF to Kafka
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 (Section 5.8.4)

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.

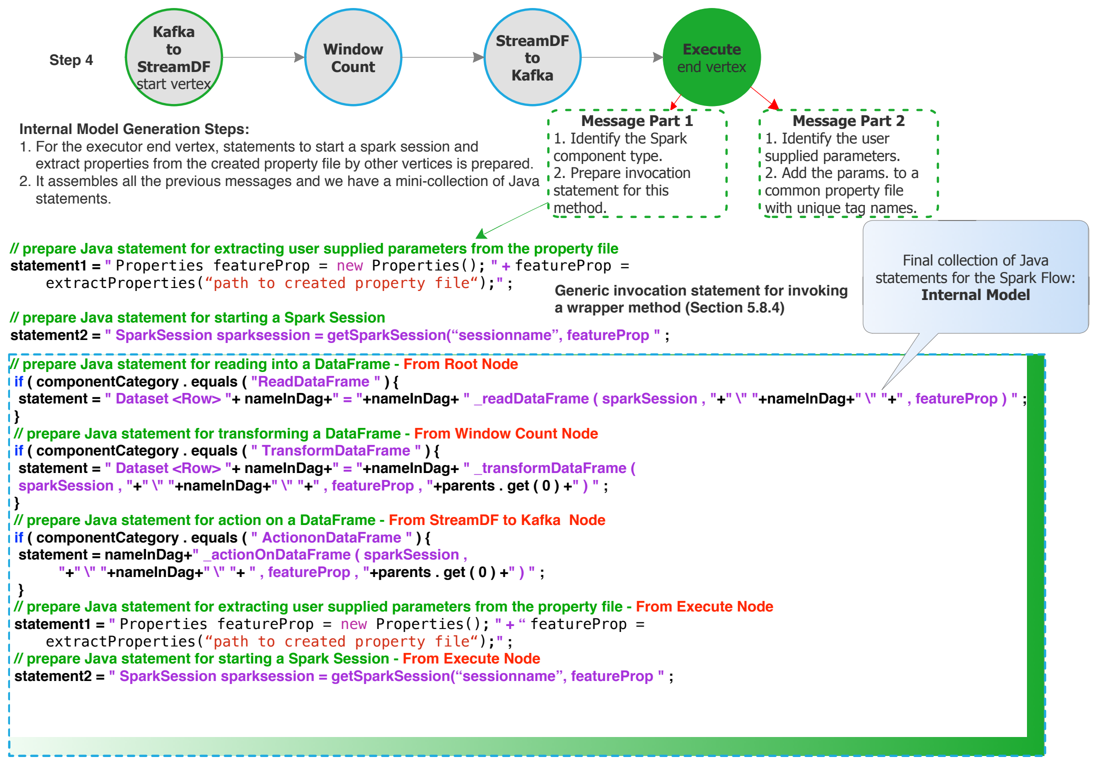
```
// prepare Java statement for transforming a DataFrame
if ( componentCategory . equals ( " TransformDataFrame " ) {
statement = " Dataset <Row> "+ nameInDag+" = "+nameInDag+ " _transformDataFrame (
sparkSession , "+" \" "+nameInDag+" \" "+" , featureProp , "+parents . get ( 0 ) +" ) " ;
}
```

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

```
microBatchDuration=5000
RDDToDF3-dataTypes=String,Double,Double,String
```

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

Figure A.7: Internal model representation of use case 3 Spark flow: Steps. (contd. from Figure A.2, contd. on Figure A.8)


Step 4
Kafka to StreamDF start vertex
Window Count
StreamDF to Kafka
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 (Section 5.8.4)
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 Window Count 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 StreamDF to Kafka  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 " ;
```

Figure A.8: Internal model representation of use case 3 Spark flow: Steps. (contd. from Figure A.3)

B Appendix for Flink

B.1 Flink Code Listings

```java
DataStream<TrafficObservation> filteredTraffic = trafficObservationDataStream.
    filter(new FilterFunction<TrafficObservation>() {
  @Override
  public boolean filter(TrafficObservation input) throws Exception {
    final double EARTH_RADIUS = 6371;
    double refLat = 43.462197; // Enter desired latitude here
    double refLng = -3.810048; // Enter desired longitude here
    double radius = 1; // Enter desired radius here
    double currentLat = input.getLatitude();
    double currentLng = input.getLongitude();
    double dLat = Math.toRadians(refLat - currentLat);
    double dLng = Math.toRadians(refLng - currentLng);
    double sindLat = Math.sin(dLat / 2);
    double sindLng = Math.sin(dLng / 2);
    double va1 = Math.pow(sindLat, 2) + Math.pow(sindLng, 2)* Math.cos(Math.
    toRadians(currentLat)) * Math.cos(Math.toRadians(refLat));
    double va2 = 2 * Math.atan2(Math.sqrt(va1), Math.sqrt(1 - va1));
    double distance = EARTH_RADIUS * va2;
    return distance < radius;
  }

});
```

Listing B.1: Auto-generated codes for data filtering via the GPS component

```java
AfterMatchSkipStrategy strat = AfterMatchSkipStrategy.noSkip();
  Pattern<TrafficObservation, TrafficObservation> myPattern = Pattern.<
    TrafficObservation>begin("start", strat)
    .where(new SimpleCondition<TrafficObservation>() {
        @Override
        public boolean filter(TrafficObservation trafficObservation) throws
    Exception {
            if (trafficObservation.getCharge() >= 50)
                return true;
            return false;
        }
      }).followedBy("middle")
      .where(new SimpleCondition<TrafficObservation>() {
      @Override
      public boolean filter(TrafficObservation trafficObservation) throws
    Exception {
          if (trafficObservation.getCharge() >= 60)
            return true;
          return false;
        }
    }).within(Time.minutes(10))
    .followedBy("end").where(new SimpleCondition<TrafficObservation>() {
      @Override
      public boolean filter(TrafficObservation trafficObservation) throws
    Exception {
        if (trafficObservation.getCharge() >= 75)
          return true;
        return false;
      }
    }).within(Time.minutes(10));

  PatternStream<TrafficObservation> patternStream = CEP.pattern(filteredTraffic
    , myPattern);

  DataStream<SmartSantanderAlert> alerts = patternStream.select(new
    PatternSelectFunction<TrafficObservation, SmartSantanderAlert>() {
    @Override
    public SmartSantanderAlert select(Map<String, List<TrafficObservation>> map
    ) throws Exception {
      TrafficObservation event = map.get("end").get(0);
      return new SmartSantanderAlert("Charge went too high in " + event.
    toString());
    }
  });
```

Listing B.2: Auto-generated code for pattern detection in real-time data

Bibliography

[1] D. J. Abadi et al. “Aurora: A New Model and Architecture for Data Stream Management”. In: *The VLDB Journal* 12.2 (Aug. 2003), pp. 120–139. ISSN: 1066-8888. DOI: 10.1007/s00778-003-0095-z. URL: http://dx.doi.org/10.1007/s00778-003-0095-z.

[2] T. Abdelzaher et al. “Introduction to Control Theory And Its Application to Computing Systems”. In: *Performance Modeling and Engineering*. Ed. by Z. Liu and C. H. Xia. Boston, MA: Springer US, 2008, pp. 185–215. ISBN: 978-0-387-79361-0. DOI: 10.1007/978-0-387-79361-0_7. URL: https://doi.org/10.1007/978-0-387-79361-0_7.

[3] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[4] G. A. Agha et al. “A Foundation for Actor Computation”. In: *J. Funct. Program.* 7.1 (Jan. 1997), pp. 1–72. ISSN: 0956-7968. DOI: 10.1017/S095679689700261X. URL: http://dx.doi.org/10.1017/S095679689700261X.

[5] J. Agrawal et al. “Efficient Pattern Matching over Event Streams”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 147–160. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376634. URL: http://doi.acm.org/10.1145/1376616.1376634.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: http://www.worldcat.org/oclc/12285707.

[7] Akka. *Implementation of the Actor Model. Build powerful reactive, concurrent, and distributed applications more easily*. [Online; accessed 25-December-2017]. URL: https://akka.io/.

[8] K. Akpinar, K. A. Hua, and K. Li. “ThingStore: A Platform for Internet-of-things Application Development and Deployment”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS ’15. Oslo, Norway: ACM, 2015, pp. 162–173. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2771833. URL: http://doi.acm.org/10.1145/2675743.2771833.

[9] M. Al-Kateb et al. “Hybrid Row-column Partitioning in Teradata”. In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1353–1364. ISSN: 2150-8097. DOI: 10.14778/3007263.3007273. URL: https://doi.org/10.14778/3007263.3007273.

[10] Amazon. *Amazon Kinesis*. [Online; accessed 24-April-2018]. URL: https://aws.amazon.com/kinesis/.

[11] Apache. *Apache Apex: Enterprise-grade unified stream and batch processing engine.* [Online; accessed 27-May-2019]. URL: https://apex.apache.org.

[12] Apache. *Apache Beam: An advanced unified programming model*. [Online; accessed 27-May-2019]. URL: https://beam.apache.org.

[13] Apache. *Apache Hadoop YARN*. [Online; accessed 27-May-2019]. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[14] Apache. *Apache Storm*. [Online; accessed 27-May-2019]. URL: https://storm.apache.org.

[15] Apache. *Beam Capability Matrix*. [Online; accessed 05-June-2019]. URL: https://beam.apache.org/documentation/runners/capability-matrix/.

[16] Apache. *Flink Programming Concepts*. [Online; accessed 09-May-2019]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/programming-model.html.

[17] Apache. *FlinkCEP - Complex event processing for Flink*. [Online; accessed 05-June-2019]. URL: https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html.

[18] Apache. *GraphX: GraphX is Apache Spark's API for graphs and graph-parallel computation*. [Online; accessed 27-May-2019]. URL: https://spark.apache.org/graphx/.

[19] Apache. *NiFi*. [Online; accessed 05-September-2018]. URL: https://nifi.apache.org/docs.html.

[20] Apache. *The Apache Hive data warehouse software*. [Online; accessed 27-May-2019]. 2008. URL: https://hive.apache.org.

[21] Apache. *Flume*. [Online; accessed 27-May-2019]. 2017. URL: https://flume.apache.org.

[22] Apache. *Apache Hadoop*. [Online; accessed 26-November-2018]. 2018. URL: http://hadoop.apache.org.

[23] Apache. *Hadoop MapReduce Tutorial*. [Online; accessed 26-November-2018]. 2018. URL: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[24] Apache. *HDFS Architecture Guide*. [Online; accessed 26-November-2018]. 2018. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[25] Apache. *Spark Streaming Programing Guide*. [Online; accessed 12-December-2018]. 2018. URL: https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html/.

[26] Apache. *Spark Structured Streaming Programming Guide*. [Online; accessed 12-December-2018]. 2018. URL: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[27] Apache. *Pig*. [Online; accessed 27-May-2019]. 2019. URL: https://pig.apache.org/.

[28] Apache Kafka. *A Distributed Streaming Platform*. [Online; accessed 24-April-2019]. 2018. URL: https://kafka.apache.org.

[29] Apache Spark. *Cluster Modes*. [Online; accessed 24-April-2018]. URL: https://spark.apache.org/docs/latest/cluster-overview.html.

[30] Apache Spark. *Spark SQL, DataFrames and Datasets Guide*. [Online; accessed 24-April-2018]. URL: https://spark.apache.org/docs/latest/sql-programming-guide.html.

[31] Apache Spark. *RDD Programming Guide*. [Online; accessed 16-May-2019]. 2019. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[32] *Apache Zeppelin*. [Online; accessed 22-June-2018]. URL: https://zeppelin.apache.org/docs/0.7.0/.

[33] A. Arasu, S. Babu, and J. Widom. “The CQL Continuous Query Language: Semantic Foundations and Query Execution”. In: *The VLDB Journal* 15.2 (June 2006), pp. 121–142. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. URL: http://dx.doi.org/10.1007/s00778-004-0147-z.

[34] M. Armbrust et al. “Spark SQL: Relational Data Processing in Spark.” In: *SIGMOD Conference*. Ed. by T. K. Sellis, S. B. Davidson, and Z. G. Ives. ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: https://doi.org/10.1145/2723372.2742797.

[35] C. Attiogbé, P. André, and G. Ardourel. “Checking Component Composability”. In: *Software Composition*. Ed. by W. Löwe and M. Südholt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 18–33. ISBN: 978-3-540-37659-0.

[36] L. Atzori, A. Iera, and G. Morabito. “The Internet of Things: A survey”. In: *Computer Networks* 54.15 (2010), pp. 2787 –2805. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2010.05.010. URL: http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[37] C. Y. Baldwin and K. B. Clark. “Modularity in the Design of Complex Engineering Systems”. In: *Complex Engineered Systems: Science Meets Technology*. Ed. by D. Braha, A. A. Minai, and Y. Bar-Yam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 175–205. ISBN: 978-3-540-32834-6. DOI: 10.1007/3-540-32834-3_9. URL: https://doi.org/10.1007/3-540-32834-3_9.

[38] A. Balmin et al. “Adaptive Processing of User-Defined Aggregates in Jaql”. In: *IEEE Data Eng. Bull.* 34.4 (2011), pp. 36–43. URL: http://sites.computer.org/debull/A11dec/adaptivemr2.pdf.

[39] R. S. Barga et al. “Consistent Streaming Through Time: A Vision for Event Stream Processing”. In: *CIDR*. 2007, pp. 363–373.

[40] D. Bau et al. “Learnable Programming: Blocks and Beyond”. In: *Commun. ACM* 60.6 (May 2017), pp. 72–80. ISSN: 0001-0782. DOI: 10.1145/3015455. URL: http://doi.acm.org/10.1145/3015455.

[41] G. Berry and G. Gonthier. “The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation”. In: *Sci. Comput. Program.* 19.2 (Nov. 1992), pp. 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V. URL: http://dx.doi.org/10.1016/0167-6423(92)90005-V.

[42] K. S. Beyer et al. “Jaql: A Scripting Language for Large Scale Semistructured Data Analysis”. In: *PVLDB* 4.12 (2011), pp. 1272–1283. URL: http://www.vldb.org/pvldb/vol4/p1272-beyer.pdf.

[43] M. Blackstock and R. Lea. "WoTKit: A Lightweight Toolkit for the Web of Things". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. Newcastle, United Kingdom: ACM, 2012, 3:1–3:6. ISBN: 978-1-4503-1603-3. DOI: 10.1145/2379756.2379759. URL: http://doi.acm.org/10.1145/2379756.2379759.

[44] B. Calder. "Inside Windows Azure: The Challenges and Opportunities of a Cloud Operating System". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 1–2. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2560008. URL: http://doi.acm.org/10.1145/2541940.2560008.

[45] B. Calder. "Inside Windows Azure: The Challenges and Opportunities of a Cloud Operating System". In: *SIGARCH Comput. Archit. News* 42.1 (Feb. 2014), pp. 1–2. ISSN: 0163-5964. DOI: 10.1145/2654822.2560008. URL: http://doi.acm.org/10.1145/2654822.2560008.

[46] B. Calder. "Inside Windows Azure: The Challenges and Opportunities of a Cloud Operating System". In: *SIGPLAN Not.* 49.4 (Feb. 2014), pp. 1–2. ISSN: 0362-1340. DOI: 10.1145/2644865.2560008. URL: http://doi.acm.org/10.1145/2644865.2560008.

[47] F. J. Cangialosi et al. "The Design of the Borealis Stream Processing Engine". In: *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. Asilomar, CA, 2005.

[48] M. Carkci. *Dataflow and Reactive Programming Systems: A Practical Guide*. 1st ed. USA: CreateSpace Independent Publishing Platform, 2014. ISBN: 1497422442, 9781497422445.

[49] J. Catozzi and S. Rabinovici. "OS Support for VLDBs: Unix Enhancements for the Teradata Data Base". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 696–701. ISBN: 1-55860-379-4. URL: http://dl.acm.org/citation.cfm?id=645921.673297.

[50] J. Catozzi and S. Rabinovici. "Operating System Extensions for the Teradata Parallel VLDB". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 679–682. ISBN: 1-55860-804-4. URL: http://dl.acm.org/citation.cfm?id=645927.672210.

[51] C. Cecchinel et al. "An Architecture to Support the Collection of Big Data in the Internet of Things". In: *IEEE World Congress on Services*. 2014, pp. 442–449. DOI: 10.1109/SERVICES.2014.83.

[52] U. Çetintemel et al. "The Aurora and Borealis Stream Processing Engines". In: *Data Stream Management: Processing High-Speed Data Streams*. Ed. by M. Garofalakis, J. Gehrke, and R. Rastogi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 337–359. ISBN: 978-3-540-28608-0. DOI: 10.1007/978-3-540-28608-0_17. URL: https://doi.org/10.1007/978-3-540-28608-0_17.

[53] S. Chandrasekaran et al. "TelegraphCQ: Continuous Dataflow Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 668–668. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872857. URL: http://doi.acm.org/10.1145/872757.872857.

[54] J. Chen et al. "NiagaraCQ: A Scalable Continuous Query System for Internet Databases". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: ACM, 2000, pp. 379–390. ISBN: 1-58113-217-4. DOI: 10.1145/342009.335432. URL: http://doi.acm.org/10.1145/342009.335432.

[55] M. Chen, S. Mao, and Y. Liu. "Big Data: A Survey". In: *Mobile Networks and Applications* 19.2 (Apr. 2014), pp. 171–209. ISSN: 1572-8153. DOI: 10.1007/s11036-013-0489-0. URL: https://doi.org/10.1007/s11036-013-0489-0.

[56] S. Cirani et al. "IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios". In: *IEEE Sensors Journal* 15.2 (2015), pp. 1224–1234. ISSN: 1530-437X. DOI: 10.1109/JSEN.2014.2361406.

[57] M. F. Costabile et al. "Building environments for end-user development and tailoring." In: *HCC*. IEEE Computer Society, 2003, pp. 31–38. ISBN: 0-7803-8225-0. URL: http://dblp.uni-trier.de/db/conf/vl/hcc2003.html#CostabileFFMP03.

[58] K. N. Cukier and V. Mayer-Schoenberger. "The Rise of Big Data". In: *Foreign Affairs* (2013). URL: https://www.foreignaffairs.com/articles/2013-04-03/rise-big-data.

[59] N. Cunniff and R. P. Taylor. "Empirical Studies of Programmers: Second Workshop". In: ed. by G. M. Olson, S. Sheppard, and E. Soloway. Norwood, NJ, USA: Ablex Publishing Corp., 1987. Chap. Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension, pp. 114–131. ISBN: 0-89391-461-4. URL: http://dl.acm.org/citation.cfm?id=54968.54976.

[60] J. Damji. *A Tale of three Spark APIs*. [Online; accessed 24-December-2018]. URL: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.

[61] F. Daniel and M. Matera. *Mashups: Concepts, Models and Architectures*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-55048-5 978-3-642-55049-2. (Visited on 02/25/2016).

[62] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[63] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[64] F. C. Delicato et al. "Towards an IoT Ecosystem". In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS '13. Montpellier, France: ACM, 2013, pp. 25–28. ISBN: 978-1-4503-2048-1. DOI: 10.1145/2489850.2489855. URL: http://doi.acm.org/10.1145/2489850.2489855.

[65] F. DeRemer and H. H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small". In: *IEEE Transactions on Software Engineering* SE-2.2 (1976), pp. 80–86. ISSN: 0098-5589.

[66] H. Derhamy et al. "A survey of commercial frameworks for the Internet of Things". In: *ETFA*. 2015, pp. 1–8. doi: 10.1109/ETFA.2015.7301661.

[67] T. Desell, K. E. Maghraoui, and C. A. Varela. "Malleable applications for scalable high performance computing". In: *Cluster Computing* 10.3 (2007), pp. 323–337. issn: 1573-7543. doi: 10.1007/s10586-007-0032-9. url: https://doi.org/10.1007/s10586-007-0032-9.

[68] G. M. D'silva et al. "Real-time processing of IoT events with historic data using Apache Kafka and Apache Spark with dashing framework". In: *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. May 2017, pp. 1804–1809. doi: 10.1109/RTEICT.2017.8256910.

[69] T. Dunning and E. Friedman. *Streaming architecture: new designs using Apache Kafka and MapR streams*. " O'Reilly Media, Inc.", 2016.

[70] H. Eichelberger, C. Qin, and K. Schmid. "Experiences with the Model-based Generation of Big Data Pipelines". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband*. Ed. by B. Mitschang et al. Bonn: Gesellschaft für Informatik e.V., 2017, pp. 49–56.

[71] R. C. Fernandez et al. "Liquid: Unifying Nearline and Offline Big Data Integration". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. 2015. url: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf.

[72] A. Filieri et al. "Control Strategies for Self-Adaptive Software Systems". In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (Feb. 2017), 24:1–24:31. issn: 1556-4665. doi: 10.1145/3024188. url: http://doi.acm.org/10.1145/3024188.

[73] M. P. Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[74] E. Friedman and K. Tzoumas. *Introduction to Apache Flink*. O'Reilly, Sept. 2016.

[75] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[76] B. Gedik et al. "SPADE: The System S Declarative Stream Processing Engine". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1123–1134. isbn: 978-1-60558-102-6. doi: 10.1145/1376616.1376729. url: http://doi.acm.org/10.1145/1376616.1376729.

[77] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 29–43. issn: 0163-5980. doi: 10.1145/1165389.945450. url: http://doi.acm.org/10.1145/1165389.945450.

[78] J. Goodwin. *Learning Akka*. Packt Publishing, 2015. url: https://www.packtpub.com/application-development/learning-akka.

[79] Google. *Blockly: A JavaScript library for building visual programming editors*. [Online; accessed 27-May-2019]. url: https://developers.google.com/blockly/.

[80] Google. *BigQuery*. [Online; accessed 27-May-2019]. 2010. url: https://cloud.google.com/bigquery/.

[81] M. Gorlick and A. Quilici. "Visual programming-in-the-large versus visual programming-in-the-small". In: *Proceedings of 1994 IEEE Symposium on Visual Languages*. 1994, pp. 137–144. doi: 10.1109/VL.1994.363631.

[82] T. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131 –174. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.1996.0009.

[83] A. Gyrard et al. "Cross-Domain Internet of Things Application Development: M3 Framework and Evaluation". In: *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. 2015, pp. 9–16. DOI: 10.1109/FiCloud.2015.10.

[84] N. Health. *How IBM's Node-RED is hacking together the internet of things*. Ed. by TechRepublic.com. [Online; posted 13-March-2014]. 2014. URL: http://www.techrepublic.com/article/node-red/.

[85] S. Heo et al. "IoT-MAP: IoT mashup application platform for the flexible IoT ecosystem". In: *Internet of Things (IOT), 2015 5th International Conference on the*. 2015, pp. 163–170. DOI: 10.1109/IOT.2015.7356561.

[86] C. Hewitt. "Viewing Control Structures As Patterns of Passing Messages". In: *Artif. Intell.* 8.3 (June 1977), pp. 323–364. ISSN: 0004-3702. DOI: 10.1016/0004-3702(77)90033-9. URL: http://dx.doi.org/10.1016/0004-3702(77)90033-9.

[87] C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[88] B. Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[89] M. Hirzel et al. "IBM Streams Processing Language: Analyzing Big Data in motion". In: *IBM Journal of Research and Development* 57.3/4 (2013), 7:1–7:11. ISSN: 0018-8646. DOI: 10.1147/JRD.2013.2243535.

[90] R. Holwerda and F. Hermans. "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. 217–225. DOI: 10.1109/VLHCC.2018.8506483.

[91] IBM. *IBM SPSS Modeller*. [Online; accessed 22-June-2018]. URL: https://www.ibm.com/products/spss-modeler.

[92] IBM. *Node-RED, Flow-based programming for the Internet of Things*. [Online; accessed 10-May-2016]. URL: http://nodered.org/.

[93] M. H. Iqbal and T. R. Soomro. "Big data analysis: Apache storm perspective". In: *International journal of computer trends and technology* 19.1 (2015), pp. 9–14.

[94] *JavaPoet*. [Online; accessed 24-April-2018]. URL: https://github.com/square/javapoet/blob/master/README.md.

[95] P. Johansson and H. Holmberg. *On the Modularity of a System*. 2010.

[96] A. Katsifodimos and S. Schelter. "Apache Flink: Stream Analytics at Scale". In: *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. Apr. 2016, pp. 193–193. DOI: 10.1109/IC2EW.2016.56.

[97] Keras. *Keras: The Python Deep Learning library*. [Online; accessed 27-May-2019]. URL: https://keras.io.

[98] J. Kim and J. W. Lee. "OpenIoT: An open service framework for the Internet of Things". In: *Internet of Things (WF-IoT)*. 2014, pp. 89–93. DOI: 10.1109/WF-IoT.2014.6803126.

[99] M. Kiran et al. "Lambda architecture for cost-effective batch and speed big data processing". In: *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE. 2015, pp. 2785–2792.

[100] R. Kleinfeld et al. "glue.things: A Mashup Platform for wiring the Internet of Things with the Internet of Services". In: *Proceedings of the 5th International Workshop on Web of Things*. WoT '14. ACM, 2014, pp. 16–21. ISBN: 978-1-4503-3066-4. DOI: 10.1145/2684432.2684436. URL: http://doi.acm.org/10.1145/2684432.2684436 (visited on 05/11/2016).

[101] D. Krajzewicz et al. "Recent Development and Applications of SUMO - Simulation of Urban MObility". In: *International Journal On Advances in Systems and Measurements* 5.3&4 (2012), pp. 128–138.

[102] J. Kreps, N. Narkhede, and J. Rao. "Kafka: a Distributed Messaging System for Log Processing". In: 2011.

[103] B. W. Lampson. "Lazy and Speculative Execution in Computer Systems". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: ACM, 2008, pp. 1–2. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411205. URL: http://doi.acm.org/10.1145/1411204.1411205.

[104] Y. Li, T. Tan, and J. Xue. "Understanding and Analyzing Java Reflection". In: *ACM Trans. Softw. Eng. Methodol.* 28.2 (Feb. 2019), 7:1–7:50. ISSN: 1049-331X. DOI: 10.1145/3295739. URL: http://doi.acm.org/10.1145/3295739.

[105] Linkedin. [Online; accessed 05-June-2019]. URL: https://www.linkedin.com.

[106] B. Livshits, J. Whaley, and M. S. Lam. "Reflection Analysis for Java". In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 139–160. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_11. URL: http://dx.doi.org/10.1007/11575467_11.

[107] D. Ma. "Offering RSS Feeds: Does It Help to Gain Competitive Advantage?" In: *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*. 2009, pp. 1–10. DOI: 10.1109/HICSS.2009.327.

[108] T. Mahapatra et al. "Graphical Spark Programming in IoT Mashup Tools". In: *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*. 2018, pp. 163–170. DOI: 10.1109/IoTSMS.2018.8554665.

[109] T. Mahapatra et al. "Stream Analytics in IoT Mashup Tools". In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 227–231. DOI: 10.1109/VLHCC.2018.8506548.

[110] T. Mahapatra, I. Gerostathopoulos, and C. Prehofer. "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. Stuttgart, Germany: ACM, 2016, pp. 11–16. ISBN: 978-1-4503-4874-4. DOI: 10.1145/3017995.3017998. URL: http://doi.acm.org/10.1145/3017995.3017998.

[111] T. Mahapatra and C. Prehofer. "Service Mashups and Developer Support". In: *Digital Mobility Platforms and Ecosystems* (2016), pp. 48 –65. DOI: 10 . 14459 / 2016md1324021.

[112] T. Mahapatra et al. "Designing Flink Pipelines in IoT Mashup Tools". In: 2316.03 (2018), pp. 41–53. URL: http://ceur-ws.org/Vol-2316/paper3.pdf.

[113] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st. Greenwich, CT, USA: Manning Publications Co., 2015. ISBN: 1617290343, 9781617290343.

[114] D. Mason and K. Dave. "Block-based versus flow-based programming for naive programmers". In: *2017 IEEE Blocks and Beyond Workshop (B B)*. 2017, pp. 25–28. DOI: 10.1109/BLOCKS.2017.8120405.

[115] X. Meng et al. "MLlib: Machine Learning in Apache Spark." In: *CoRR* abs/1505.06807 (2015). URL: http://dblp.uni-trier.de/db/journals/corr/corr1505.html#MengBYSVLFTAOXX15.

[116] Microsoft. *Microsoft Azure*. [Online; accessed 22-June-2018]. URL: https://docs.microsoft.com/en-us/azure/hdinsight/.

[117] S. Minni. *Apache Kafka Cookbook*. Packt Publishing Ltd, 2015.

[118] MIT. *SCRATCH*. [Online; accessed 27-May-2019]. URL: https://scratch.mit.edu.

[119] F. A. F. Moreno. "Modularizing Flink programs to enable stream analytics in IoT Mashup tools = Modularización de programas Flink para el análisis de datos en tiempo real en herramientas de Mashup para IOT". 2018. URL: http://oa.upm.es/52898/.

[120] J. P. Morrison. "Flow-based programming". In: *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*. 1994, pp. 25–29.

[121] J. P. Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Paramount, CA: CreateSpace, 2010. ISBN: 1451542321, 9781451542325.

[122] D. R. Musser and C. A. Varela. "Structured Reasoning About Actor Systems". In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 37–48. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541334. URL: http://doi.acm.org/10.1145/2541329.2541334.

[123] P. Neumann. "Principled Assuredly Trustworthy Compusable Architectures". In: *DARPA Final Report, SRI Project P11459, December* (Jan. 2004).

[124] M. A. Overton. "The IDAR Graph". In: *Commun. ACM* 60.7 (June 2017), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/3079970. URL: http://doi.acm.org/10.1145/3079970.

[125] pandas. *Python Data Analysis Library*. [Online; accessed 27-May-2019]. URL: https://pandas.pydata.org.

[126] C. Peltz. "Web services orchestration and choreography". In: *Computer* 36.10 (2003), pp. 46–52. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1236471.

[127] A. Pintus, D. Carboni, and A. Piras. “Paraimpu: a platform for a social web of things”. In: *Proceedings of the 21st international conference companion on World Wide Web*. ACM, 2012, pp. 401–404. URL: http://dl.acm.org/citation.cfm?id=2188059 (visited on 05/10/2016).

[128] F. Pramudianto et al. “IoT Link: An Internet of Things Prototyping Toolkit”. In: *Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)*. 2014, pp. 1–9. DOI: 10.1109/UIC-ATC-ScalCom.2014.95.

[129] C. Prehofer and L. Chiarabini. “From Internet of Things Mashups to Model-Based Development”. In: *COMPSAC, 2015 IEEE 39th Annual*. IEEE, July 2015, pp. 499–504. ISBN: 978-1-4419-1673-0.

[130] C. Prehofer and I. Gerostathopoulos. “Modeling RESTful Web of Things Services: Concepts and Tools”. In: *Managing the Web of Things*. 2017.

[131] C. Prehofer and D. Schinner. “Generic Operations on RESTful Resources in Mashup Tools”. In: *Proceedings of the 6th International Workshop on the Web of Things*. WoT ’15. Seoul, Republic of Korea: ACM, 2015, 3:1–3:6. ISBN: 978-1-4503-4045-8. DOI: 10.1145/2834791.2834795. URL: http://doi.acm.org/10.1145/2834791.2834795.

[132] A. Rao, A. Bihani, and M. Nair. “Milo: A visual programming environment for Data Science Education”. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 211–215. DOI: 10.1109/VLHCC.2018.8506504.

[133] A. Ryman. “Simple Object Access Protocol (SOAP) and Web Services”. In: *Proceedings of the 23rd International Conference on Software Engineering*. ICSE ’01. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 689–. ISBN: 0-7695-1050-7. URL: http://dl.acm.org/citation.cfm?id=381473.381580.

[134] B. Saha et al. “Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1357–1369. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742790. URL: http://doi.acm.org/10.1145/2723372.2742790.

[135] Santander City Council. *Santander Open Data - REST API Documentation*. [Online; accessed 01-June-2018]. 2018. URL: http://datos.santander.es/documentacion-api/.

[136] W. d. Santos et al. “Scalable and Efficient Data Analytics and Mining with Lemonade”. In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 2070–2073. ISSN: 2150-8097. DOI: 10.14778/3229863.3236262. URL: https://doi.org/10.14778/3229863.3236262.

[137] Scala. *The Scala Programming Language*. [Online; accessed 05-June-2019]. URL: https://www.scala-lang.org.

[138] S. Schmid, I. Gerostathopoulos, and C. Prehofer. “QryGraph: A Graphical Tool for Big Data Analytics”. In: *SMC’16*. 2016.

[139] R. Schutt and C. O’Neil. *Doing data science*. 2014.

[140] scikit-learn. *Machine Learning in Python*. [Online; accessed 27-May-2019]. URL: https://scikit-learn.org/stable/.

[141] SciPy.org. *NumPy*. [Online; accessed 27-May-2019]. URL: https://www.numpy.org.

[142] SciPy.org. *SciPy library*. [Online; accessed 27-May-2019]. URL: https://www.scipy.org.

[143] N. Seyfer, R. Tibbetts, and N. Mishkin. "Capture Fields: Modularity in a Stream-relational Event Processing Langauge". In: *Proceedings of the 5th ACM International Conference on Distributed Event-based System*. DEBS '11. New York, New York, USA: ACM, 2011, pp. 15–22. ISBN: 978-1-4503-0423-8. DOI: 10.1145/2002259.2002263. URL: http://doi.acm.org/10.1145/2002259.2002263.

[144] R. Soulé et al. "A Universal Calculus for Stream Processing Languages". In: *Programming Languages and Systems*. Ed. by A. D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 507–528. ISBN: 978-3-642-11957-6.

[145] K. G. Srinivasa and A. K. Muppalla. *Guide to High Performance Distributed Computing - Case Studies with Hadoop, Scalding and Spark*. Computer Communications and Networks. Springer, 2015.

[146] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. USA: John Wiley & Sons, Inc., 2006. ISBN: 0470025700.

[147] M. Stonebraker, U. Çetintemel, and S. Zdonik. "The 8 requirements of real-time stream processing". In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[148] C. L. Talcott. "Composable Semantic Models for Actor Theories". In: *Higher-Order and Symbolic Computation* 11.3 (1998), pp. 281–343. ISSN: 1573-0557. DOI: 10.1023/A:1010042915896. URL: https://doi.org/10.1023/A:1010042915896.

[149] D. Thangavel et al. "Performance evaluation of MQTT and CoAP via a common middleware". In: *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. 2014, pp. 1–6. DOI: 10.1109/ISSNIP.2014.6827678.

[150] The Apache Software Foundation. *Dataflow Programming Model, v1.5*. [Online; accessed 01-June-2018]. 2018. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/programming-model.html.

[151] *The Data Science Venn Diagram*. [Online; accessed 25-November-2018]. URL: https://s3.amazonaws.com/aws.drewconway.com/viz/venn_diagram/data_science.html.

[152] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Compiler Construction*. Ed. by R. N. Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 179–196. ISBN: 978-3-540-45937-8.

[153] Touk. *Nussknacker. Streaming Processes Diagrams*. [Online; accessed 27-May-2019]. URL: https://touk.github.io/nussknacker/.

[154] Twitter. *It's what's happening*. [Online; accessed 05-June-2019]. URL: https://twitter.com.

[155] C. Varela and G. Agha. "Programming Dynamically Reconfigurable Open Systems with SALSA". In: *SIGPLAN Not.* 36.12 (Dec. 2001), pp. 20–34. ISSN: 0362-1340. DOI: 10.1145/583960.583964. URL: http://doi.acm.org/10.1145/583960.583964.

[156] V. K. Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[157] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996. ISBN: 0-13-508301-X.

[158] Voldemort. *Project Voldemort*. [Online; accessed 27-May-2019]. URL: https://www.project-voldemort.com/voldemort/.

[159] J. Wang, Z. Xu, and J. Zhang. “Implementation Strategies for CSV Fragment Retrieval over HTTP”. In: *2015 12th Web Information System and Application Conference (WISA)*. 2015, pp. 223–228. DOI: 10.1109/WISA.2015.27.

[160] A. Wegener et al. “TraCI: An Interface for Coupling Road Traffic and Network Simulators”. In: *11th Communications and Networking Simulation Symposium (CNS)* (2008).

[161] T. White. *Hadoop: The Definitive Guide*. 1st. O’Reilly Media, Inc., 2009. ISBN: 0596521979, 9780596521974.

[162] K. N. Whitley, L. R. Novick, and D. Fisher. “Evidence in Favor of Visual Representation for the Dataflow Paradigm: An Experiment Testing LabVIEW’s Comprehensibility”. In: *Int. J. Hum.-Comput. Stud.* 64.4 (Apr. 2006), pp. 281–303. ISSN: 1071-5819. DOI: 10.1016/j.ijhcs.2005.06.005. URL: http://dx.doi.org/10.1016/j.ijhcs.2005.06.005.

[163] H. Wickham and G. Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st. O’Reilly Media, Inc., 2017. ISBN: 1491910399, 9781491910399.

[164] Wikipedia. *Composability*. [Online; accessed 27-May-2019]. URL: https://en.wikipedia.org/wiki/Composability.

[165] Wikipedia. *.properties article*. [Online; accessed 27-May-2019]. URL: https://en.wikipedia.org/wiki/.properties.

[166] Wikipedia. *The Free Encyclopedia*. [Online; accessed 05-June-2019]. URL: https://www.wikipedia.org.

[167] J. Yu et al. “Understanding Mashup Development”. In: *IEEE Internet Computing* 12.5 (Sept. 2008), pp. 44–52. ISSN: 1089-7801. DOI: 10.1109/MIC.2008.114. URL: http://dx.doi.org/10.1109/MIC.2008.114.

[168] M. Zaharia et al. “Spark: Cluster Computing with Working Sets.” In: *HotCloud*. Ed. by E. M. Nahum and D. Xu. USENIX Association, 2010. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[169] M. Zaharia et al. “Discretized Streams: Fault-Tolerant Streaming Computation at Scale.” In: *SOSP*. Ed. by M. Kaminsky and M. Dahlin. ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. URL: http://dblp.uni-trier.de/db/conf/sosp/sosp2013.html#ZahariaDLHSS13.

[170] M. Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing.” In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: https://doi.org/10.1145/2934664.

[171] P. Zecevic and M. Bonaci. *Spark in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2016. ISBN: 1617292605, 9781617292606.