FP7 Project ASAP

Adaptable Scalable Analytics Platform

ASAP D5.1 Workflow management model

WP 5 – Adaptive Data Analytics

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
1.0	18 Feb 2015	V. Kantere, M. Fialtov	First Version
2.0	27 Feb 2015	V. Kantere, M. Filatov	Final Version

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Contents

List of Figures

Chapter 1

Introduction

The analysis of Big Data is a core and critical task in multifarious domains of science and industry. Such analysis needs to be performed on a range of data stores, both traditional and modern, data sources that are heterogeneous in their schemas and formats, and on a diversity of query engines.

The users that need to perform such data analysis may have several roles, like, business analysts, engineers, end-users, scientists etc. Users with different roles may need different aspects of information deduced from the data. Therefore, the various users need to perform a variety of tasks, like simple or complex data operations and queries, data mining, algorithmic processing, text retrieval, data annotation, etc. Moreover, they may need to perform such tasks in different scheduling schemes, for example short or long-running queries in combinations with a one-time or a continuous output. Finally, the users may differ in their expertise with respect to their data management skills, as well as on their interest in implementation specifics. Thus, a system for Big Data analytics should enable the expression of simple tasks, as well as combinations of tasks, in a manner that describes the application logic of the tasks and is adaptable to the user role, interest and expertise.

To fulfil the above requirements we propose a novel workflow model for the expression of analytics tasks on Big Data. The proposed model allows for the expression of the application logic while abstracting the execution details of tasks and the details on the data formats and sources. The model enables the separation of task dependencies from task functionality, as well as the adaptation of the level of description of execution semantics, i.e. the execution plan. In this way, the model can be easily, i.e. intuitively and in a straightforward manner, used by many types of users, with various levels of data management expertise and interest in the implementation. Therefore, using the proposed model, a user is not only able to express a variety of application logics for Big Data analytics, but also to set her degree of control on the execution of the workflow.

This means that model enables the user to express specific execution semantics for some parts of the workflow and leave the execution semantics of other parts abstract. The latter are decided by the analytics system at the processing time of the workflow.

1.1 Purpose of this Document

This document presents the proposed workflow model. This includes the declaration of the workflow and the accompanying execution semantics. The workflow is defined in terms of a graph. Vertices in the graph include single or groups of tasks, and edges in the graph represent dependencies between the groups of tasks included in vertices. We discuss the execution semantics of vertices and edges and we propose the creation of the analysed form of the workflow, which depicts the execution semantics of the original version. Furthermore, we make an initial discussion on methods that are necessary in order to manage the workflow in order to optimise its execution. Finally, we depict the proposed workflow model on specific use cases from the telecommunication domain.

1.2 Motivating Applications

Distributed data processing is the core application that motivates our research in the ASAP project. Storing and analyzing large volumes and a variety of data efficiently is the cornerstone use case. A goal of the project ASAP is to design and develop an analytics platform for the two analytics applications of consortium partners WIND and IMR, regarding telecommunication analytics and web analytics, respectively. In short, ASAP aims to build a unified, open-source execution framework for scalable data analytics. ASAP aims to develop the technology to facilitate the development and execution of general-purpose analytics queries over irregular data.

Cellular networks data sources bring a new quantity and quality to the analysis of mobility data. We are interested in the analysis of GSM/UMTS data in the context of existing and upcoming application scenarios. Possible use cases focus on handling, interpretation and analysis of cellular data. Wind as a TLC Operator is interested in Privacy-Aware Mobility Mining to improve its portfolio services according to the legal context: In the new ecosystems the right to the protection of the private sphere must coexist with the right to access to knowledge and to services as a common good. Through the analysis of CDR data (voice and SMS) and the correlation with Wind Customers (CRM data), it is possible to identify the real tourist flows, linking and counting the unique Wind customer making or receiving calls in states/provinces/cities other than that of residence. In this way the traffic would be automatically "sorted" and easily traceable and then analyzed. The proposed use case is important from the marketing point of view, because it can make possible, starting from the CRM data for each customer (and from her trajectories), to address a Social Network Analysis (SNA) taking into account the legal constraints coming from the privacy rules.

This analysis may be useful for various situations. Examples are the reconstruction and optimization of the transportation of things and people, the behavior analysis of people flows for the promotion of tourism, etc.

The Web Analytics application includes several data flow processes and query workflows. In short, developers implement stages of computation pipelines, which are then synthesized by workflow designers, and executed to answer user queries. Initially, the use cases are described and specified to target a small data set of almost 18k documents, averaging 200kb per document; in total 139GB of data. This is so that use cases are easy to deploy outside of the actual IMR data center, to facilitate research and development by the ASAP research partners. The plan is to eventually deploy and test the ASAP platform on the larger, actual data set in the IMR data center. Chapter 7 presents one basic indicative IMR use case.

1.3 Structure of this Document

The rest of this document is structured as follows. Chapter 2 presents the workflow definition, Chapter 3 presents the workflow execution, Chapter 4 discusses workflow manipulation, Chapter 5 discusses workflow optimisation, Chapter 6 summarises related work and Chapter 7 presents workflow examples for use cases. Chapter 8 concludes the document.

Chapter 2

Workflow Definition

The goal of the workflow is to enable the expression of the logical definition of user applications, which include *data processing*, i.e. *data accessing* and *computation*, as well as *dependencies* between instances of data processing. Computation may refer to algebraic computation or to more elaborate, algorithmic computation. The workflow models such applications as a graph. The vertices in the graph represent application logic and the edges represent the flow of data. Application logic includes (a) the analysis of data, and (b) the modification of data. Edges are directed and connect the vertices that produce and consume data. The rationale for adopting a graph model for the definition of a workflow is that the latter can enable the expression of application logic in an intuitive manner.

There are three types of vertices in a workflow, namely *root* vertices, *sink* vertices and *plain* vertices. The root vertices have only outgoing edges and they represent entry points of the application logic. Figure 2.1 explains the notation in all the figures that represent workflows and workflow parts. We require that each workflow has at least one root vertex. The sink vertices have only incoming edges and they represent final points of the application logic. We do not require that each workflow has one or more sink vertices. The vertices that are not of type root or sink are plain vertices, which means that they have both incoming and outgoing edges. For applications that include many phases of data modifications or analysis, we expect that most vertices in respective workflows are plain, as they represent points in the application logic where data are both produced and consumed. Workflows that do not have sink vertices are those that express an application logic of continuous execution. It is easy to see that workflows without sink vertices are graphs with cycles. Figure 2.2a shows a workflow with two root and two sink vertices. Figure 2.2b shows a workflow with no sink vertices, and, therefore, a cycle. Nevertheless, a workflow may comprise both acyclic sub-graphs and sub-graphs with cycles. A trivial case of such a workflow is one that expresses the

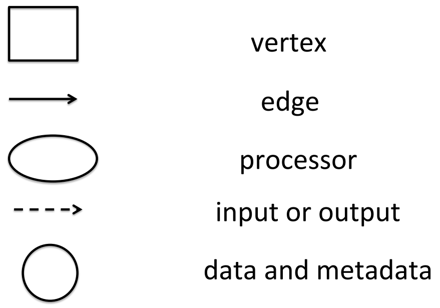


Figure 2.1: Notation of figures with workflows

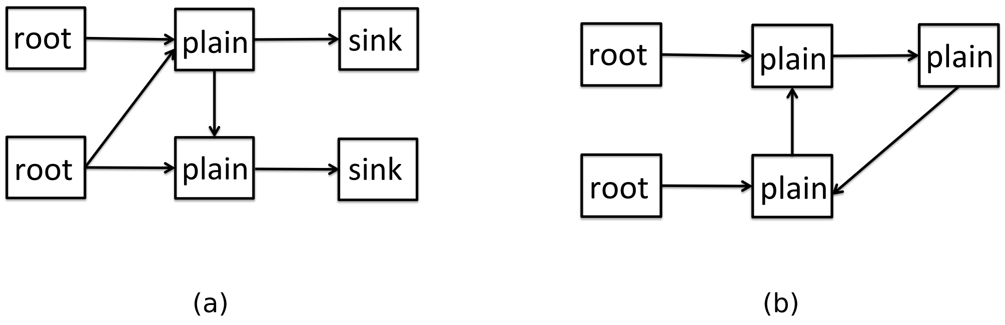


Figure 2.2: Workflow examples

logic of continuous querying that also outputs processed data, e.g. some final results to be archived.

The formal definition of a workflow is the following:

Definition 1 . *A workflow is a directed graph $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \mathcal{V}_r \cup \mathcal{V}_s \cup \mathcal{V}_p$ is a set that consists of three sets of vertices, the root $\mathcal{V}_r$, sink $\mathcal{V}_s$ and plain $\mathcal{V}_p$ vertices. The three sets do not overlap, i.e. $\mathcal{V}_r \cap \mathcal{V}_s \cap \mathcal{V}_p = \emptyset$, and there should be at least one root vertex, i.e. $\mathcal{V}_r \neq \emptyset$. Also, $\mathcal{E} = \{E_1, \dots, E_m\}$ is a set of edges. An edge $E \in \mathcal{E}$ is an ordered pair of vertices, i.e. $E = \{(V_i, V_j) | V_i, V_j \in \mathcal{V}\}$.*

Vertices and edges of workflows have properties. The properties of a vertex are related to tasks of the application part represented by this vertex, as well as corresponding metadata. The properties of an edge are related to data flow, and respective metadata, represented by this edge.

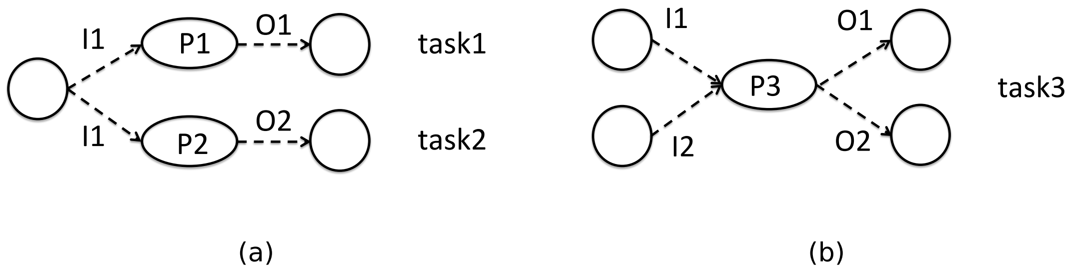


Figure 2.3: Task examples

2.1 Vertices

Each vertex in a workflow represents one or more tasks of data processing. Each task T is a set of *inputs*, *outputs* and a *processor*. An input inputs data to a processor; the latter represents the core of the data processing of the task, and, furthermore, an output outputs data generated from the processor. Therefore, inputs and outputs are related to descriptions of data and respective metadata. Figure 2.3 shows task examples. Figure 2.3a shows two tasks that have a shared input and one output each. Figure 2.3b shows a task with two inputs and two outputs.

Definition 2 *A vertex $V \in \mathcal{V}$ corresponds to non-empty set of tasks $\mathcal{T} \neq \emptyset$ such that each task $T \in \mathcal{T}$ is a set of inputs $\mathcal{I}$, outputs $\mathcal{O}$ and a processor P, i.e. $T = \{\mathcal{I}, \mathcal{O}, P\}$. Each input $I \in \mathcal{I}$ and output $O \in \mathcal{O}$ is a pair of data D and metadata descriptors M , i.e. $I = (D_I, M_I)$ and $O = (D_O, M_O)$.*

As defined, a vertex may represent one or multiple tasks of the application logic. These tasks may share or not inputs, but they do not share processors and outputs. The inputs and outputs of the tasks of a vertex can be related to incoming and outgoing edges of this vertex, but they do not identify with edges: inputs and outputs represent consumption and production of data, respectively, and edges represent flow of data. Similarly, vertices do not identify with processors. This semantic differentiation is necessary in order to allow the management of the dependencies in the workflow through graph manipulation separately from the management of data processing and computation in the workflow. Hence, it is easy to see that a vertex of any type, root, sink, or plain, may consist of tasks with non-empty sets of inputs and outputs, since the latter do not imply the existence of incoming or outgoing edges, respectively. Nevertheless, the incoming and outgoing edges of vertices are related in a 1-1 fashion with some inputs and outputs, respectively, of vertices. Therefore, if $\mathcal{E}_I$ and $\mathcal{E}_O$ are the sets of

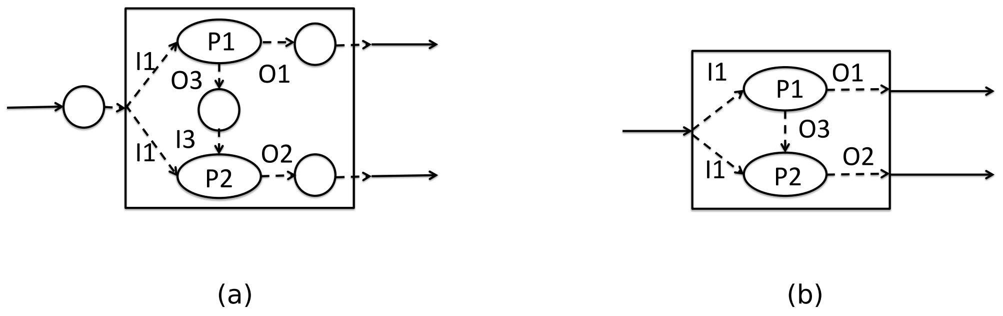


Figure 2.4: A vertex with multiple tasks

incoming and outgoing edges, respectively of a vertex V, and $\mathcal{T}$ is the corresponding set of tasks, then $|\mathcal{E}_I| \subset | \cup_{T.\mathcal{I}} |$, $\forall T \in \mathcal{T}$ and $|\mathcal{E}_O| \subset | \cup_{T.\mathcal{O}} |$, $\forall T \in \mathcal{T}$.

Figure 2.4 shows an example of a vertex with two tasks. (Figure 2.4a is the detailed representation and Figure 2.4b is a simplified representation where the cycles that represent data are omitted). The tasks share input $I1$ and each has one output, $O1$ and $O2$ that each are an input to an edge outgoing from this vertex. The input $I1$ is the output of an edge incoming to this vertex. Also, one of the tasks has one more output, $O3$, which is an additional input, $I3$, of the other task. The input/output $O3/I3$ is not related with any edge, meaning that these data are not input to tasks that correspond to any dependent vertex. Figure 2.5 shows a vertex with one task, which creates a histogram of the input data. The task outputs the histogram itself and a set of additional statistics. These two outputs are separated and are input to two different edges in order to feed two different tasks. The histogram is further processed and the additional statistics are logged.

A vertex needs to correspond to at least one task, but it can also correspond to more than one task of the application. Such tasks may or may not adhere to any sort of relation, e.g. concerning associations or similarities of their inputs, outputs or processors. Nevertheless, the reason why the proposed model allows the definition of vertices with multiple tasks, is to enable the user to express such associations or similarities. Therefore, the definition of a vertex in a way that it consists of multiple tasks, enables the definition of workflows that are intuitive with respect to the rationale of the application logic.

Figure 2.6 shows a vertex that represents a SQL query. The vertex includes separate tasks for different parts of the SQL query. All the tasks share the input data, and one of them, the task that represents a *join*, has an additional input. Each task has an output. Note that the output data of this vertex, which is the input to the outgoing edge, is actually the output data after executing the whole group of tasks represented by this

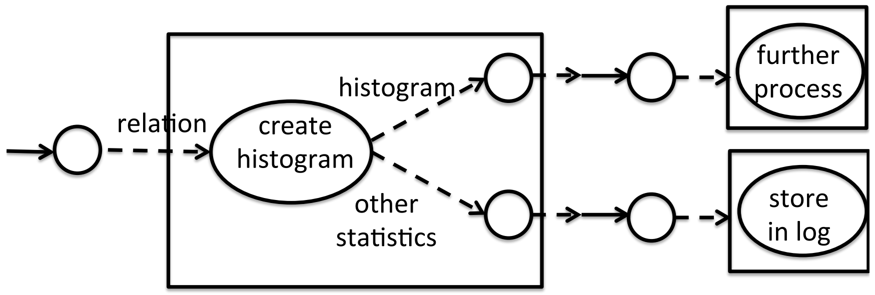


Figure 2.5: A vertex with a task with two outputs

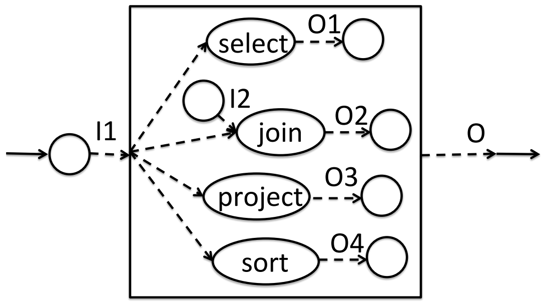


Figure 2.6: A vertex that represents a SQL query

vertex. Notably, the output of the vertex can be any of the $O1$, $O2$, $O3$, $O4$, depending on the execution plan of this group of tasks. Allowing the user to define vertices with multiple tasks, enables her to represent in a vertex part of the application logic that she considers to be, conceptually, one unified complex task, without requiring her to define at the same time the way that this complex task should be executed, i.e. the execution semantics of it.

The processors in tasks realise the application logic, which is, as mentioned earlier, the analysis or the modification of data. Section 2.4 gives details on the concept of the processor and discusses proposed instantiations.

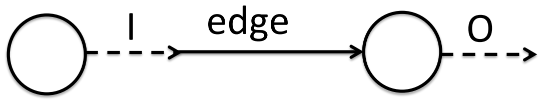


Figure 2.7: Edge with the respective input and output

2.2 Edges

Each edge in a workflow corresponds to a pair of an input I and an output O of the same data D. As mentioned, the I and the O of an edge correspond in a 1-1 fashion to an I and an O, respectively of a task. The data D are accompanied by metadata M, which can be different for the input and the output of the same edge. Figure 2.7 depicts the input and the output of an edge. Figure 2.8 shows an edge that connect two vertices, with one task each. The output of one task becomes the input of the other, via the dependency created by the edge connecting the two vertices. (Figure 2.8a shows the detailed representation of this example, and Figure 2.8b shows the simplified representation of this example, where the cycles representing the data are omitted.)

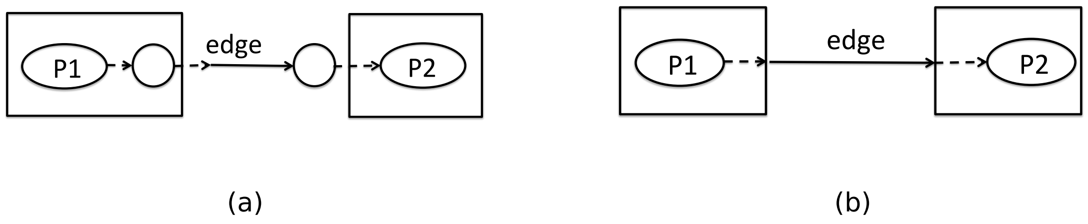


Figure 2.8: An edge connecting two vertices with one task each

Formally:

Definition 3 *An edge $E = (V_i, V_j)$, $V_i, V_j \in \mathcal{V}$, in the workflow corresponds to a pair of an input and an output (I, O). Input I is a pair of data D and some metadata M_I, i.e. $I = (D, M_I)$, and output O is a pair of data D and some metadata M_O, i.e. $O = (D, M_O)$. Input I is equivalent with an output O' of a task that corresponds to vertex V_i, i.e. $\exists T \in V_i.\mathcal{T}, \exists O' \in V_i.T.\mathcal{O}$ such that $I \equiv O'$. Also, output O is equivalent with an input I' of a task that corresponds to vertex V_j, i.e. $\exists T \in V_j.\mathcal{T}, \exists I' \in V_j.T.\mathcal{I}$ such that $O \equiv I'$.*

Hence, an edge defines the flow of data from one vertex to another according to some metadata that describe production and consumption information for these data. The production and consumption information can be the same or different and are related to (a) the data flow (b) the data persistence (c) the data reliability. Other types of metadata may be added in future work. In general,such metadata can be any information that plays a role in determining the execution plan of the workflow. More specifically, the metadata types are:

- **Flow.** Metadata concerning the data flow pattern. The values of this type may be the following:
  - Batch: The data flow in batches. Concerning input data, this means that they are consumed in batches (e.g. all data need to be available and accessed for processing to begin), and concerning output data, this means that they are available in batches, (e.g. produced data are stored in memory but they cannot be accessed until they are dumped in permanent storage).
  - Stream: The data flow as a stream. Concerning input data, this means that they are consumed in a streaming manner (i.e. data are processed as soon as they are available as an input), and concerning output data, this means that they are available in a streaming manner, i.e. as soon as they are produced (e.g. either from memory or from permanent storage).
- **Persistence.** Metadata concerning the data persistence in storage. The values of this type may be the following:
  - Persistent: The data are stored in permanent storage. Therefore, concerning output data, they remain available after a task completes, and concerning input data, they remain available during and after they are used as input.
  - Volatile: The data are not stored in permanent storage or they are stored only for a limited time. For example, concerning output data, they may not remain available after a task completes, and concerning input data, they may not remain available after they are used as input.
- **Reliability.** Metadata concerning the correctness or completeness of data. The values may be the following:
  - Reliable: The data are considered to be correct and complete[1].
  - Non-reliable: The data are considered to be either incorrect and/or incomplete.

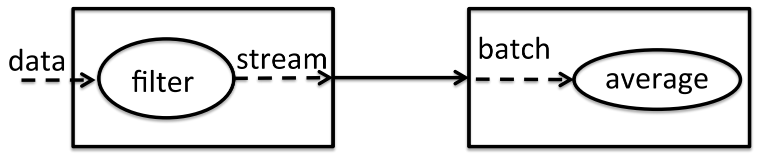


Figure 2.9: Example of an edge with different flow metadata of input and output data

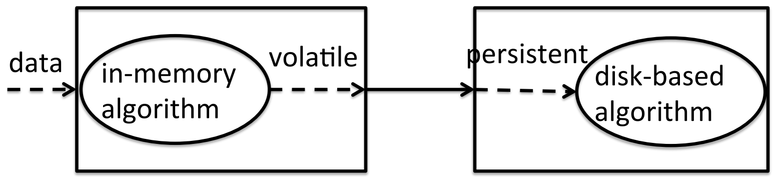


Figure 2.10: Example of an edge with different persistence metadata of input and output data

The above values of the metadata types may, even further, have properties, especially related to quantification. For example, the flow values may be characterised by size (e.g. the size of the batches or streams or the range of their varying sizes) or the rate of their availability in time; the persistence values may be characterised by the lifetime of data or required guarantees for availability; and the reliability values may be characterised by probability estimations.

As mentioned, the metadata of the input and the output pair of an edge can be different. Thus, the values for the same metadata type for the input and the output of an edge can be different. For example, concerning the data flow, the input value can be 'batch', while the output value is 'stream'. Figure 2.9 shows an edge that connects a vertex with a task that filters some data, with a vertex with a task that computes the average of the filtered data. The filtering task outputs data in a streaming manner, but the task that computes the average takes as a input a batch of data. Therefore, even though the input and output data‘ of the connecting edge is the same, the respective metadata concerning the flow of data differ.

Concerning the data persistence, for example, the output value can be persistent

[1]Correctness and completeness of data may have application-specific semantics.

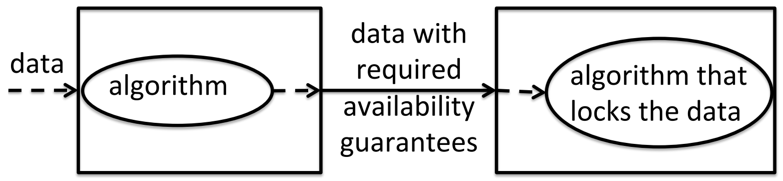

data
algorithm
data with required availability guarantees
algorithm that locks the data

Figure 2.11: Example of an edge with different availability metadata of input and output data

with a lifetime of $1sec$, while the input value can be 'volatile', meaning no lifetime at all after consumption. Figure 2.10 shows an edge that connects a vertex with an in-memory algorithmic task, with a vertex with a disk-based algorithmic task. The in-memory algorithm outputs data that are stored in the main memory and not in the disk, thus they are volatile; whereas the disk-based algorithm reads these data from the disk. Therefore, the persistence metadata of the input and output of the connecting edge are different.

Figure 2.11 shows another case of different metadata with respect to data persistence. The input and output data of the edge connecting two vertices are both persistent, i.e. stored in the disk, but their availability requirements differ. The algorithm in the left vertex outputs one copy of some data; the algorithm in the right, i.e. the dependent, vertex, locks this copy in order to process the data; yet, these data should be always available for reading by other tasks.

Furthermore, concerning data reliability, for example, the input and the output data may have different guarantees. A task may output data with some reliability guarantee, e.g. data can be considered correct with a 0.5 probability, whereas the task that takes this data as a input may require that they can be considered correct with a 0.8 probability.

It is interesting to note that combinations of the metadata values for the input and output data of an edge create different execution plans. This issue is discussed further in Chapter 3.

2.3 Data

The data D of inputs and outputs of edges, as well as of inputs and outputs of tasks consists of information on the data source where these data reside, as well as information on the data *unit*. Formally:

Definition 4 *The data D of an input $I = (D, M_I)$ or an output $O = (D, M_O)$ is a set $D = \{S, u, \mathcal{A}\}$, where S is the data source, u is the basic data unit and $\mathcal{A}$ includes additional information. The data source is a pair $S = (n, t)$ of the name n and the type t of the source. The unit u takes values from a constraint domain $\mathcal{D}$, which includes the names of the basic units for known types of data sources.*

The type t of a data source can be one of the well known ones, e.g. ‘relational’, ‘rdf’, ‘xml’, ‘key-value’ etc. The unit for each type is unique and pre-specified; e.g. the unit of the relational type is the ’tuple’, the unit for the ’rdf’ type is the ’triple’ and the unit for the ’key-value’ type is the ’pair’. Moreover, data may include the description of additional information, such as relation and attributes names, as well as schema information (e.g. primary and foreign key constraints) and information on the respective processing engine, (e.g. engines of NoSQL databases, relational DBMSs like MySQL, etc).

2.4 Processors

The tasks included in vertices take as input data and metadata, process the data using a processor and output some data and metadata. Each processor can have an abstract definition and several implementations, i.e. one or more implementations per platform. For example a processor that implements a ‘join’ for two data inputs, has an abstract definition, and can be implemented for a relational DBMS and a NoSQL database. In order for a processor to be used on a specific platform, it is required that this processor is implemented for the specific platform. The same holds for processors that perform more complex operations, such as algorithmic computation. A processor definition includes restrictions on the type and number of inputs and specifies the number and type of outputs. Defined and implemented processors form a library from which a user can select processors to describe tasks. Users can define their own processors and should provide respective implementations, in which input and output data can be in the form of raw bytes/records/key-value pairs etc.

In the following we give examples of the definition of basic processors, namely the *select*, *calc* and *join*:

$$O(select, I) = \{r \mid r \in I \ \wedge \ SelectPredicate(r)\}$$
$$O(calc, I) = \{r \cup \{attr : value\} \mid r \in I \ \wedge \ value := CalcExpression(r)\}$$
$$O(join, I_1, I_2) = I_1 \bowtie I_2 = \{t \cup s \mid t \in I_1 \ \wedge \ s \in I_2 \ \wedge \ JoinPredicate(t \cup s)\}$$

The input and output data of a processor are accompanied by metadata that describe their type, format and other characteristics. The metadata defined for each processor have a generic tree format (JSON, XML etc). In order to allow for extensibility, the first levels of the meta-data tree are predefined; yet, users can add their ad-hoc subtrees to define their customized processors .

The generic metadata tree for data definitions is the following:

```
{<input | output>: {
        "constraints": {
                "data_info": {
                        "attributes": {
                                <attr1..n>: {
                                        "type": <type>}},
                        "engine": {
                                "DB": <db_meta>}}
        }}
```

The generic metadata tree for processors is the following:

```
{<operator_name>: {
        "constraints": {
                <input1..n>
                <output1..m>
                "op_specification": {
                        "algorithm": <alg_meta>}}
                }}
```

The property <alg_meta> for the processors defined above is the following:

For *select*:

```
{"select": {
        "select_condition": <select_predicate>
        }}
```

All rows for which the <select_predicate> is ‘True’ are returned:

<select_predicate> ::= {<field_name> <comparison_operator> <value>}

<comparison_operator> ::= [’>’ | ’<’ | ’>=’ | ’<=’ | ’==’ | ’!=’]

For *calc*:

```
{"calc": [{
        "calc_attr": <calc_attr>,
        "calc_value": <calc_expression>
        }]}
```

The *calc* processor produces data with new attribute <calc_attr> and value calculated by <calc_expression>:

<calc_expression> ::= <attr_name> <action_operator> [<attr_name> | <value>]

<action_operator> ::= [['+' | '-' | '*' | '/'] | ['concat' | 'substring'] | ['u' | 'n' | '\']] <value>.
The <action_operator> depends of the attribute type.

For *join*:

```
{"join": {
        "join_condition": <attr_name | attrs equality>
        }}
```

The *join* processor has a minimum of two input data.

The following is an example of a customised *join* processor:

```
{"joinOp": {
        "constraints": {
                "input1": {
                        "data_info": {
                                "attributes": ["$attr1", "$attr2"]}},
                "input2": {
                        "data_info": {
                                "attributes": ["$attr1", "$attr2"]}},
                "output1": {
                        "data_info": {
                                "attributes": ["$attr1", "$attr2"]}},
                "op_specification": {
                        "algorithm": {
                                "join": {
                                        "join_condition":
                                          "input1.$attr1=input2.$attr1"}
                                             }}}
        }}
```

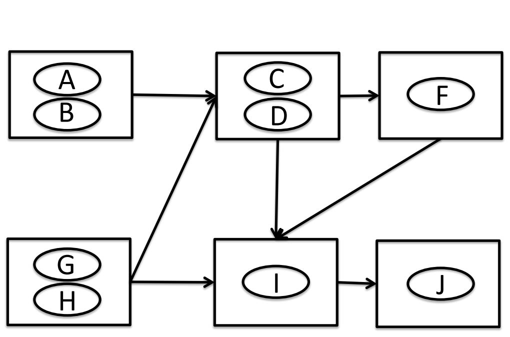
Chapter 3

Workflow Execution

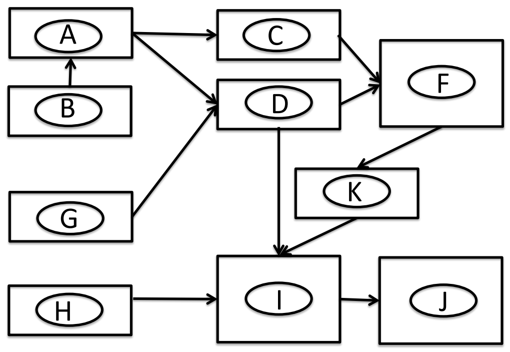
A workflow represents the dependencies among processing tasks that analyse or modify data, as well as the input and output data of these tasks, together with respective metadata. The defined workflow structure allows for the user to depict the application logic in mind in a straightforward and intuitive manner. This is achieved with two design choices for the workflow structure: (a) the semantic abstraction and separation of the description of processing tasks from the dependencies of processing tasks, and (b) the association of vertices with one or multiple tasks.

The first choice enables the user to describe the application logic and the processing units in the application independently, allowing for easy changes and updates of the workflow structure, as well as a modular and gradual definition of a workflow. This choice also allows the user to be agnostic with respect to the execution semantics of the dependencies between tasks. This execution semantics is determined based on the combination of input and output data and metadata of edges, and will be discussed in the following.

The second choice enables the user to depict in the workflow structure the semantic dependencies of processing units with the depiction of edges and vertices, allowing her to be agnostic on the execution semantics of the set of processors that correspond to a single vertex. Therefore, a user can define a vertex with multiple processors, which, as a group, define a complex operation on the data. Such an operation may be a traditional way of data querying, for example, a vertex may correspond to a SQL query of the Select-Project-Join form; or the operation may be a processing module that comprises simple and complex computation units, like algorithms, for example a data mining algorithm and some sorting of the output data.



(a)



(b)

Figure 3.1: The original and analysed version of a workflow

3.1 The Analysed Workflow

The workflow structure alleviates from the user the burden of determining any execution semantics for the application logic. The execution semantics of the workflow includes the execution of tasks of vertices and the execution of input-output dependencies of edges. The determination of the execution semantics of vertices and edges leads to an execution plan of the workflow. We refer to this plan as the *analysed* workflow. The latter is actually an enhancement of the initial workflow with more vertices, and substitution of vertices and/or edges in the initial workflow with others.

More specifically, in the analysed workflow, an edge with different input and output metadata, may be replaced with two edges and a new vertex; the new vertex corresponds to a new task that takes the data and metadata of the input of the initial edge and produces the data and metadata of the output of the initial edge. In other words, since the data of the input and the output of an edge are equivalent, this task changes only the metadata. Such vertices are *associative*, as they encompass associative tasks. Also, a vertex that includes multiple tasks, in the original workflow, is replaced, in the analysed workflow, with a set of new vertices that each includes one task of the original vertex. The new vertices may or may not be connected with new edges.

Figure 3.1 shows an example with the original and the analysed version of a workflow. The original workflow in Figure 3.1a has 9 tasks, 3 vertices with two tasks each and 3 vertices with 1 task each. The analysed workflow has more vertices: each one of the 3 vertices with two tasks are replaced with two vertices with 1 task each; also, the edge connecting vertices with tasks F and I is replaced with two more edges and an associative vertex, which includes the associative task K. The analysed version shows that the execution of the vertex with tasks A, B is planned as: first, execution of task B; second, the output of B is input to task A; and third, execution of A. The analysed version also shows that the dependency of tasks C, D on tasks A, B in the original version, means that the output of task A is input to both tasks C and D, which are executed in parallel. Furthermore, the dependency of task F from tasks C, D in the original version, is executed with the input of the outputs of both C and D to task F.

Definition 5 *An associative vertex V_a corresponds to an associative task $T_a = \{I_a, O_a, P_a\}$, where $I_a = (D, M_I)$ and $O_a = (D, M_O)$.*

An associative vertex together with a pair of new edges replaces an edge in the initial workflow. Such new edges are also called associative. We call this triple the *associative triple* of an edge.

Definition 6 *An associative triple $A = (V_a, E_a, E'_a)$ of an edge $E = (V_i, V_j)$ is a set that consists of an associative vertex V_a and a pair of associative edges E_a, E'_a, where $E_a = (V_i, V_a)$ and $E'_a = (V_a, V_j)$. If E, E_a and E'_a correspond to (I, O), (I_a, O_a) and (I'_a, O'_a), respectively, then it holds that $I \equiv I_a \wedge O_a \equiv I'_a \wedge O'_a \equiv O$.*

Essentially, the eventual replacement of edges with associative triples realises the execution semantics of the replaced edge, by creating, through the processor of the associative vertex, an explicit execution plan of the dependency represented by the replaced edge. In Section 3.2 we discuss the types of processors for associative vertices.

Furthermore, in the analysed workflow, a vertex that corresponds to multiple tasks is replaced with an *associative subgraph* that contains a set of new vertices that correspond to these tasks. This set contains vertices that correspond to the tasks of the initial vertex: each new vertex corresponds to one task; vertices may correspond 1-1 to tasks, but it can be the case that two or more vertices correspond to the same task[1]. Naturally, the incoming edges of the initial vertex may have to be replicated, since they may correspond to the input of more than one tasks. The outgoing edges, however, remain the same, as each corresponds to the output of one task. The replacing subgraph may also contain new edges that connect the replacing vertices. Such edges represent the dependencies between tasks related to their execution semantics, and not related to the semantics of the application logic, as expressed by the user.

Definition 7 *An associative subgraph $G_a(\mathcal{V}_a, \mathcal{E}_a)$ of a vertex V consists of a set of new vertices $\mathcal{V}_a$ and a set of new edges $\mathcal{E}_a$. If V corresponds to a set of tasks $\mathcal{T}$, then it holds that $\forall V \in \mathcal{V}_a, \exists T \in \mathcal{T}$ such that V corresponds to T, and $\cup_{V:T}, \equiv \mathcal{T}$.*

[1]Replication of tasks using many associative vertices that correspond to the same task of an original vertex may be necessary for the optimisation of the workflow execution.

Hence, the analysed workflow is the initial workflow where some edges and vertices are replaced by associative triples and subgraphs, respectively.

Definition 8 *An analysed workflow is a directed graph $G^A(\mathcal{V}^A, \mathcal{E}^A)$, where $\mathcal{V}^A = \mathcal{V} \cup \mathcal{V}_{new}$ - $\mathcal{V}_{rep}$ and $\mathcal{E}^A = \mathcal{E} \cup \mathcal{E}_{new}$ - $\mathcal{E}_{rep}$. The set $\mathcal{V}_{rep} \subseteq \mathcal{V}$ includes the replaced vertices and the set $\mathcal{E}_{rep} \subseteq \mathcal{E}$ includes the replaced edges in a workflow $G(\mathcal{V}, \mathcal{E})$. It holds that $\forall V \in \mathcal{V}_{rep}$ $\exists\ G_a(\mathcal{V}_a, \mathcal{E}_a)$ and $\forall E \in \mathcal{E}_{rep}$ $\exists\ A = (V_a, E_a, E'_a)$. Also, it holds that $\cup_{\mathcal{V}_a, V_a} \equiv \mathcal{V}_{new}$, where $\mathcal{V}_a \in \cup_{\mathcal{G}_a}$ and $V_a \in \cup_A$; and $\cup_{\mathcal{E}_a, E_a, E'_a} \equiv \mathcal{E}_{new}$, where $E_a, E'_a \in \cup_A$ and $\mathcal{E}_a \in \cup_{\mathcal{G}_a}$.*

The analysed workflow represents the execution semantics of the application logic represented by the initial workflow.

3.2 Execution Semantics of Edges

The input and output of an edge describe data and information concerning the production and consumption, respectively, of these data. This information, as discussed in Section 2.2 can be related to properties concerning the flow, the persistence and the reliability of data. This set of properties may be extended in the future.

The qualitative or quantitative difference in the values of the same property for the input and output of an edge implies some sort of compatibility or incompatibility between the tasks in the connected vertices, and creates constraints for the execution of the dependency represented by the edge. Therefore, the property values of the input and output metadata require appropriate execution semantics of the edge. An appropriate associative triple that replaces the edge in the analysed form of the initial workflow. The associative triple contains an associative vertex that corresponds to a new task, which realises the execution semantics required by the input and output metadata of the initial edge. This is an associative task and includes an associative processor. An associative processor is dedicated to a specific associative task. These tasks belong to the following categories:

- **Scheduling Tasks.** This category includes tasks that realise different data flow patterns, by scheduling the propagation of data from the production to the consumption vertex. Such tasks may realise various degrees of:
  - Sequential data propagation: The data are propagated from input to output in a sequential manner. For example, the trivial case of an edge that has $I = O = (D, \{`stream', rate = 1msec\})$, necessitates a task that realises the execution semantics of a pipeline.

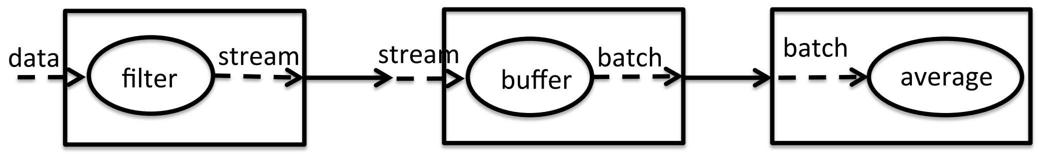


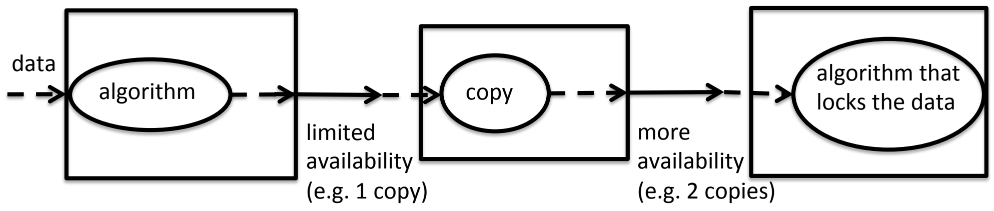
Figure 3.2: Example of a scheduling task

- Concurrent data propagation: The data are propagated from input to output in a concurrent manner. For example, the trivial case of an edge that has $I = O = (D, \{`batch', size = 10MB\})$, necessitates a task that realises the execution semantics of a buffer of size that equals 10MB.

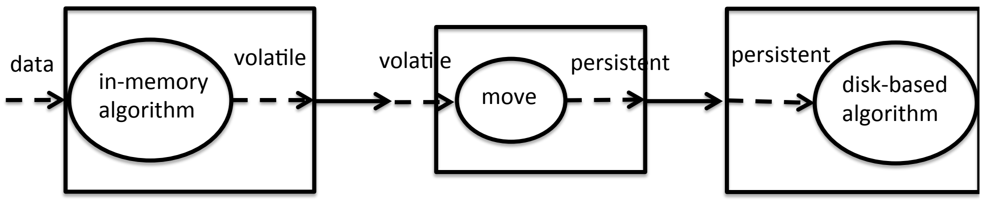
Figure 3.2 shows the associative triple that substitutes an edge that connects two vertices with a filtering task and a task that computes an average value, as in Figure 2.9. The triple includes a new edge that connects the filtering task with the associative task, and a new edge that connect the associative task with the *average* task. The associative task performs buffering of the streaming input data, so that they can be input as a batch in the *average* task. Therefore, the data, but also the flow metadata of the input and the output for both the new edges are the same: the left edge has as input and output streaming data, whereas the right edge has as input and output batch data.

- **Availability Tasks.** This category includes tasks that realise different patterns of data persistence, by replicating and moving data in order to change their availability as formed by the production vertex and guarantee their availability as requested by the consumption vertex. Figure 3.3 shows examples of triples with associative tasks related to availability. Such tasks may realise various degrees of:

  - Data replication: The produced data are replicated in order to be available for consumption. The replication may be performed for several reasons. In case of produced data that are volatile, i.e. they are stored in memory, they are replicated in permanent storage; in case of produced data that have some degree of persistence in permanent storage, they are replicated in the same storage in order to increase their availability in terms of time; in case of produced data that are totally persistent, they are replicated in the same or other


data
algorithm
limited availability (e.g. 1 copy)
copy
more availability (e.g. 2 copies)
algorithm that locks the data

(a)



(b)

Figure 3.3: Example of availability tasks

storage in order to increase availability in terms of processing throughput.

Figure 3.3a shows the associative triple that replaces the edge connecting two algorithmic tasks, as in Figure 2.11. The associative task makes a copy of the data output by the first algorithm, so that one copy is available for other tasks to read, while one copy is locked by the dependent algorithmic task.

– Data movement: The produced data are moved in order to be available for consumption. The movement may be performed for several reasons. Persistent data may be moved to other storage in order to be closer to the processing in the consumption vertex and, therefore, increase availability in terms of processing throughput. Also, persistent data may be moved in order to reduce the risk of unavailability due to source or engine failures.

Figure 3.3b shows the associative triple that replaces the edge connecting an in-memory and a disk-based algorithmic task, as in Figure 2.10. The associative task moves the data from the main memory to the disk, so that the

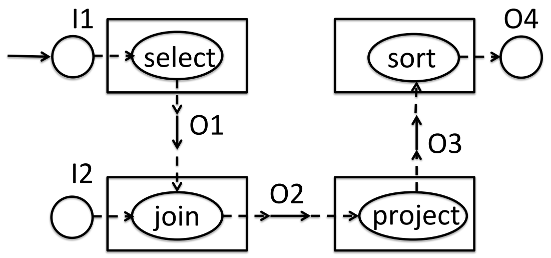


Figure 3.4: Example of an analysed workflow for a vertex that represents a SQL query

dependent algorithm can access them.

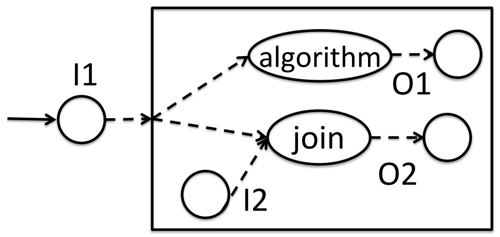
- **Cleaning Tasks.** This category includes tasks that realise different patterns of data reliability, by controlling and mending the quality of the data as formed by the production vertex and guarantee their reliability as requested by the consumption vertex[2]. Such tasks may realise various degrees of:
  - Data checking: The produced data are checked for the correctness and completeness in order to decide if the reliability requested by the processing in the consumption vertex can be guaranteed.
  - Data mending: The produced data are processed in order to be corrected and/or completed, so that the reliability requested by the processing in the consumption vertex can be guaranteed.

Each task category corresponds to a library of processors that implement specific instances of this category. Such a library contains actually code for processors, and it can be extended with new processors.

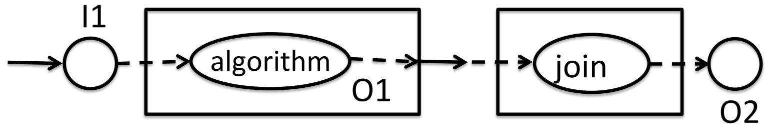
3.3 Execution Semantics of Vertices

A vertex in the initial workflow corresponds to a set of tasks that are defined, by the user, to be performed as a unit in the application logic. The degree of detail, in terms of processing, of such a set, depends on the nature of the application, the user role, the user experience and knowledge in creating workflows etc. This means that different users may describe the same part of an application logic in different granularity, and, consequently, by defining a different set of vertices with a small or big number of tasks

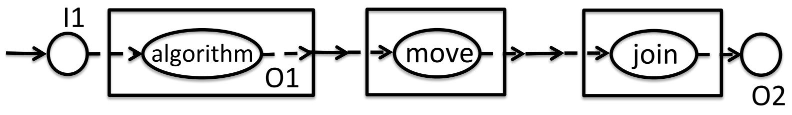
[2]Note that such tasks may involve also human interaction and may be performed online or offline.



(a)



(b)



(c)

Figure 3.5: Example of an analysed workflow for a vertex that represents complex computation

each. For example, such an application logic may refer to a SQL query, with a *Select-Project-Join-Sort* form. If the user is a business analyst, then she may define one single vertex that corresponds to all four tasks, i.e. *Select*, *Project*, *Join*, *Sort*, as in Figure 2.6, or, if the user is a programmer, she may define one vertex per task, as in Figure 3.4. In the last case, the user has to also define a set of edges that connect the four vertices. This set of edges, (which may be empty, in case of parallel execution), in essence, indicates how the user perceives the execution semantics of the tasks. Therefore, the user may impose the execution semantics for a set of tasks, or she may be agnostic to this. In the last case, the execution semantics of these tasks should be defined in the analysed workflow. This is achieved by replacing the initial vertices with associative subgraphs, that define the respective execution semantics. In the example of the SQL query above, the vertex corresponding to the four tasks is replaced by a subgraph that contains one vertex per task. This subgraph is actually a query execution plan, e.g. a plan that is created by a DBMS optimiser. The structure of this plan depends on the set of tasks, as well as on the type of the execution platform and engine.

Figure 3.5 shows another example of an application logic that can be described differently by different users, who aim to define different levels of detail with respect to the execution planning of this logic. Figure 3.5a shows a vertex that includes two tasks, an algorithmic processing on some input data, and a join of these data with some other data. The user that defines this vertex is not interested, or does not know the execution details of this complex task that includes two separate simpler tasks. Using this representation, she aims to depict that these two tasks should be executed together, after the tasks of the vertices on which this vertex depends, and before the tasks of vertices that depend on this vertex. Figure 3.5b shows that another user, represents the same two tasks with two connected vertices, with which she dictates that the join should be executed on the data processed first by the algorithm. Furthermore, Figure 3.5c shows that another user, dictates even more detail in the execution plan of these two tasks, by adding one more vertex that includes a task that moves the data, e.g. from one disk to another.

The associative subgraph that replaces a vertex in the initial workflow may not lead to the optimal execution of the workflow. As discussed in Chapter 5, such a subgraph may be changed in the optimised workflow, in order to optimise efficiency, or even some other performance quality.

3.4 Execution with Constraints

The creation of the analysed workflow as described in the previous sections assumes that there are no execution constraints, other than the dependencies of tasks in the application login. However, the application logic may be accompanied by other constraints, attached to tasks, such as deadlines and milestones. Such constraints need to be taken into account in the definition of the overall execution semantics. Therefore, in an extension of the current work, we will extend the definition of the original and the analysed workflow accordingly.

Chapter 4

Workflow Manipulation

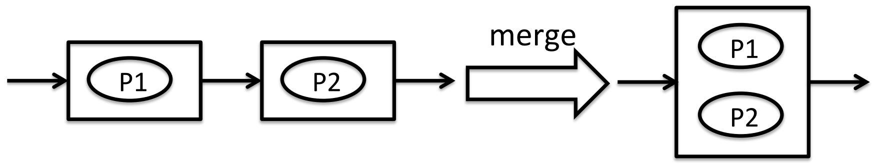
The application logic expressed by a workflow should be implemented and executed in a manner that is efficient or, in general, optimal in terms of some performance qualities, such as data availability or reliability. Optimality can be achieved by changing the execution of the original workflow. In order to change the workflow execution we need to change the structure of the analysed form of the workflow. In this chapter we discuss an initial proposal of methods for manipulating the workflow in order to change its structure and execution. The goal is to propose methods that do not change the application logic while changing the structure of the workflow. Such methods can use one of the following:

- **Operations.** The restructuring of the workflow can be performed using operations defined to change the vertices or the tasks included in vertices.

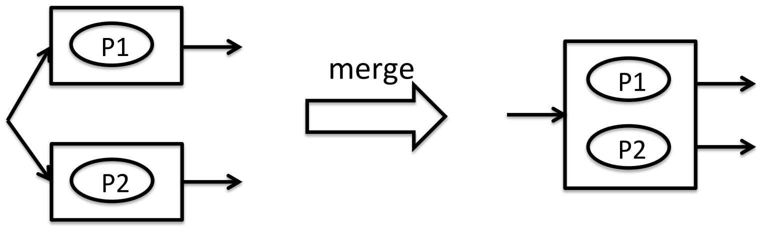
- **Properties.** The restructuring of the workflow can be performed employing properties of processors in the tasks included in vertices.

4.1 Operations on Vertices

Since the workflow is a graph, we can restructure it by changing either the paths between vertices or by adding and/or removing vertices. Changing the paths between vertices implies the change of inputs and output edges (and, therefore, data and metadata) of *existing* vertices, (and, therefore, existing groups of tasks, in general), without any guarantee that the conceptual dependencies among tasks, and furthermore, final outputs, as these are defined in the application logic, are preserved. Therefore, we do not proceed with the proposal of methods that change the graph edges.



(a)



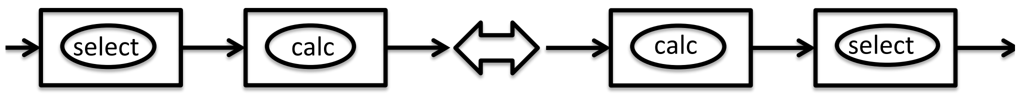
(b)

Figure 4.1: Examples of operation ‘merge’

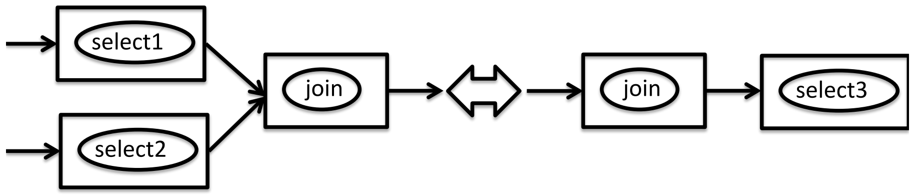
Oppositely, changing the vertices, implies substituting the existing vertices with new ones, for which, it may be possible to guarantee that the conceptual dependencies among tasks, and, furthermore, final outputs of the workflow, as defined in the application logic, are preserved. Therefore, we are designing methods for changing the workflow structure by applying operations on vertices. As an initial proposal of such operations, we describe the operations *merge* and *split*.

Merge. The *merge* operation takes as an input two vertices and produces one new vertex that includes the tasks of both initial vertices. The vertices that are merged can be connected with an edge, i.e. together they represent some task dependency(ies), or not, i.e. there is no task dependency between them. The goal of the *merge* operation is to allow for a united optimisation of the tasks included in the two initial vertices. Figure 4.1a shows a sequential arrangement of the two merged vertices, and Figure 4.1b shows a parallel arrangement of the two merged vertices.

Split. The *split* operation takes as input one initial vertex and produces two new vertices that, together, include all the tasks included in the initial vertex. The two new vertices may or may not be connected. The goal of the *split* operation is to lead to separate optimisation of subgroups of tasks included in the initial vertex.



(a)



(b)

Figure 4.2: Examples of commutativity

4.2 Operations on Tasks

Beyond operations on vertices, we intend to propose operations on tasks included in vertices. One such operation can be the *union* of tasks. The union operation can take as an input two or more tasks and produce one new that is able to produce the outputs of all these tasks. As an example, the union of two sequential instances of *select* is always possible and produces one new *select* instance, where *selectPredicate* is a combination of predicates of the initial *select* instances.

4.3 Properties of Processors

In order to change the workflow structure, beyond operations applied on vertices and tasks, we need to employ properties of processors, such as: distributivity, associativity and commutativity. Such properties need to hold, for operations on tasks and vertices to be performed with a correct output, i.e. the output as defined by the user in the original workflow. We intend to prove such properties for defined processors. Naturally, each defined processor is a special case in terms of how it processes the input data, and, thus, we need to prove the properties for processors individually or in groups.

Figure 4.2 shows the rearrangement of tasks for pairs of basic processors defined in Section 2.4. The commutativity property is not always satisfied in such pairs. For the pair *calc - select* the rearrangement in Figure 4.2a cannot be made if the *select* processor takes as input data produced by *calc*. The opposite rearrangement can be performed always. For the pair *select - join* the restriction is equality of *selectPredicate* of the two *select* processors. The rearrangement of the pair *join - select* in Figure 4.2b can be made always. For the pair *calc - join* the restriction is equality of *CalcExpression* and *attri* in the *calc* processors.

Chapter 5

Workflow Optimization

As described in the previous chapter, currently we are developing methods for handling the workflow. These methods will be employed for the creation of the *optimised* version of the workflow, in terms of performance (usually performance is interpreted as efficiency, but we may explore other performance qualities, such as fault-tolerance).

The optimisation of the workflow will be first explored on subgraphs of the analysed workflow. Such subgraphs may contain several paths and can be considered as complex tasks. Identification and optimisation of complex tasks has two merits on the long run: first, the optimisation of the workflow is modular, therefore the search space for solutions is smaller and easier to explore; second, we will be able to create a library of optimised complex tasks, which we can use as templates for the optimisation of the same (or similar tasks) in other workflows.

A challenge in the optimisation of a workflow is that the latter contains tasks that span multiple data stores and query engines. As an example, a query may include processors on two types of data, e.g. relational and key-value. In such cases, it is necessary to perform two types of optimisation: *active* optimisation and *lazy* optimisation. We call 'active' the optimisation that is performed on individual data stores and engines, and we call 'lazy' the optimisation that is performed between data stores and engines. Active optimisation is actually passed on to the respective platforms, whereas lazy optimisation has to be performed in a intra-platform manner. To realise lazy optimisation we will need to develop special tasks, with which we will *augment* the analysed workflow. These tasks may have to implement translation or modification of data, data movement and replication. For example, consider that the processor in the example mentioned above is a *join* between a relational and key-value store; in this case, data from the key-value store may need to be transformed, trivially, in relational tuples and stored in the relational store. In general, the goal of such tasks will be the splitting and/or merging of initial tasks on multiple sites and platforms.

Furthermore, we will pursue optimisation of the analysed workflow in presence of execution constraints, such as deadlines and milestones. In case of constraints, first we should determine incompatibilities among accompanying constraints. Then, the modular workflow optimisation should be performed by prioritising subgraphs with constraints.

The optimisation of a workflow will be pursed on two axes, namely: optimisation via graph reconfiguration and via optimal resource management.

5.1 Optimization via Graph Reconfiguration

The first axis of optimisation will be the reconfiguration of the graph of the analysed workflow. To do this we will employ results from our research on the development of methods for the manipulation of the workflow. We will devise a methodology to apply the developed methods, on the lines of traditional query plan optimisation.

Although, static optimisation of the workflow may be possible in presence of statistical estimations for the performance of implemented processors on different platforms and for various data sources, we aim to develop techniques for the dynamic optimisation of the workflow, as this is executed. Distributed data processing is dynamic by nature and it is difficult to statically determine optimal concurrency and data movement methods a priori. More information is available during runtime, like data samples and sizes, which can assist in optimizing the execution plan further.

5.2 Optimization via Optimal Resource Management

The resources available to the user may vary over time and over different executions of the same workflow. It becomes paramount to be able to efficiently use all available resources to run a workflow as fast as possible during one instance of execution and predictably over different instances of execution.

5.3 Optimization of Multiple Workflows

Beyond optimising single workflows, we will work on the optimisation of multiple workflows. We will aim to find common or similar subgraphs that we can optimise once and execute once or a few times.

Chapter 6

Related work

As business processes become more data-intensive and more reliant on the use of computers, larger volumes of data are recorded faster and with greater precision. This trend has spurred significant increase in the complexity of analytics tasks. Additional difficulties arise from the need to deal with the multiple data types (e.g., relational, key-value, graph, etc.) that various services produce or consume. Modern workflows have become increasingly long and complex [6]. Also, the processing in workflows can be greatly diverse, ranging from simple data operations to implementations of complex algorithms, e.g. data mining algorithms, graph processing algorithms, etc, or custom procedures related to specific businesses. Finally, various constraints and policies may be applied to workflow execution, related to performance, deadlines, and optimisation of various dimensions, related to as efficiency, cost, fault-tolerance etc.

An important and big domain that urges for the solutions of such problems in data analytics, is the management of big scientific data. In [8] distinguished researchers recognise the need to address issues of creation, reuse, provenance tracking, performance optimization, and reliability of scientific workflows. Furthermore they note that scientific applications require workflow systems that support dynamic event-driven analyses, handling streaming data, accommodate interaction with users, intelligent assistance and collaborative support for workflow design, and enabling result sharing across collaborations. The proposed workflow model and the ongoing work on workflow management aims to the achievement of these goals.

Current research and industry try to cope with the above situation and overcome the problems of variety, complexity, size and dynamicity of analytics tasks, from an engineering and a scientific point of view. Overall, solutions have aimed to (a) create programming models, (b) develop execution engines, and (c) design workflow management systems.

Concerning programming models, modern research on big data analysis has focused mostly on employing the MapReduce [3] programming paradigm. The latter organises the processing on distributed servers, with parallel task execution and intermittent communication and data transfers between data shards and system parts, aiming at redundancy and fault-tolerance. Another programming model considered for big data analysis, is the Bulk Synchronous Parallel (BSP) model [22]. BSP has been proposed for designing parallel algorithms and it serves a purpose similar to the Parallel Random Access Machine (PRAM) model. BSP differs from PRAM by not taking communication and synchronization for granted. An important part of analyzing a BSP algorithm rests on quantifying the synchronization and communication needed. While BSP was proposed as a model for parallel processing, it is a good fit for distributed systems, too. As such, it was recently adopted by Google in the design of Pregel [11]. Its main advantage over the MapReduce model is that BSP is superior in handling graph-based and iterative computations, which are common in machine learning algorithms.

The proposed workflow model is designed with the anticipation of parallel programming, using models such as MapReduce and BSP. The workflow provides for the definition of abstract processors that realise simple and complex processing tasks; such processors can be implemented employing a parallel programming model. Moreover, the proposed workflow model provides for the parallel execution of data processing, employing such processors; the actual degree and plan of execution parallelisation can vary depending on the implementations of the processors, and can be depicted in the analysed version of the workflow. Furthermore, our intension is to enable the manipulation and the optimisation of the analysed version a priori or dynamically, so that the degree and plan of execution parallelisation can be controlled and changed before or during execution.

Concerning execution engines, MapReduce has given rise to the Hadoop Ecosystem, including a number of DBMSs that can be deployed in a distributed cloud-based environment, such as Pig [13] and Hive [21]. Hadoop++ [5] tries to cleanly extend Hadoop by applying relational DB techniques and re-using the notion of physical execution plan. Spark [19] is another execution engine, created for analytics and optimized for iterative MapReduce computations. Spark uses an in-memory data representation to avoid unnecessary storage of intermediate data and can express complex workflows with multiple MapReduce steps. Moreover, Spark includes a large library of machine-learning tools and support for SQL-like queries. Furthermore, Nephele [2] is the execution engine of Stratosphere [1], a research project which focuses on developing the next-generation Big Data Analytics Platform and addressing the shortcomings of MapReduce implementations. Stratosphere transformed to the Flink [7] project, which aims at in-memory data analytics and is related to Apache. Flink is a data processing system and an alternative to Hadoop's MapReduce component. It comes with its own runtime, rather than building on top of MapReduce. As such, it can work completely independently of the Hadoop ecosystem.

The ASAP framework, which will employ the proposed workflow model, is complementary to the above technologies, as it can be deployed on top of them, for the higher-level management of application workflows.

Concerning workflow management systems, they have emerged in order to provide easy-to-use specification of tasks that have to be performed during data analysis. An essential problem that these systems need to solve is the combination of various tools for workflow execution and optimization over multiple engines into a single research analysis / system. The field of workflow management is a relatively new field of research, but there are already some promising results.

One of the oldest research projects to deal with the general problem of querying multiple heterogeneous data sources is Artemis [16]. Artemis uses ontologies and metadata, and integrates metadata in terms of semantics. The project identifies the problem of continuous metadata integration. ASAP will tackle this problem employing methods for continuous scheduling of workflows that is adaptive to parameter calibration and requirements for deadline and millstones. Furthermore, the proposed workflow model enables the the creation and execution of associative tasks that process and integrate intermediate results.

The system HFMS [17] builds on top of previous work on multi-engine execution optimization [18]. Their study is more focused on optimization and execution across multiple engines. The design of flows, (workflows in our terminology), in HFMS is agnostic to a physical implementation. Data sets need not be bound to a data store, and operators need not be bound to an execution engine. HFMS handles flows as DAGs (i.e. Directed Acyclic Graphs) encoded in xLM, a proprietary language for expressing data flows. Alternatively, flows may be written in a declarative language (e.g., SQL, Pig, Hive) and imported. xLM captures structural information, design metadata (e.g., functional and non-functional requirements, physical characteristics like resource allocation), operator properties (e.g., type, schemata, statistics, engine and implementation details, physical characteristics like memory budget), and so on. Flows can be optimized at a logical level. HFMS uses many optimization strategies, such as operator swap, parallelization, recovery points, function shipping, data shipping, decomposition, etc. On physical level optimization occurs for single engine and multi-engine execution. The actual engines used are the Hadoop MapReduce distributed execution engine and a centralized PostgreSQL [15] database.

While their approach to multi-engine workflow optimization is on the same path as the approach we take in ASAP, there is an essential difference: HFMS focuses in local optimisations of specific operators and operations, ASAP aims at an overall optimisation of the workflow execution. Moreover, the proposed ASAP workflow model aims at modularity of workflow manipulation, expressibility of application logic, and adaptability to the user interests and role, goals that are out of the scope of the HFMS flow model definition.

Pegasus [14] is another workflow management system that allows users to easily express multi-step computational tasks. The workflows are formalized in the form of a DAX (i.e., Directed Acyclic graph in XML), in which the tasks are represented as nodes and task dependencies as edges. Pegasus offers APIs for Java, Python and Perl, offers support for MySQL, PostgreSQL, Oracle and Microsoft databases and can run on Amazon EC2 infrastructure. The Pegasus workflow description is separated from the description of the execution environment [4] and [10]. Keeping the workflow description resource-independent i.e. abstract, provides a number of benefits: (1) workflows can be portable across execution environments, and (2) the workflow management system can perform optimizations at 'compile time' and/or at 'runtime'. A drawback of the abstract workflow representation approach with compile time and runtime workflow modifications is that the workflow being executed can be different than what the user anticipated when she submitted the workflow. As a result, in Pegasus a lot of effort is devoted toward developing a monitoring and debugging system that can connect the two different workflow representations in a way that makes sense to the user.

The proposed workflow model overcomes such problems, by separating the definition of the dependancies and the processing tasks in the application logic. In this way, the user has control in the detail of execution semantics that she describes, and, moreover, the execution semantics determined by the system, through the creation of the analysed version of the workflow, do not change the dependencies in the application logic defined by the user.

Taverna [12] is an open source domain-independent workflow management system, which includes a suite of tools used to design and execute scientific workflows. Research in [23] is focused on the issue of the analysis of data from heterogeneous and 'incompatible' sources. Taverna allows user to define how her data flows between the services (web services, Java services, R scripts and so on), without having to worry how she is going to invoke these services. Also, Taverna provides several underlying tools to orchestrate tasks in a pipeline (data flow) by user. The Taverna suite is written in Java and access to databases is performed through JDBC.

While Taverna includes tools for the composition and enactment of bioinformatics workflows (for the life sciences community), the composition of workflows is done through a graphical user interface and does not provide sophisticated methods for the efficient execution of workflows.

Apache Tez [20] is an extensible framework for building high performance batch and interactive data processing applications, coordinated by YARN in Apache Hadoop. Tez improves the MapReduce paradigm by improving its speed, while maintaining the ability of MapReduce to scale to petabytes of data. Tez models data processing as a DAG, with a simple Java API used to express it. The task design in Tez is based on the Input-Processor-Output model, as in the task definition in the proposed workflow model. Typically, inputs and outputs exist in pairs; outputs generate 'DataMovementEvent(s)', which are processed by inputs. The inputs know how to process such events and how to interpret data. This is a key difference from the proposed workflow model, in which inputs and outputs of dependent tasks are not connected and do not have to exist in pairs. Our model dictates that inputs and outputs of edges, and not of tasks, represent the dependencies between tasks and realise the relation between their inputs/outputs. In this way, tasks are inherently independent from each other, allowing modular manipulation of tasks and task groups, as well as separate manipulation of task execution and dependencies in the application logic. Furthermore, Tez works over a singe dataset, whereas the proposed model, in the ASAP framework, enables the definition of workflows on multiple, distributed and heterogeneous datasets.

Finally, other projects have also dealt with the problem of workflow definition and execution. The Stratosphere project [1] tackles the challenge of executing workflows with the PACT programming model, based on the Nephele execution engine [2]. This approach introduces the notion of workflows in cloud-based systems, but the solution is not mature enough to give the necessary efficiency or full-fledged capabilities of adaptive execution. Furthermore, GraphX [9] is a distributed graph processing framework on top of Spark. It provides an API for expressing graph computation that can model the Pregel abstraction. It also provides an optimized runtime for this abstraction.

Chapter 7

Example Use Cases

The Telecommunication Analytics and the WebAnalytics applications include many possible query workflows, on-line queries, off-line long-running computations and ad-hoc analytics. This chapter presents several indicative use cases selected for relevance to the ASAP research.

7.1 Detecting and predicting traffic jams

An important use case of telecommunications analytics is the detection and prediction of traffic jams. The telecommunication data are anonymised at regular times. The use case involves the processing of the anonymized location data for clustering along time and space. Figure 7.1 depicts the respective workflow. Data with respect to location and time predicates are selected and joined. Then parallel processing, *calc1* and *calc2*, calculate ranges of space coordinates and time periods, respectively. These are further processed to cacluate speed, *calc3*. The speed values are input to an implementation of the k-Means algorithm that performs clustering according to a pre-defined set of criteria. Criteria may include specific area, the cut-off speed, or other parameters of the clustering algorithm. The results of the clustering algorithm are stored to several databases (relational and graph databases). The computed and stored clusters can be queried to discover traffic that occurs with regularity (detecting transport system bottlenecks) or without any regularity (anomalies, car accidents etc).

```
{"DATA": {
        "constraints": {
                "data_info": {
                        "attributes": [
```

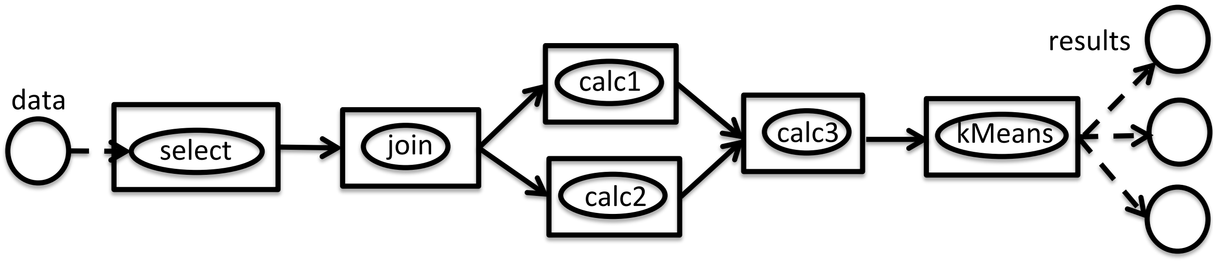


Figure 7.1: Workflow for the detection and the prediction of traffic jams

```
                                {"customer_id": {"type": "Varchar(15)"}}
                                   ,
                                {"coord": {"type": "(Integer, Integer)"}
                                   },
                                {"time": {"type": "Integer"}}
                        ]}}
        }}
```

Limiting the scope of analysis:

```
{"SELECT": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["customer_id", "coord", "
                                   time"]}},
                "output": {
                        "data_info": {
                                "attributes": ["customer_id", "coord", "
                                   time"]}},
                "op_specification": {
                        "algorithm": {
                                "select": {
                                        "select_condition": [
                                                "lb < input.coord < rb",
                                                "st < input.time < et"
                                        ]}}}}
        }}
```

where lb - left bound, rb - right bound, st - start time, et - end time

```
{"JOIN": {
        "constraints": {
                "input1": {
                        "data_info": {
                                "attributes": ["customer_id"]}},
                "input2": {
                        "data_info": {
                                "attributes": ["customer_id"]}},
                "output1": {
                        "data_info": {
                                "attributes": ["customer_id", "coord1",
                                   "coord2", "time1", "time2"]}},
                "output2": {
                        "data_info": {
                                "attributes": ["customer_id", "coord1",
                                   "coord2", "time1", "time2"]}},
                "op_specification": {
                        "algorithm": {
                                "join": {
                                        "join_condition":
                                          "input1.customer_id=input2.
                                             customer_id"}}}}
        }}
```

```
{"CALC1": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["coord1", "coord2"]}},
                "output": {
                        "data_info": {
                                "attributes": ["scoord"]}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "scoord",
                                        "calc_expression": "coord1-coord
                                           2"}]}}}
        }}
```

```
{"CALC2": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["time1", "time2"]}},
                "output": {
                        "data_info": {
                                "attributes": ["stime"]},
                                "algorithm": {
                                        "sort": {
                                                "sortingOrder": ["stime"
                                                  ]}}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "stime",
                                        "calc_expression": "time1-time2"
                                           }]}}}
        }}
```

```
{"CALC3": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["scoord", "stime"]}},
                "output": {
                        "data_info": {
                                "attributes": ["speed"]}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "speed",
                                        "calc_expression": "scoord/stime
                                           "}]}}}
        }}
```

```
{"K-MEANS": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["speed"]}},
                "output1..n": {
                        "data_info": {
                                "attributes": ["speed", "type"]}},
                "op_specification": {
                        "algorithm": {
                                "clustering": {
                                        "criteria": "speed_limits"}}}}
        }}
```

speed_limits for different types of users (pedestrians, cyclists, drivers)

7.2 Peak detection

Another important use case is the detection of peaks in the telecommunication traffic. This use case focuses on a dataset named Call Detail Records (CDR), which stores records of calls and is anonymised. CDR contains only recent data, e.g. the data of the last day. The use case involves processing of the anonymized CDR data by first selecting a spatial region and a temporal period (*select*). For this region and period, the number of calls is calculated (*calc1*). Data and calculations from CDR are archived (*archive*) in other storage (*history*). After calls are count, the application proceeds with algorithmic processing that detects peaks (*calc2*). The objective of this processing is to detect peaks in load, according to a set of criteria. Criteria may include the minimum size of a region and/or period, the cut-off distance, or other parameters for selecting regions and periods. These parameters should be adjustable by the analytics engineer, marketing expert, etc., who uses the peak analysis results. The results of this workflow are added to a database (relational or graph DBMS) that contains peaks detected in previous data. The database of peaks can then be queried by a user to discover clusters of calls that occur with regularity e.g., every week, discover clusters of calls that occur without any regularity, ; or similar ad-hoc queries based on the pre-computed peak data. The workflow for this use case is shown in Figure 7.2.

```
{"DATA": {
        "constraints": {
                "data_info": {
```

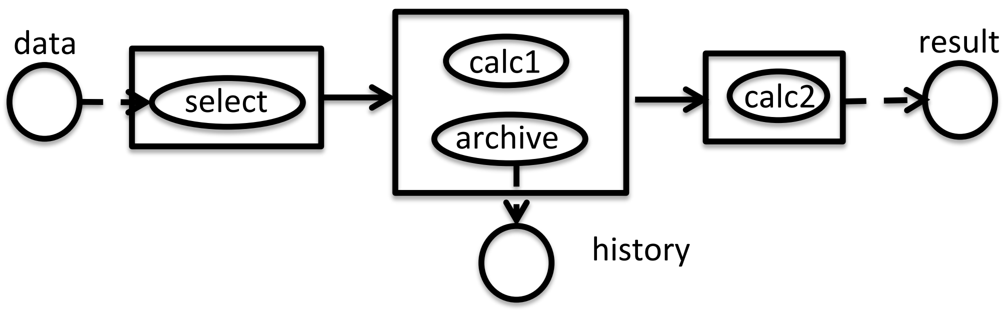


Figure 7.2: Workflow for the detection of peaks

```
                        "attributes": [
                                {"customer_id": {"type": "Varchar(15)"}}
                                   ,
                                {"coord": {"type": "(Integer, Integer)"}
                                   },
                                {"time": {"type": "Integer"}},
                                {"duration": {"type": "Integer"}},
                                {"tel_number": {"type": "Varchar(12)"}}
                        ]}}
        }}
```

Limiting the scope of analysis:

```
{"SELECT": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["customer_id", "coord", "
                                   time"]}},
                "output": {
                        "data_info": {
                                "attributes": ["customer_id", "duration"
                                   , "tel_number"]}},
                "op_specification": {
                        "algorithm": {
                                "select": {
                                        "select_condition": [
                                                "lb < input.coord < rb",
                                                "st < input.time < et"
                                        ]}}}}
        }}
```

where lb - left bound, rb - right bound, st - start time, et - end time

```
{"CALC1": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["customer_id"]}},
                "output": {
                        "data_info": {
                                "attributes": ["scoord"]}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "count",
                                        "calc_expression": "count()"}]}}
                                           }
        }}
```

```
{"AVERAGE": {
        "constraints": {
                "input1": {
                        "data_info": {
                                "attributes": ["count"]}},
                "input2": {
                        "data_info": {
                                "attributes": ["calls_num", "exp_val"]}}
                                   ,
                "output": {
                        "data_info": {
                                "attributes": ["count"]}},
                "op_specification": {
                        "algorithm": {
                                "update": ["exp_val"],
                                "append": [{
                                        "to": "calls_num",
                                        "what": "count"
                                        }]}}}
        }}
```

```
{"CALC2": {
        "constraints": {
                "input1": {
                        "data_info": {
                                "attributes": ["count"]}},
                "input2": {
                        "data_info": {
                                "attributes": ["exp_val"]}},
                "output": {
                        "data_info": {
                                "attributes": ["is_peak", "diff"]}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "is_peak",
                                        "calc_expression": "(count-
                                           exp_val) > tolerance"},
                                        "calc_attr": "diff",
                                        "calc_expression": "count-
                                           exp_val"}]}}
        }}
```

7.3 NLP-classification

This use case captures a typical form of the current IMR Web analytics pipeline. Figure 7.3 presents a workflow for this use case. Data are selected from the document store based on some conditions (*select*). These data are processed in order to extract some text (*calc*), and the extracted text is moved to a different data store and stored there with other text and annotations (*move*). The output data are further processed via NLP-classification.

```
{"DATA": {
        "constraints": {
                "data_info": {
                        "attributes": [
```

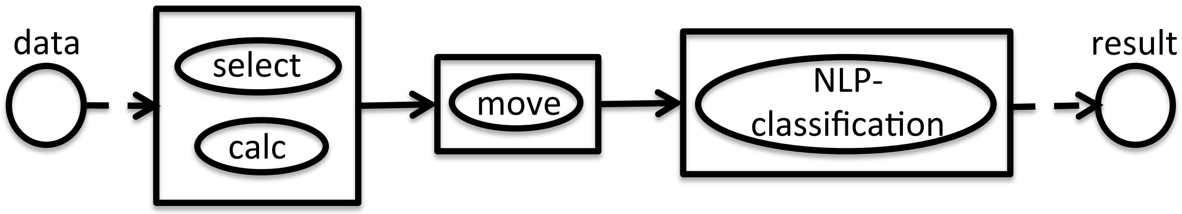


Figure 7.3: Workflow for NLP-classification

```
                                {"text": {"type": "Varchar"}}
                        ]}}
        }}
```

Selecting documents from the document store:

```
{"SELECT": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["text"]}},
                "output": {
                        "data_info": {
                                "attributes": ["text"]}},
                "op_specification": {
                        "algorithm": {
                                "select": {
                                        "select_condition": [
                                                "q in input.text"
                                        ]}}}}
        }}
```

where q - the submitted initial query

CALC stage extracts the textual content from the documents, then appends the extracted plain texts to the annotated documents as additional metadata.

```
{"CALC": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["text"]}},
                "output": {
                        "data_info": {
                                "attributes": ["text", "annotation"]}},
                "op_specification": {
                        "algorithm": {
                                "calc": [{
                                        "calc_attr": "annotation",
                                        "calc_expression": "extract_text
                                           ()"}]}}}
        }}
```

```
{"MOVE": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["text", "annotation"]},
                        "engine": {
                                "DB": "Elasticsearch"}},
                "output": {
                        "data_info": {
                                "attributes": ["text", "annotation"]},
                        "engine": {
                                "DB": "HDFS"}},
                "op_specification": {
                        "algorithm": {
                                "convert": [{
                                        "from": "Elasticsearch",
                                        "to": "HDFS"}]}}}
        }}
```

In the final stage of this use case performs NLP classification on this data. The stage also appends classification results to the annotated documents as additional metadata.

```
{"NLP-classification": {
        "constraints": {
                "input": {
                        "data_info": {
                                "attributes": ["text", "annotation"]}},
                "output": {
                        "data_info": {
                                "attributes": ["text", "annotation", "
                                   classification"]}},
                "op_specification": {
                        "algorithm": {
                                "classification": {
                                        "technique": "tfâĂŞidf"}}}}
        }}
```

Chapter 8

Conclusion

This document describes the proposed workflow model for the expression of analytics tasks on Big Data. This includes the declaration of the workflow and the accompanying execution semantics. The model enables the separation of task dependencies from task functionality. Using the proposed model, a user is able to express a variety of application logics and to set her degree of control on the execution of the workflow. Furthermore, we make an initial discussion on methods that are necessary in order to manipulate the workflow with the further goal to optimise its execution. Finally, we depict the proposed workflow model on specific use cases from the telecommunication and web analytics domains.

Bibliography

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[2] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51(1), January 2008.

[4] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 2014. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[5] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[6] M. Ferguson. Architecting a big data platform for analytics, 2012.

[7] Apache flink. `http://flink.apache.org/`.

[8] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *IEEE COMPUTER VOL*, 40(12):24–32, 2007.

[9] Apache graphx. `https://spark.apache.org/graphx/`.

[10] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, pages 22:1–22:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[11] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, pages 135–146, New York, NY, USA, 2010. ACM.

[12] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.

[13] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD ’08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[14] Pegasus. `http://pegasus.isi.edu/`.

[15] Postgresql. `http://www.postgresql.org/`.

[16] R. Tuchinda, S. Thakkar, Y. Gil and E. Deelman. Artemis: Integrating scientific data on the grid. In *Conference on Innovative Applications of Artificial Intelligence (IAAA)*, page 25.

[17] A. Simitsis, K. Wilkinson, U. Dayal, and Meichun Hsu. Hfms: Managing the lifecycle and complexity of hybrid analytic data flows. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1174–1185, April 2013.

[18] Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, pages 829–840, New York, NY, USA, 2012. ACM.

[19] Apache spark. `https://spark.apache.org/`.

[20] Apache tez. `http://hortonworks.com/hadoop/tez/`.

[21] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[22] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[23] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan R. Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole A. Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(Webserver-Issue):557–561, 2013.

FP7 Project ASAP

Adaptable Scalable Analytics Platform

End of ASAP D5.1 Workflow management model

WP 5 – Adaptive Data Analytics

Nature: Report

Dissemination: Public