FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D5.3
Data Processing Deployment

WP 5 – Adaptive Data Analytic

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	19 Aug 2016	V. Kantere, M. Filatov	Initial Version
0.2	25 Aug 2016	V. Kantere, M. Filatov	First Revision
0.3	28 Aug 2016	V. Kantere, M. Filatov	Second Revision
1.0	31 Aug 2016	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.

Abstract This deliverable is a report on the optimization module of the Platform of Analytics Workflows (PAW). This module enables both single-workflow and multi-workflow optimization. The report first gives a quick overview of the PAW architecture, then delves into the implementation details of optimization algorithms and showcases the efficiency of the proposed optimization techniques, applied on three real-world applications and their data, as well as on a synthetic benchmark.

Keywords Analytics Workflows; Multi-engine Systems; Multi-workflow Optimization.

Contents

List of Figures

List of Tables

1 Introduction

Enterprises today employ a variety of data repositories and processing engines to meet their needs for analytics. In addition to an SQL engine, a business might use a Map-Reduce engine, a scripting engine, and so on. This diversity of systems enables rapid development of new analytics, but also increases the management complexity. Even a simple analytics environment comprising a data warehouse and an ETL system might include many analytics scripts. This is a management challenge and, as more

repositories and processing engines are added to the mix, the complexity increases substantially.

In such a heterogeneous environment, analytics programs and computations span multiple execution engines and storage repositories and they typically form data flows, which we call workflows. Due to the importance and complexity of workflows, Workflow Management Tools constitute a multi-million dollar market (e.g., Forrester research [11]). There is a plethora of commercial ETL tools available. The traditional database vendors provide ETL solutions along with the DBMSs: IBM with InfoSphere Information Server [15], Oracle with Oracle Warehouse Builder [19], and so on. There also exist independent vendors that cover a large part of the market (e.g., Informatica with Powercenter [14] and Taverna [25]). Nevertheless, an in-house development and processing of ETL workflows is preferred for many data warehouse projects over an ETL tool, due to the significant cost of purchasing and maintaining latter. The usage of existing commercial solutions comes with a major drawback. Each one of them follows a different design approach, offers a different set of operators, and provides a different internal language to represent essentially similar functions.

Although Extract-Transform-Load (ETL) tools are available in the market for more than a decade, only in the last few years researchers started to realize the importance of this field. There have been several efforts towards (a) modeling analytics tasks and automation of the design process (e.g., [6]), (b) monitoring workflow execution over multiple engines (e.g., [17]), (c) optimization of specific workflow parts (e.g., optimization of the loading phase in RiTE [26], iterative parts in Stratosphere [1] and Apache Flink [10], scheduling policies [12], parallel collection processing in Emma [2] and so on), and (d) some first results towards the optimization of the whole workflow (e.g., HFMS [23]). In this report we present a module, that implements algorithms of optimizing the entire workflow, so the last point (d) in this list.

Complementary to the above, all these works provide methods of a single-workflow optimization, while our module also enables a joint optimization of several workflows. We demonstrate a novel technique for multi-workflow optimization that is implemented as part of our system called PAW (Platform for Analytics Workflows), a fully open-source platform[1] for the design, analysis and execution of analytics workflows. To the best of our knowledge, there is no previous work on multi-workflow optimization. The first version of PAW is presented in deliverable D5.2 [28]. A workflow created in PAW is prepared for execution in three steps: First, the tasks are analyzed and the workflow is augmented with associative tasks; the new version of the workflow, which we call the *analyzed* workflow. Second, workflows are manipulated by swapping, composing/decomposing and factorizing/distributing transitions, in order to achieve workflows that have equivalent outputs with their original state, but have a form that can result in optimized execution. Third, PAW schedules the execution of a set of workflows following the novel technique of multi-workflow optimization. This technique is based on the joint execution of the common parts of two or more workflows.

[1] Source available in `https://github.com/project-asap/workflow`

1.1 Overview of PAW

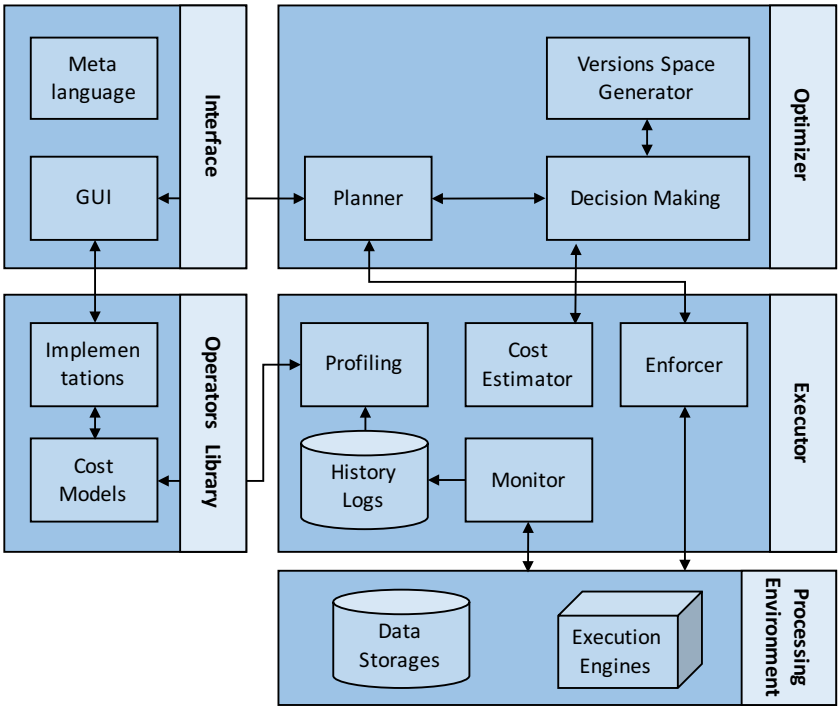


Figure 1: The architecture of PAW

PAW is part of Adaptable Scalable Analytics Platform (ASAP) [3], but it can also stand as an independent tool for workflow management and optimization. Figure 1 depicts the architecture of PAW. The latter consists of four layers: Operators Library, Interface, Optimizer, and Executor. These provide for workflow design, optimization, and execution dispatch, respectively. Workflows are executed on a set of *execution engines* and *storage repositories* of the multi-engine environment.

Operators library. This library contains operators, and their corresponding *implementations* with *cost functions*. The operators are classified as, either logical operators, which perform the core analytics jobs over the data, or the associative operators, which serve as 'glue' between different engines and perform move and transformation operations.

Interface. The *GUI* allows users to interactively create and/or modify a workflow, and add new operators to the *Library*. The user designs a workflow graph in the interactive tool and describes data and operators in the *Tree-metadata language*, which captures structural information, operator properties (e.g., type, data schemas, statistics, engine and implementation details, physical characteristics like memory budget), and so on. The metadata tree is user extensible. To allow for extensibility, the first levels of the metadata tree are predefined. Users can add their ad-hoc subtrees to define their custom data or operators. Figure 2 shows the generic metadata tree for an operator.

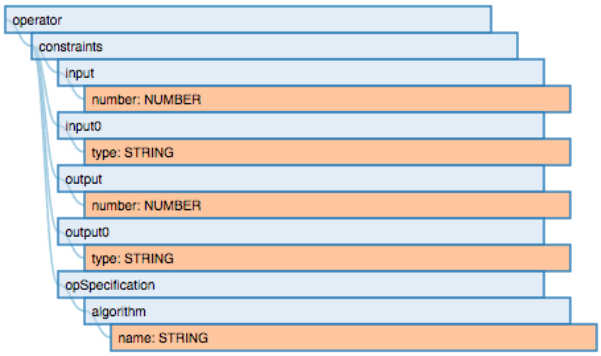


Figure 2: The generic metadata tree for operator

Optimizer. The orchestration of the optimization process is performed by the *Planner*. It takes as an input a workflow from the *Interface* and sends it to the *Decision Making* module, that returns back an optimized version of a workflow. All possible versions are produced in the *Versions Space Generator* and their costs are estimated by the *Cost Estimator*. The *Decision Making* module chooses the version with the minimal cost as an optimal one.

Executor. The executor performs several tasks. The *Enforcer* schedules workflows for execution, generates executable code and dispatches workflow fragments to execution engines. The *Monitor* observes the system state, tracks the progress of executing workflows and stores *History Logs* of runs. These logs are used to construct more precise *cost functions* of operators through the *Profiling* module. As an execution system, PAW uses IReS [16].

PAW implements a novel workflow model, that was presented in deliverable D5.1 [29]. A workflow W is a directed, acyclic graph (DAG) $G = (V, E)$. The vertices V represent data processing tasks and the edges E represent the flow of data. Each task is a set of *inputs*, *outputs* and an *operator*. Data and operators need to be accompanied by a set of metadata, i.e., properties that describe them. Such properties include input data types and parameters of operators, the location of data objects or operator invocation scripts, data schemas, implementation details, engines etc.

1.2 Purpose of the document

This document serves as a report on the optimization module of PAW and accompanies its prototype implementation. Its purpose is to delve into the details of optimization techniques and showcases their efficiency. This includes detailed description and pseudo-code of optimization algorithms and a benchmark suited for the problem of experimenting with a broad range of workflows. Furthermore, we demonstrate the work of WMT on specific use-cases from D9.2 [4]. Finally, we evaluate optimization algorithms on a set of experimental tests.

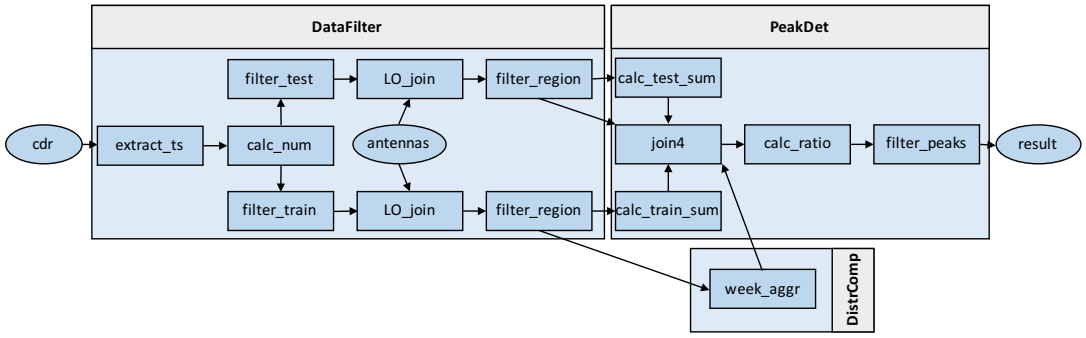


Figure 3: Original 'Peak Detection' workflow

1.3 Document structure

The rest of this document is structured as follows:

- Chapter 2 gives a brief overview of the workflow model and states a problem of optimization. Moreover, it gives a motivation example driven by the use-case scenario of D9.2 [4].
- Chapter 3 presents a single-workflow optimization, with respect to time efficiency. This chapter includes operators categorization and based on it heuristics.
- Chapter 4 presents a thorough technique for the multi-workflow optimization.
- Chapter 5 gives details on the benchmark suited for the problem of experimenting with a broad range of workflows. It provides a principled way for constructing workflows. In this chapter we propose a categorization of workflow structures, which covers frequent design cases.
- Chapter 6 describes the main configuration parameters and a set of measures to be monitored in the experimental tests for the evaluation of optimization algorithms. These tests consist of both synthetic workflows, generated in proposed benchmarking tool, and workflows of real-world applications.
- Chapter 7 summarizes related work in the topic of the workflow optimization.
- Chapter 8 concludes the deliverable.

2 Problem Discussion

2.1 Motivating example

Figure 3 shows a real-world, analytic workflow from WIND, which involves processing of the anonymised Call Data Records (CDR) to populate a report on a dashboard. The

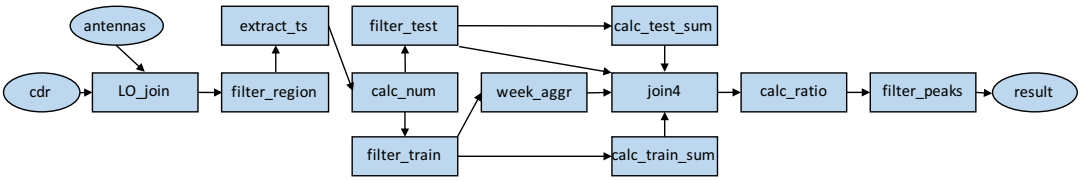


Figure 4: Optimized version of 'Peak Detection' workflow

report lists peaks in calls and their ratios to an averaged number of calls over a training period (one month). The peaks are defined by "differences from typical".

This workflow starts with extracting day of the week, hour of the day from timestamp for each call record (*extract_ts*). The *calc_num* sums calls at one-hour intervals. Then, two filters split the data to training and test datasets. The next step is to limit analysis to specific geographical regions. Then, the number of calls in training period is averaged over each mobile tower region, day in a week and hour in a day (*week_aggr*); this is the typical distribution of calls. Next, *calc_test_sum* and *calc_train_sum* produce sum of calls in each day of the test and training datasets. Then, test and training data are joined and the ratio of calls to average number is produced. The *filter_peaks* finds ratios that are over a specified limit. These peaks is the sought information.

Initially this workflow comprised from three complex UDFs (*DataFilter*, *DistrComp*, that matches to *Distribution_Computation* in the operator library, *PeakDet*). Our first goal is to optimize the workflow for performance over multiple engines; such an optimization involves the consideration of these UDFs as single operators versus a series of basic operators that as a whole can replace the UDFs. We aim to create alternative workflow versions, generate respective execution plans, estimate their costs and select an optimal plan for execution. Figure 4 shows the optimized version of this workflow that is selected by our technique for execution. In the remainder of this report, we revisit this example in order to demonstrate how the proposed optimization technique works. In our experimental study we show the performance of the original and optimized workflow on real datasets of variable size.

Next, we consider the situation when the user sends to execute two *Peak Detection* workflows with different parameters in vertices *filter_region* (different regions of exploration). Here it is our second goal: optimize such multi-workflow system. First, we perform an individual optimization for each workflow. Then, we aim to combine these two workflows to one joint workflow, so that one or more common subgraphs of original workflows, appear only once in the joint workflow and, therefore, are executed only once. Further, in the report we show how the whole space of possible joint workflows is produced and the optimized version is selected for execution. The result of optimization of such system is shown in the experiments section 6.

2.2 Context

In this section we briefly recall the context and used concepts. Thoroughly it is described in previous deliverables D5.1 [29] and D5.2 [28].

We assume a data processing environment that comprises numerous machines (e.g. a cluster) on which a variety of processing engines are installed (e.g. traditional and modern DBMSs). Each engine has access to a local data store. This environment may include multiple instances of the same engine (e.g. PostgreSQL). The processing environment takes as input logical workflows of data processing on a set of input data $\mathcal{D}_i$, producing a set of output data $\mathcal{D}_o$. The logical workflows (hereafter workflow) $W_1, \ldots, W_n$ are directed acyclic graphs (DAGs), $G_k = (\mathcal{V}_k, \mathcal{E}_k)$ where $\mathcal{V}_k$ is the set of vertices and $\mathcal{E}_k$ the set of edges of the graph G_k. Therefore, $W_k = \{G_k, \mathcal{D}_i, \mathcal{D}_o\}$. Each vertex $v \in \mathcal{V}$ represents a logical processing operator and each edge $e \in \mathcal{E}, e = (v_1, v_2)$ the flow of data between two operators v_1 and v_2. Therefore, an operator v processes the data output by other operators v', for which there exists edges $e = (v', v)$ in G.

Each workflow W corresponds to multiple alternative execution plans $\mathcal{W}_e$, in which the logical operators $v \in \mathcal{V}$ are matched 1-1 with specific implemented operator versions v_e, running on a specific engine instance; also, an execution plan may include associative operators a_e that move data from one engine to another and new edges that connect the associative operators $a_e \in \mathcal{A}_e$ with the implemented ones $v_e \in \mathcal{V}_e$. Furthermore, the input and output data are matched with specific local data stores, $\mathcal{D}_{ie}$ and $\mathcal{D}_{oe}$. Therefore, an execution plan is $W_e = \{G_e, \mathcal{D}_{ie}, \mathcal{D}_{oe}\}$, where $G_e = \{\mathcal{V}_e \cup \mathcal{A}_e, \mathcal{E}_e\}$. Every vertex in W_e has an execution cost c and the execution cost of the whole execution plan is: $c(W_e) = \sum c(v_e) + \sum c(a_e)$.

Goal. Our goal is to create and select for execution, the execution plan W_e of a workflow W with a minimal execution cost, i.e. $W_e \in \mathcal{W}_e \; s.t. \; c(W_e) \leq c(W'_e), \forall W'_e \in \mathcal{W}_e$.

Solution. We do this with the following steps:

1. For the original workflow W, we create equivalent optimized versions $\mathcal{W}_l$.
2. For each version $W_l \in \mathcal{W}_l$ we create the respective execution plans $\mathcal{W}_{le}$.
3. For each execution plan $W_{le} \in \mathcal{W}_{le}$ for all $W_l \in \mathcal{W}_l$ we estimate the respective cost $c(W_{le})$.
4. We select the execution plan with the minimal cost.

Step 1 of the above process performs the optimization of the workflow at the a logical level. The challenge is to restructure the original workflow in a manner that: (a) ensures that the resulting version is equivalent to the original one, i.e. given the same input data it produces the same output data, and (b) ensures that the resulting version will correspond to execution plans among which, at least one will have cost smaller than the cost of each of the execution plans that correspond to the original workflow, i.e. for each optimized workflow $W_l \in \mathcal{W}_l$, there exists $W_{le} \in \mathcal{W}_{le}$, so that $c(W_{le}) \leq c(W_e)$, for all $W_e \in \mathcal{W}_e$. Step 2 refers to the matching of the created optimized workflow versions $\mathcal{W}_l$ with implemented versions for each one of the involved logical

operators, and, in case two adjacent operators are matched with implementations on different engines or machines, infusion of an associative operator that moves data from one engine/machine to the other. Steps 3 and 4 refer to estimating the execution cost of all the alternative execution plans of all the optimized workflow versions, and selecting the one with the minimal cost.

Let's extend the goal from one to a set of workflows. The solution of it is the same list of steps with a supplement of these two placed between step 1 and 2:

1.1 Find common parts CPs for pairs of workflows, that are sent for execution

1.2 Combine by common parts CPs and produce all possible joint workflows $\mathcal{W}_j^l$

In the following we describe all these process steps in detail, taking a bottom-up approach, i.e. we start from the description of operators, we continue with the creation of the alternative execution plans of a workflow and the estimation of their cost, next, we presents the whole algorithm of the single-workflow optimization, then we turn to a process of finding of common parts and combining by them a pair or a set of workflows to a joint workflow, and we finish with the multi-workflow optimization algorithm.

2.3 Operators

Each logical operator has an abstract definition and one or several implementations, that can span several types of engines (i.e. one or more implementations per engine). For example, a traditional logical *join* has an abstract definition, and can be implemented for a relational DBMS and a NoSQL database. A logical operator can correspond to a simple operation or to complex algorithmic computation. In both cases, in order to use a logical operator in an execution plan, we need to have a tailored implementation for every engine on which it is going to be executed. The definition of a logical operator includes: the type and number of inputs, the number and type of outputs, and mandatory and optional attributes of the operator; it can also include functions to compute cardinality and processing cost. For example, operators *filter* and *calc* from our library both have one input and one output and are defined for relational data. The *filter* operator retrieves tuples from a relation, limiting the results to only those that meet a specific criterion and the *calc* operator produces tuples supplemented with additional attribute, calculated by a specific expression. Formally they are defined as follows:

$O(filter, I) = \{r \mid r \in I \ \wedge \ FilterPredicate(r)\}$

$O(calc, I) = \{r \cup \{attr : val\} \mid r \in I \ \wedge \ val := CalcExpression(r)\}$

Logical operators are categorized as:

- **Blocking operators**, which require knowledge of the whole dataset, e.g., a *groupBy*, a *join* or a *sort*.
- **Non-blocking operators**, which process each tuple separately, e.g., operators *filter*.

- **Restrictive operators** output a smaller data volume than the incoming data volume, e.g. *filter*.

Logical operators and their implementations are stored in the library. Table 5 shows implemented operators from the library of our real processing environment and the categorization of the respective logical operators. Users can register their own new logical operators together with respective implementations, or new implementations of existing logical operators. Some examples of the operators that has been added to the library by the users of our system are: *PeakDet* (peak detection), *Stereotype_Classification*, *Distribution_Computation* and *User_Profiling*, which are complex operators used in telecommunication data analytics. These operators have been decomposed to a set of basic operators (Figure 3) for more possibilities for optimization. Some of the operators can be used associatively in the creation of execution plans of workflows. These are the operators that move data from one engine to another, or one machine to another. Table 5 shows 2 such operators, *Move_Hive_Postgres* and *Move_Postgres_Hive*, which move a dataset from Hive to PostgreSQL and the opposite.

2.4 Execution plans of workflows

In this section we describe the creation of the execution plans for a workflow and the estimation of their cost.

2.4.1 Creation of execution plans

The creation of an execution plan W_e for a workflow $W = \{\mathcal{V}, \mathcal{E}\}$ is achieved in two steps:

1. Every logical operator $v \in V$ is matched with a respective implemented operator, v_e that can be executed on a specific machine and engine.

2. For every pair of adjacent implemented operators (v_{e1}, v_{e2}) we check if these operators are on the same engine and machine; if they are not on the same engine or machine, we check if there is an associative operator a_e that can be used to transfer the output data of v_{e1} to the engine and machine where v_{e2} resides, so that the latter can use it as input. If this is not possible, the creation of the execution plan fails. If it is, then:

3. The set of edges $\mathcal{E}_e$ is instantiated with edges that correspond 1-1 with the edges $\mathcal{E}$ of the original graph G. The edge $(v_{e1}, v_{e2}) \in \mathcal{E}_e$ is substituted with two new edges (v_{e1}, a_e) and (a_e, v_{e2}). The execution plan is a new DAG with vertices the union of all implemented operators $\mathcal{V}_e$ and all associative operators A_e, and edges $\mathcal{E}_e \leftarrow (\mathcal{E}_e - \bigcup_{(v_{e1}, v_{e2})}) \cup \bigcup_{\{(v_{e1}, a_e), (a_e, v_{e2})\}}$.

Operator	Blocking	Non-blocking	Restrictive
Filter		x	x
Calc		x	
Join	x		
Filter_Calc		x	x
Filter_Join	x		x
Projection		x	x
groupBy Sort	x		
DataFilter	x		
PeakDet	x		
KMeans	x		
Stereotype_ Classification	x		
Distribution_ Computation	x		
User_Profiling	x		
TF-IDF	x		
lr_train	x		
lr_classify	x		
Move_Hive_Postgres	x		
Move_Postgres_Hive	x		
w2v_train	x		
w2v_vectorize	x		
grep	x		
Join4	x		
Left_Outer_Join	x		

Figure 5: Categorization of operators

A logical operator (e.g. logical join) may have various implementations for the same engine, (e.g. merge-sort) and across multiple engines (e.g., relational *join*, Hadoop/Pig *join*). Also, a specific implementation may be or not available in a specific machine (even if it runs the respective engine). Therefore, we first try to match each logical operator of a workflow with all respective implementations that exist in the processing environment. This action results possibly in the matching of parts of the workflow to different machines or different engines (that reside in the same or different machines). In order to achieve the execution of the workflow as a whole by executing its parts in the matched machines and engines, we need to transfer data from one machine/engine to another. This can be achieved with the infusion of associative implemented operators that move data (as described in Section 2.3). If such operators do not exist, then the creation of the execution plan fails. If they exist, they are used as intermediaries between the execution of two parts of the workflow that are connected, i.e. the output

of a workflow part is the input to the other.

2.4.2 Cost estimation of an execution plan

As mentioned, the cost of each execution plan W_e created by the process of Section 2.4.1 is the summation of the cost of every implemented operator and all infused associative operators: $c(W_e) = \sum c(v_e) + \sum c(a_e)$. The cost $c(v_e)$ is based on: (a) existing cost functions of the implemented operator v_e (e.g. given by the user), and (b) statistical summaries of the outputs of operators, i.e. empirical graphs of execution time versus data input size that are produced by test runs or history logs of runs of the implemented operator v_e.

The cost function (a) of an implemented operator v_e involves measures like CPU, memory (e.g., buffer sizes), I/O, and communication costs. Naturally, different implementations of a logical operator (including implementations for different engines) have different cost functions. In most cases, the developer or provider of an implemented operator v_e does not disclose a cost function (and the source code of the operator may not be available either). In such frequent cases, we treat the operator as a 'black box' and we run a series of micro-benchmarks to study the operator's behavior under different configurations. Based on the results of micro-benchmarking, we build a cost function for an implemented operator. As an extra, but optional, step, we enable users to run their workflows with a sample of their data and we use the obtained statistics to fine-tune the cost functions of employed implemented operators, before using them in cost estimation of execution plans of workflows.

The statistical summaries of the outputs of operators, (b), show, essentially, the selectivity of the operator with respect to the size of the input data. Once more, if these summaries are not given with the registration of a logical operator in the library, we create them by running micro-benchmarks using available implementations of the logical operator.

The cost $c(a_e)$ of an associative operator a_e that moves data captures different costs involved in the data moving process, like data shipping cost, the cost of initializing the target engine, the monetary cost of such an action, network bandwidth etc. Naturally $c(a_e)$ is always a function that increases with the size of the data (frequently linearly); therefore, as the data to be moved from one engine to the other increases, the respective moving cost increases too, making execution plans that include such an associative operator very expensive.

3 Single-workflow optimization

In this chapter we describe the single-workflow optimization. We describe how we create the search space of the optimization and the search algorithms.

Search space The search space of possible execution plans for a workflow is multi-dimensional. It includes 3 dimensions: (a) the dimension that represents the set of

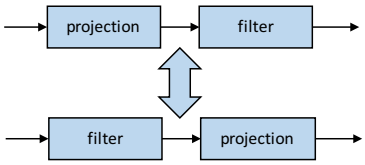


Figure 6: *Swap*ping of two vertices

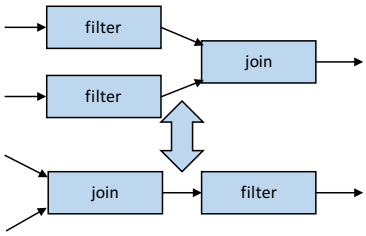


Figure 7: Example of factorization/distribution

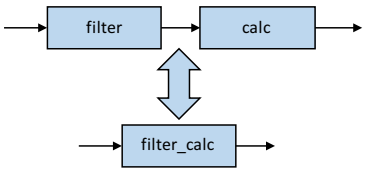


Figure 8: Composing/decomposing of filter and calc

possible equivalent versions of the original workflow, (b) the dimension that represents the set of alternative implementations of logical operators, and (c) the dimension that represents the execution of implemented operators on different machines and engines. Dimension (a) is independent from (b) and (c), whereas (b) and (c) are interdependent, as a specific implemented operator may run on a specific type of engine, and on machines where this engine is installed and the input data are stored locally. Taking the original workflow as input, our optimization technique produces equivalent versions by applying transitions on the logical operators of sets of adjacent vertices (Step 1 of the *Solution* in Section 3). In Section 3.1 we describe in detail how the equivalent versions are created.

Search techniques Following the procedure described in Section 2.4.1, every equivalent workflow (i.e. a point in dimension (a) of the search space) is matched with implemented operators (points in dimension (b)) and engines that run on the available machines (points in dimension (c)) in order to produce respective execution plans. The cost of execution plans is estimated as described in Section 2.4.2. Therefore, the search for the optimal execution plan(s) is led by the exploration of alternative equivalent versions of the workflow. We use two algorithms to explore the search space of dimension (a). The first is an exhaustive algorithm that constructs all alternative equivalent versions and their respective execution plans by iterating on the number of possible transitions. The second is a heuristic algorithm that creates only some of the alternative equivalent versions based on heuristics. In Section 3.2 we give the details of the algorithms.

3.1 Creating equivalent workflows

In this section we describe the transitions that we apply to the workflow in order to create equivalent versions that may lead to plans with smaller cost estimation.

3.1.1 Transitions

A transition is applied on a set of adjacent vertices of a workflow W and transforms it to an equivalent workflow W_l.

Definition of transitions. We introduce 5 transitions. Their input workflow is $W = \{\mathcal{V}, \mathcal{E}\}$ and their output is workflow $W_l = \{\mathcal{V}_l, \mathcal{E}_l\}$. The transition *swap* changes the order of a pair of adjacent operators v_1, v_2. The transition *compose* replaces a subgraph of the initial workflow with one operator that is capable of performing the combined operation of all the operators in this subgraph; ordinarily, *compose* is applied on a pair of of adjacent operators and replaces them with a new one capable of performing their combined operation; the transition *decompose* does the opposite of *compose* and replaces one complex operator with a subgraph of new operators that are capable to jointly perform the operation of the replaced complex operator; ordinarily *decompose* is applied in order to split one complex operator into a pair of more basic operators. Transition *factorize* is applied on a subgraph that consists of a set of operators $\mathcal{V}_f$ the output of which is input to (or the input of which is output of) one operator (we call this a *branching* operator and we denote it with v_b); it *swap*s all operators $v_f \in \mathcal{V}_f$ with v_b and composes them in one complex operator v_t that is capable of performing the operations of all operators $v_f \in \mathcal{V}_f$. Ordinarily *factorize* is applied on two operators that perform the same operation, e.g. the same *filter* and are adjacent to the same branching operator, e.g. a *join*, or (from Table 5) *Join4*, *Left_Outer_Join*, *User_Profiling*, or on operators that output copies of the processed data; in this case the transition *swap*s the pair of identical operators with the branching one and eliminates one of the first. Transition *distribute* performs the opposite of *factorize*, i.e. *swap*s a branching operator v_b with the following adjacent v_d, and replicates the latter so that one operator v_t identical to v_b is included adjacently in all paths leading to v_b. Ordinarily, *distribute* is applied to a branching operator like *join* and an operator following it, e.g. a *filter*, so that the latter can be applied separately to the inputs of the first. display an example for each one of the transitions. The following are the formal definitions of the transitions:

- $swap(W, v_1, v_2)$, $v_1, v_2 \in \mathcal{V}, (v_1, v_2) \in \mathcal{E}$: this transition is applied to a pair of adjacent operators v_1 and v_2, and produces a new workflow W' in which the positions of v_1 and v_2 have been interchanged, i.e. $\mathcal{V}_l = \mathcal{V}$ and $\mathcal{E}_l = (\mathcal{E} - \{(v_1, v_2), (v_i, v_1), (v_2, v_j)\}) \cup \{(v_2, v_1), (v_i, v_2), (v_1, v_j)\}, \forall v_i, v_j s.t. (v_i, v_1), (v_2, v_j) \in \mathcal{E}$.

- $compose(W, v_{12}, v_1, v_2)$, $v_1, v_2 \in \mathcal{V}, (v_1, v_2) \in \mathcal{E}$: this transition is applied to a pair of adjacent operators v_1 and v_2, and produces a new workflow W' in which these are replaced by a new operator v_{12}, i.e. $\mathcal{V}_l = (\mathcal{V} - \{v_1, v_2\}) \cup v_{12}$ and $\mathcal{E}_l = (\mathcal{E} - \{(v_1, v_2), (v_i, v_1), (v_2, v_j)\}) \cup \{(v_i, v_{12}), (v_{12}, v_j)\}, \forall v_i, v_j s.t. (v_i, v_1), (v_2, v_j) \in \mathcal{E}$. Composition of a set of operators can be defined in a recursive manner.

- $decompose(W, v_{12}, v_1, v_2)$, $v_{12} \in \mathcal{V}$: this transition is applied to one operator v_{12} and produces a new workflow W' in which this is replaced by a pair of adjacent operators v_1 and v_2, i.e. $\mathcal{V}_l = (\mathcal{V} - v_{12}) \cup \{v_1, v_2\})$ and $\mathcal{E}_l = (\mathcal{E} - \{(v_i, v_{12}), (v_{12}, v_j)\}) \cup \{(v_1, v_2), (v_i, v_1), (v_2, v_j)\}$, $\forall v_i, v_j s.t. (v_i, v_{12}), (v_{12}, v_j) \in \mathcal{E}$. Decomposition of a set of operators can be defined in a recursive manner.

- $factorize(W, v_b, \mathcal{V}_f, v_t)$, $\mathcal{V}_f \subset \mathcal{V}, v_b \in \mathcal{V}, (v_f, v) \in \mathcal{E}$, $\forall v_f \in \mathcal{V}_f$: this transition is

applied to a set (two or more) operators v_f that are adjacent to the same operator v_b and performs swapping of v_f with v_b and composition of all $v_f \in \mathcal{V}_f$ into one operator v_t, i.e. $\mathcal{V}_l = (\mathcal{V} - \mathcal{V}_f) \cup v_t$ and $\mathcal{E}_l = (\mathcal{E} - \{(v_f, v_b), (v_i, v_f), (v_b, v_j)\}) \cup \{(v_i, v_b), (v_b, v_l), (v_l, v_j)\}, \forall v_f \in \mathcal{V}_f$ and $\forall v_i, v_j, s.t.\ (v_i, v_f), (v_b, v_j) \in \mathcal{E}$.

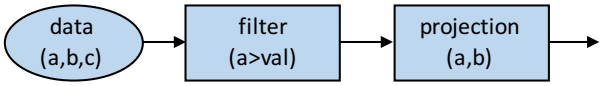
- $distribute(W, v_b, \mathcal{V}_t, v_d)$, $v_b, v_d \in \mathcal{V}, (v_b, v_d) \in \mathcal{E}$: this transition is applied to adjacent operators v_b and v_d and performs swapping of v_d with v_b and inclusion of a new operator v_l,adjacently to v_b, in all workflow paths leading to v_b, i.e. $\mathcal{V}_l = (\mathcal{V} - v_d \cup \mathcal{V}_t$ and $\mathcal{E}_l = (\mathcal{E} - \{(v_i, v_b), (v_b, v_d), (v_d, v_j)\}), \cup \{(v_i, v_t), (v_t, v_b), (v_b, v_j)\}$, $\forall v_i, v_j,\ s.t. (v_i, v_b), (v_b, v_j) \in \mathcal{E}$.

Functionality of transitions. The 5 transitions defined above are introduced because they can create optimization opportunities of parts of the input workflow.

Transition *swap* allows for pushing highly selective operators toward the root of the workflow; in this way, the size of the data that are input to the workflow, $\mathcal{D}_i$, can be significantly reduced early during the execution of the workflow, leading to lighter data load, and, therefore, faster execution, for many of the operators of the workflow.

Transitions *compose* and *decompose* allow for the replacement of complex operators with a set of simpler ones that can perform the same operation and vice versa; this creates opportunities for optimization that are adaptive to the specific environment (available machines and engines, distribution of data locally to machines and engines, size of data, and current workload of machines). Specifically, the complex operator and the respective equivalent set of operators most probably have implementations for one or a few engines. Depending on the location and distribution of the data on the machines and engines, as well as their workload, it may be best to execute either an implementation of the complex operator (naturally on one engine and machine) or the equivalent set of operators (distributed on different engines and machines). In the first case, there are optimization opportunities created by the specific engine on which the complex operator is executed, and in the second there are optimization opportunities that are created by the distribution of execution of operators on a combination of machines and engines.

Transition *factorize* allows for the replacement of multiple identical operators that (i) all feed one *branching* operator and take as input different datasets, with one such operator that is performed on the sum of the datasets, after the latter have been processed by the *branching* operator, or (ii) oppositely, all are fed by the same *branching* operator, with one operator performed on the input data of the *branching* operator. The optimization derives from the fact that the operation of the replaced operators is performed only once instead of several times, and, moreover, on a reduced in size aggregated dataset. Inversely, transition *distribute* allows for the replacement of one operator with multiple identical ones, which are distributed on the input (or output) paths of a preceding (or succeeding) *branching* operator. The optimization opportunity is created either by the parallelization of the execution of the operation of the identical operators, their distribution over the input dataset, or even by the reduction of size of the aggregated input data due to their being pushed toward the root of the



schemas	filter	projection
functionality	a	∅
generated	∅	∅
projected-out	∅	c
input	a,b,c	a,b,c
output	a,b,c	a,b

Figure 9: A workflow example and a corresponding schemas table of operators and data of this workflow

workflow.

As an example, the original workflow of Figure 3, designed by the user, consists of three complex operators (*DataFilter*, *DistrComp*, that matches to *Distribution_Computation* in the operator library, *PeakDet*) implemented on a scripting engine; the optimization technique *decompose*s these operators to basic operators that have implementations in RDBMSs. This decomposition gives more possibilities for workflow versioning through the application of transitions. In the optimized version of Figure 4 vertices *LO_Join* and *filter_region* of decomposed UDF *DataFilter* are first *swap*ped with *filter_train*, *filter_test* and then *factorize*d over *calc_num*, their common predecessor.

Another example is a real workflow from a sales use-case, shown in Figure 33 and described in Section 6.2.2. During the optimization process a vertex *filter by prod&reg* is twice *distribute*d over *join2 by prod&reg* and *join1 by prod&reg*. Then it is *decompose*d to *filter by prod* and *filter by reg*; the second vertex of these two is *swap*ped several times towards the input data source *tweets*. The vertex *filter by prod* is composed with *select product* and the produced vertex is called *select&filter product*. Figure 34 displays the optimized workflow version.

3.1.2 Schemas of operators

Apart from the input and output data schemas, each logical operator is accompanied by the following schemas:

- Functionality schema. This schema is a list of attributes, being a subset of (the union of) the input schema, denoting the attributes which take part in the computation performed by the task.
- Generated schema. This schema is a list of attributes, being a relative complement of input schema in output schema and are generated due to the processing of the tasks.

swap	filter	calc	join	filter_calc	filter_join	projection
filter	✓	✓	✓	✓	✓	If $filter.fs \cap projection.pos = \emptyset$
calc	If $calc.gs \cap filter.fs = \emptyset$	If $calc1.gs \cap calc2.fs = \emptyset$	If $calc.gs \cap join.fs = \emptyset$	If $calc.gs \cap filter\_calc.fs = \emptyset$	If $calc.gs \cap filter\_join.fs = \emptyset$	If $calc.fs \cap projection.pos = \emptyset$
join	If $filter.fs \subset join.i1s$ or $filter.fs \subset join.i2s$	If $calc.fs \subset join.i1s$ or $calc.fs \subset join.i2s$	If $join1.fs \subset join2.i1s$ or $join1.fs \subset join2.i2s$	If $filter\_calc.fs \subset join.i1s$ or $filter\_calc.fs \subset join.i2s$	If $filter\_join.fs \subset join.i1s$ or $filter\_join.fs \subset join.i2s$	If $join.fs \cap projection.pos = \emptyset$ and $projection.pos \subset join.i1s$ or $projection.pos \subset join.i2s$
filter_calc	If $filter\_calc.gs \cap filter.fs = \emptyset$	If $filter\_calc.gs \cap calc.fs = \emptyset$	If $filter\_calc.gs \cap join.fs = \emptyset$	If $filter\_calc1.gs \cap filter\_calc2.fs = \emptyset$	If $filter\_calc.gs \cap filter\_join.fs = \emptyset$	If $filter\_calc.fs \cap projection.pos = \emptyset$
filter_join	If $filter.fs \subset filter\_join.i1s$ or $filter.fs \subset filter\_join.i2s$	If $calc.fs \subset filter\_join.i1s$ or $calc.fs \subset filter\_join.i2s$	If $join.fs \subset filter\_join.i1s$ or $join.fs \subset filter\_join.i2s$	If $filter\_calc.fs \subset filter\_join.i1s$ or $filter\_calc.fs \subset filter\_join.i2s$	If $filter\_join1.fs \subset filter\_join2.i1s$ or $filter\_join1.fs \subset filter\_join2.i2s$	If $filter\_join.fs \cap projection.pos = \emptyset$ and $projection.pos \subset filter\_join.i1s$ or $projection.pos \subset filter\_join.i2s$
projection	✓	✓	✓	✓	✓	✓

Figure 10: Applicability of swap for pairs of operators

- Projected-out schema. This schema is a list of attributes that belongs to the input schema, but is not propagated further after processing.

These schemas are used to validate the correctness of a workflow and to determine the possibility of transition application. The latter is described in Section 3.1.3. Figure 9 shows a workflow consisting of an input data source (data) and two vertices (filter and projection), and the table with the schemas of operators used in this workflow.

3.1.3 Applicability of transitions

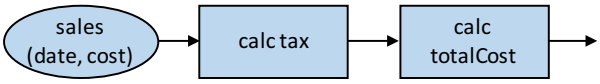
The transitions can be applied only to sets of operators that confirm to specific conditions. When a new operator is defined and added to the operator library, the developer or provider of the operator has to declare if it conforms to the conditions of a transition by filling in a specific table template. This is a $n \times n$ table where n is the number of operators in the library; the row in this table indicates the operator that is predecessor and the column indicates the operator that is successor; the cell represents a condition for the applicability of the transition for this ordered pair of operators, the condition is based on schemas of operators described above. Filling in this table is a step of the process of adding operators to the library. Defining conditions for all pairs is not necessary; if the condition for some pair of operators is not defined, then the transition for them cannot be applied. Figures 10, 11 and 12 display a part of the applicability tables for *swap*ping, (de)composing and factorizing/distributing, respectively. In the following,

compose / decompose	filter	calc	join	filter_calc	filter_join	projection
filter	filter	filter_calc	filter_join	filter_calc	filter_join	✗
calc	filter_calc	calc	✗	filter_calc	✗	✗
join	filter_join	✗	join	✗	filter_join	✗
filter_calc	filter_calc	filter_calc	✗	filter_calc	✗	✗
filter_join	filter_join	✗	filter_join	✗	filter_join	✗
projection	✗	✗	✗	✗	✗	✗

Figure 11: Applicability and a resulting operator of compose/decompose for pairs of operators

factorize / distribute	join	filter_join
filter	✓	✓
calc	If $calc.gs \cap join.fs = \emptyset$	If $calc.gs \cap filter\_join.fs = \emptyset$
join	If $join1.fs \subset join2.i1s$ or $join1.fs \subset join2.i2s$	If $filter\_join.fs \subset join.i1s$ or $filter\_join.fs \subset join.i2s$
filter_calc	If $filter\_calc.gs \cap join.fs = \emptyset$	If $filter\_calc.gs \cap filter\_join.fs = \emptyset$
filter_join	If $join.fs \subset filter\_join.i1s$ or $join.fs \subset filter\_join.i2s$	If $filter\_join1.fs \subset filter\_join2.i1s$ or $filter\_join1.fs \subset filter\_join2.i2s$
projection	✓	✓

Figure 12: Applicability of factorize and distribute transitions



schemas	calc tax	calc totalCost
functionality	cost	cost, tax
generated	tax	totalCost
projected-out	∅	∅
input	date, cost	date, cost, tax
output	date, cost, tax	date, cost, tax, totalCost

Figure 13: An example of a workflow, where *swap* can't be applied

we discuss the conditions for each transition.

Swap. One would normally anticipate that *swap*ping is already covered by traditional query optimization techniques. However, this is not true. On the contrary, we have observed that the swapping of operators deviates from the equivalent problem of "pushing vertices downward" as we normally do in the execution plan of a relational query. The major reason for this deviation is the presence of functions, which potentially change the semantics of attributes. Relational algebra does not provide any support for functions; still, the "pushing" of operators should be allowed in some cases, whereas, in some others, it should be prevented. *Swap*ping of operators in a workflow is a generalization of "pushing"operators towards both the start or the end of a workflow.

Formally, we allow the *swap*ping of two operators v_1 and v_2 if the following conditions hold:

1. The functionality schema of v_1 and v_2 is a subset of their predecessors' output schemas (both before and after the *swap*ping).

2. The input schemas of v_1 and v_2 are subsets of their predecessors' output schemas, both before and after the *swap*ping.

Conditions 1 an 2 cover two possible situations, which we describe with examples.

Let us consider a workflow (Figure 13) with one input data source *sales* and two vertices *calc tax* and *calc totalSales*. *calc tax* computes the *tax* from the *cost* for each sale record, then *calc totalSales* sums *tax* and *cost*, and records the result to a new field *totalCost*. The functionality schema of *calc totalSales* contains *tax* and *cost*. If *swap*ping is attempted, *calc totalSales* precedes *calc tax* and the schema of *sales* data (*date*, *cost*) doesn't contain *tax*; this contradicts the first condition. Thus, the *swap*ping of *calc tax* and *calc totalSales* is rejected.

Let us consider the workflow of Figure 9, in which a *projection* rejects an attribute *c* of its input schema. Then, an attempted *swap*ping produces an error since the rejected attribute *c* in the input schema of operator *filter* (now a successor of *projection*) will not be presented in the output schema of *projection*.

Factorize/Distribute. The conditions governing factorization of a set of vertices $\mathcal{V}_f$ are similar to conditions of *swap*ping. First, according to a definition of *factorize*, vertices in $\mathcal{V}_f$ are identical instantiations of the same logical operator and v_b is their common successor or predecessor. Therefore the conditions for factorization are the following:

1. The functionality schema of $\forall v_i \in \mathcal{V}_f$ and v_b, v and v_b after factorization is a subset of their predecessors' output schemas,
2. The input schemas of $\forall v_i \in \mathcal{V}_f$ and v_b, v and v_b after factorization are subsets of their predecessors' output schemas.

The distribution is governed by similar conditions; a vertex v can be cloned in a set of vertices $\mathcal{V}_t$ if:

1. The functionality schema of v and v_b, v_b and $\forall v_i \in \mathcal{V}_f$ after distribution is a subset of their predecessors' output schemas,
2. The input schemas of v and v_b, $\forall v_i \in \mathcal{V}_f$ and v_b after distribution are subsets of their predecessors' output schemas.

Compose/Decompose. Composition of two vertices, v_1 proceeding v_2, is applicable if the output schema of the new vertex is the output of v_2 and the input schema is the input of v_1. Decomposition requires that the originating vertex is a composed one, like, for example, *filter_calc*. In this case, the vertex is decomposed in two vertices *filter* and *calc*. (As described in Section 3.2, our algorithms apply transitions on workflows in which the original complex operators are initially decomposed to a set of basic operators. After this initial decomposition, attempts and application of various compositions and afterwards decompositions of operators are possible.)

3.1.4 Identification of versions

During the application of the transitions, we need to be able to discern versions from one another so that we avoid generating (and computing the cost of) the same version more than once. A workflow is a directed graph that can be topologically ordered on the basis of the predecessor-successor relationship so that an execution priority can be assigned to each vertex. In order to automatically derive vertex identifiers for all equivalent versions of a workflow we assign to each vertex the priority that stems from the topological ordering of the original workflow.

Based on the above observations, we create a version identification scheme such that: 1) each (linear) path is denoted as a string where the vertices of the path are delimited by underscores, and 2) concurrent paths are delimited by a double slash. We call the string that characterizes each version as the signature of the workflow version. Between concurrent paths, the signature of the version starts with the path including the identifier with the lowest value.

For the cases in which new vertices are derived from existing ones, we use the following rules:

- If two existing vertices a and b are *compose*d to create a new one, then the new vertex is denoted as $a + b$.
- If two existing vertices a and b are *factorize*d to create a new one, then the new vertex is denoted as $a; b$.
- If a vertex is cloned to be *distribute*d in more that one paths, then each of its clones adopts the identifier of the original vertex followed by a dot and a unique integer as a partial identifier (e.g., $a.1$; $a.2$).
- If a vertex is *decompose*d, we reuse the identifiers of its components (e.g., $a + b$ or $a; b$ can be decomposed to a and b, respectively).

3.2 Searching for optimal execution plans

In this section we present our algorithms for exploring alternative equivalent workflow versions in order to find one with an optimal execution plan.

3.2.1 Exhaustive Search

We create Basic Exhaustive Search algorithm (BES) that explores the version space exhaustively. First, BES generates a set of equivalent versions of the original workflow, in which the original complex operators are decomposed to a set of basic operators; even though there can be multiple such decompositions for one complex operator, in practice there is usually only one. Second, BES generates all possible versions by applying all the applicable transitions to every version in an iterative manner.

BES uses two lists $open$ and $close$ for keeping track of unvisited and visited versions, respectively. It also uses a version variable W_{min} for storing through iterations the workflow version having the minimum cost. The algorithm starts from all decomposed versions of the original workflow, i.e., versions W_o^i, which are given as an input parameter. At the beginning $open$ contains W_o^i and $close$ is empty. For every version W in $open$, we apply any applicable transition f to it. For every newly generated, but not already visited, version W', if the cost of it is less than the minimum cost discovered so far, then W' becomes the new W_{min}. In any case, W' is marked as visited and we proceed until there is no other version to create. Then, BES returns the optimal version W_{min}.

Clearly, the version space is finite and the algorithm terminates after having generated all possible versions and returning the optimal one.

The version space is exponentially large and in realistic environments we may need more efficient exploration methods than BES. To improve the search performance of BES we employ a set of heuristics, based on simple observations relevant to the definition of transitions. Heuristic $H1$ is based on the observation that transition *factorize* indicates that it is not necessary to try factorizing all the vertices of a workflow. Instead, a new version should be generated from an old one through a *factorize* transition that

involves only identical operators connected to a branching operator. Heuristic $H2$ is based on the observation that transition *swap* can be applied only on operators that have one input and one output.

- **H**1: Find branching operators and check if they are connected with operators that are identical instances of a logical operator. Try to *factorize* this set of operators.
- **H**2: Find (linear) paths and try to *swap* the operators in each of such paths.

Heuristics $H1$ and $H2$ accelerate the generation of the search space and are used in the exhaustive algorithm ES.

Algorithm 1: Basic Exhaustive search.

Input: An original workflow $W_o = (V, E)$
Result: A version W_{min} having the minimum cost
begin
  $W_{min} = W_o$;
  $open \leftarrow W_o$;
  $close = \varnothing$;
  while $open <> \varnothing$ **do**
    $W \leftarrow open$;
    for *all* $W' = f(W)$ **do**
      if $W' \notin open$ **and** $W' \notin close$ **then**
        if $C(W') < C(W_{min})$ **then** $W_{min} = W'$;;
        $open \leftarrow W'$;
    $close \leftarrow W$;
  return W_{min}

Algorithm 2: Exhaustive transitions application.

Function $f(W)$
  $W = (V, E)$;
  foreach $adj\_pair(v_i, v_j) \in V$ **do**
    if $compose(v_i, v_j)$ **then** $W' \leftarrow compose(v_i, v_j)(W)$;
  foreach $\{v_b, v_1, v_2\} \in V$ **do**
    if $factorize(v_b, v_1, v_2)$ **then** $W' \leftarrow factorize(v_b, v_1, v_2)(W)$;
  foreach $\{v_b, v\} \in V$ **do**
    if $distribute(v_b, v)$ **then** $W' \leftarrow distribute(v_b, v)(W)$;
  foreach $adj\_pair(v_i, v_j) \in V$ **do**
    if $swap(v_i, v_j)$ **then** $W' \leftarrow swap(v_i, v_j)(W)$;
  return W'

3.2.2 Pruning the search space

The version space is exponentially large and in realistic environments we may need more efficient exploration methods than BES. To improve the search performance, we propose two heuristics that prune the search space. Heuristic H3 moves restrictive

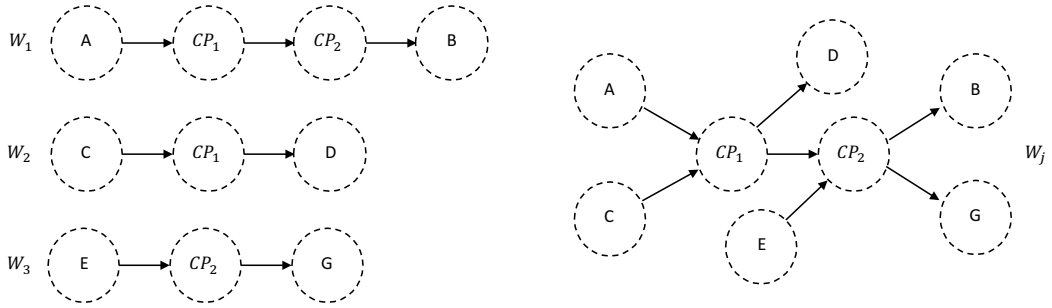


Figure 14: Three workflows and a joint workflow of them

operators towards the root of the workflow in order to reduce early the volume of the propagated data, resulting in workflow versions that have low overall cost. Heuristic H4 groups *blocking* and *non-blocking* operators separately. Since *blocking* operators need the total of a dataset to start processing, whereas *non-blocking* do not, a random sequence of a *blocking* and a *non-blocking* operators may result in delayed overall processing time, because of necessary accumulation of intermediate outputs. By grouping *blocking* and *non-blocking* operators separately we to eliminate such points of necessary accumulation of intermediate data.

- **H**3: Move restrictive operators to the root of the workflow, e.g. change *extract* $\rightarrow$ *function* $\rightarrow$ *filter* to *extract* $\rightarrow$ *filter* $\rightarrow$ *function*, if possible.
- **H**4: Group non-blocking operators together and separately from blocking operators, e.g., change *filter* $\rightarrow$ *sort* $\rightarrow$ *function* $\rightarrow$ *group* to *filter* $\rightarrow$ *function* $\rightarrow$ *sort* $\rightarrow$ *group*.

Based on the heuristics H3 and H4 we present the heuristic algorithm HS.

4 Multi-workflow optimization

Our technique of multi-workflow optimization is based on the joint execution of the common parts of workflows. Specifically, a set of workflows is combined to one joint workflow, so that one or more common subgraphs in these workflows, appear only once in the joint workflow and, therefore, are executed only once. The technique consists of four steps: (1) for each workflow generate all possible equivalent workflow versions; (2) detect common tasks and find the common parts in workflow versions; (3) estimate the processing cost of joint executions; (4) choose workflow versions and common parts in them for the joint execution.

4.1 Creating the joint workflow

A set of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ may be combined in a joint workflow denoted as $W_j = W_1 \circ W_2 \circ \cdots \circ W_{m-1} \circ W_m$. This creation is based on finding common parts

Algorithm 3: ES.

Input: An original workflow $W_o = (V, E)$
and a list of compose constraints $compose\_cons$
Result: A version W_{min} having the minimal cost and versions search space

begin
  $W_{min} = W_o$;
  $open \leftarrow W_o$;
  $close = \varnothing$;
  apply all $compose$s according to $compose\_cons$;
  $H \leftarrow Find\_Homologous\_Tasks(W_o)$;
  $D \leftarrow Find\_Distributable\_Tasks(W_o)$;
  $LPs \leftarrow Find\_Linear\_Paths(W_o)$;
  foreach $lp \in LPs$ **do**
    foreach $adj\_pair(v_i, v_j) \in lp$ **do**
      if $swap(v_i, v_j)$ **then**
        $W' \leftarrow swap(v_i, v_j)(W_o)$;
        if $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;
  foreach $pair(v_i, v_j) \in H$ **do**
    if $factorize(v_b, v_i, v_j)$ **then**
      $W' \leftarrow factorize(v_b, v_i, v_j)(W_o)$;
      if $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;
      $open \leftarrow W'$;
  foreach $W \in open$ **do**
    foreach $v \in D$ **do**
      if $distribute(v_b, v)$ **then**
        $W' \leftarrow distribute(v_b, v)(W)$;
        if $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;
        $open \leftarrow W'$;
  while $open <> \varnothing$ **do**
    $W \leftarrow open$;
    $LP \leftarrow Find\_Linear\_Paths(W)$;
    foreach $lp \in LPs$ **do**
      foreach $adj\_pair(v_i, v_j) \in lp$ **do**
        if $swap(v_i, v_j)$ **then**
          $W' \leftarrow swap(v_i, v_j)(W)$;
          if $W' \notin open$ **and** $W' \notin close$ **then**
            if $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;
            $open \leftarrow W'$;
    $close \leftarrow W$;
  return $(\{min : W_{min}\}, \{versions : close\})$

in the workflows and using them as joint subgraphs between the rest of the workflow graphs. Figure 14 depicts three workflows W_1, W_2, W_3 and a joint workflow of them, W_j. CP_1 and CP_2 represent common parts of W_1, W_2 and W_1, W_3, respectively, and $A..G$ are the remaining parts of workflows.

4.2 Finding common parts

A common part consists of common tasks. Two common tasks consist of the same operators, inputs and outputs. We detect common tasks by comparing properties of

Algorithm 4: HS.

Input: An original workflow $W_o = (V, E)$
and a list of compose constraints $compose\_cons$
Result: A version W_{min} having the minimal cost

begin
  $W_{min} = W_o$;
  $open \leftarrow W_o$;
  $close = \varnothing$;
  apply all $compose$s according to $compose\_cons$;
  $H \leftarrow Find\_Homologous\_Tasks(W_o)$;
  $D \leftarrow Find\_Distributable\_Tasks(W_o)$;
  $LP \leftarrow Find\_Linear\_Paths(W_o)$;
  foreach $lp_k \in LP$ **do**
    foreach $adj\_pair(v_i, v_j) \in lp_k$ **do**
      if $swap(v_i, v_j)$ *and* $H4(v_i, v_j)$ **then**
        $W' \leftarrow swap(v_i, v_j)(W_o)$;
        if $H5(W', W_{min})$ *and* $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;;
  foreach $pair(v_i, v_j) \in H$ **do**
    if $factorize(v_b, v_i, v_j)$ *and* $H4(v_b, v_i, v_j)$ **then**
      $W' \leftarrow factorize(v_b, v_i, v_j)(W_o)$;
      if $H5(W', W_{min})$ *and* $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;;
      $open \leftarrow W'$;
  foreach $W \in open$ **do**
    foreach $v \in D$ **do**
      if $distribute(v_b, v)$ *and* $H4(v_b, v)$ **then**
        $W' \leftarrow distribute(v_b, v)(W)$;
        if $H5(W', W_{min})$ *and* $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;;
        $open \leftarrow W'$;
  while $open <> \varnothing$ **do**
    $W \leftarrow open$;
    $LP \leftarrow Find\_Linear\_Paths(W)$;
    foreach $lp_k \in LP$ **do**
      foreach $adj\_pair(v_i, v_j) \in lp_k$ **do**
        if $swap(v_i, v_j)$ *and* $H4(v_i, v_j)$ **then**
          $W' \leftarrow swap(v_i, v_j)(W)$;
          if $W' \notin open$ **and** $W' \notin close$ **then**
            if $H5(W', W_{min})$ *and* $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;;
            $open \leftarrow W'$;
    $close \leftarrow W$;
  return W_{min}

metadata of tasks, such as input and output data schemas, parameters of operators etc.

After detecting common tasks, we look for subgraphs consisting only of common tasks and compare their structures. If such subgraphs are identical, then they constitute a *common part*. Formally, the latter is defined as follows:

Definition 1 A *common part* $CP(W_1, \ldots, W_m)$ of a set of workflows $\{W_1, \ldots, W_m\}$ is a subgraph S, so that S is part of every one of the workflows, i.e. $S \in W_1 \wedge \cdots \wedge S \in W_m$, and operators of corresponding vertices in a subgraph S of every workflow are identical.

Figure 15: Independently executable and not independently executable subgraphs

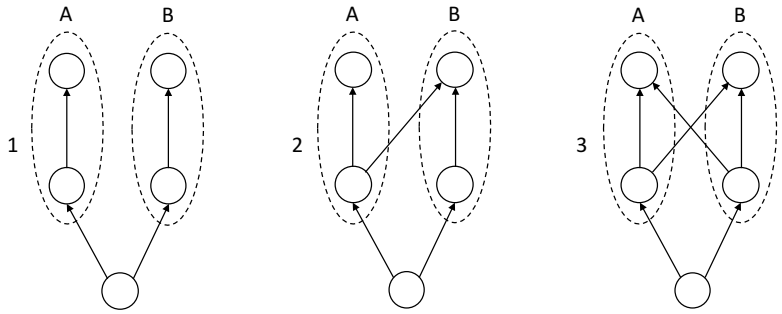


Figure 16: Mutual arrangement of subgraphs A and B

4.2.1 Evaluation of a common part

After finding a common part, we determine if it can be used for the creation of the joint workflow. We do this based on the concepts of *execution state* and *independently executable subgraph*.

An *execution state* XS of a workflow W is a state for which some of the vertices are assumed to have been executed and no vertices are executing. Further, $XS = \{XS^e, XS^{ne}\}$, where XS^e and XS^e are sets of executed and not executed vertices, respectively, and we denote as $\mathcal{XS}_W$ as the set of all execution states of a workflow W.

An *independently executable subgraph* $S \in W$ with respect to some execution state $XS_W \in \mathcal{XS}_W$, is a subgraph that can be executed without executing any vertex in $W \setminus (XS^e_W \cup S)$. There can be several execution states with respect to which S is an independently executable subgraph. Let us denote as $XS_W(S)$ the execution state with the minimal number of executed vertices with respect to which S is an independently executable subgraph.

Figure 15 depicts two workflows W_1 and W_2. In W_1, subgraph A is independently executable with respect to the execution state, the executed vertices of which are colored in blue. In W_2, subgraph A is not independently executable with respect to any execution state, because vertex 4 cannot be executed before vertex 2, and vertex 2 cannot be executed before vertex 3, so vertex 2 has to be executed between vertices 3 and 4.

The creation of a joint workflow W_j of a set of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ that have one common part CP, is possible if CP is independently executable for some execution state for every $W \in \mathcal{W}$.

4.2.2 Evaluation of a set of common parts

A set of workflows to be combined may contain not one, but several common parts. There can be cases for which not all of the common parts can be used for the creation of the joint workflow. To evaluate if a set of common parts $\mathcal{CP}$ can be used in combination for the creation of the joint workflow, we check the *mutual arrangement* of common parts in this set in pairs $CP_i, CP_j \in \mathcal{CP}$.

A *vertex* v *is reachable* from another vertex u if there is a directed path that starts from u and ends at v. A *subgraph* S *depends* on vertex v if there exists a vertex u in the subgraph and u reachable from v. The possible *mutual arrangement* of the subgraphs corresponding to two common parts CP_i and CP_j is one of the following (Figure 16):

1. Independent, if there does not exist a pair of vertices $\{v_i, v_j\}$, $v_i \in CP_i, v_j \in CP_j$ for which CP_i depends on v_j or CP_j depends on v_i.

2. CP_i depends on CP_j, if there is a vertex $v \in CP_j$ and CP_i depends on v, but there is not a vertex in CP_i so that CP_j depends on it.

3. CP_i and CP_j are cross-dependent if there are vertices $v_i \in CP_i$, $v_j \in CP_j$ and CP_j depends on v_i and CP_i depends on v_j.

Depending on their mutual arrangement in the set of workflows, a pair of common parts can be selected for the construction of the joint workflow or not: If the common parts are mutually arranged as (1) in all workflows, both can be selected; if they are mutually arranged as in (3), even in one workflow, they cannot be both selected. If they are mutually arranged as in (2) in some of the workflows, they can be both selected if they have the same dependency in all these workflows. As an example, Figure 17 depicts workflows W_1, W_2 with common parts A, B. In W_1, A depends on B, and in W_2 B depends on A. Only one of A and B can selected for the construction of the joint workflow. Hence, in some cases, we are forced to select only some of the common parts. We do this based on the estimation of processing cost of different choices for the construction of the joint workflow.

4.3 Estimation of processing costs

The processing cost C_T of a task T, is given by IReS [16]. The latter estimates the performance and cost of operators by actually running the operator in representative configuration combinations. Using these measurements, IReS trains surrogate estimator models that can be used to approximate its performance for non-tested configurations. The processing cost of a workflow W, C_W, is the sum of the cost of its tasks: $C_W = \sum_{i=1}^{n} C_{T_i}$.

Let us consider a pair of workflows $\{W_1, W_2\}$ with a common part CP and execution states $XS_{W_1}(CP)$ and $XS_{W_2}(CP)$, respectively. The cost for creating the joint workflow $W_j = W_1 \circ W_2$ is the sum of the cost of execution states $C(XS^e_{W_1}(CP))$ and $C(XS^e_{W_2}(CP))$, the cost of the common part $C(CP)$, the costs of the rest of workflows

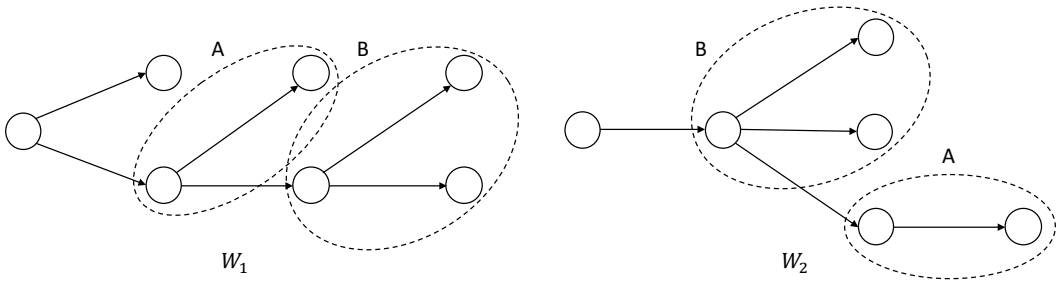


Figure 17: Cross-dependence of common-parts A and B in workflows W_1 and W_2

$C(W_1 \setminus (CP \cup XS^e_{W_1}(CP)))$, $C(W_2 \setminus (CP \cup XS^e_{W_2}(CP)))$, and a synchronization cost $C(sync)$, which captures the cost for creating the joint workflow:

$$\begin{aligned} C(W_1 \circ W_2) = C(XS^e_{W_1}(CP)) + C(XS^e_{W_2}(CP)) + C(CP) + C(sync) + \\ + C(W_1 \setminus (CP \cup XS^e_{W_1}(CP))) + C(W_2 \setminus (CP \cup XS^e_{W_2}(CP))) = \\ = C(W_1) + C(W_2) - C(CP) + C(sync) \end{aligned}$$

The processing cost of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ with common parts $\{CP_1, \ldots, CP_n\}$ is:

$$C(W_1 \circ \cdots \circ W_m) = \sum_{i=1}^{m} C(W_i) - \sum_{i=1}^{n}((n_i - 1)C(CP_i) - C(sync_i))$$

where n_i is the number of occurrences of common part CP_i in $\mathcal{W}$. After estimating the processing costs of all workflow versions and common parts, exhaustive search chooses common parts and workflows with the lowest cost.

4.4 Combining by a common part

Figure 18: Combining of W_1 and W_2 by a common part CP, located at the beginning of workflows

Let us consider two possible placements of a common part:

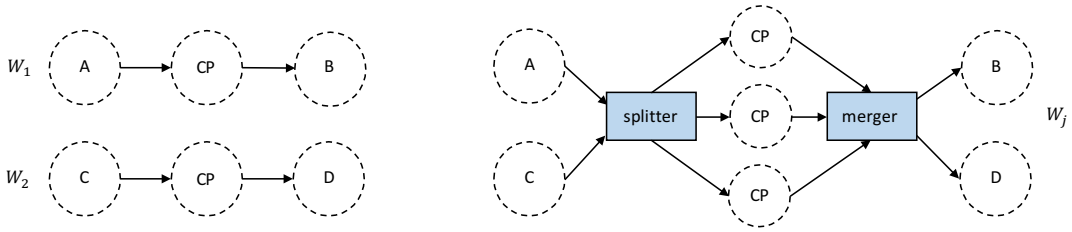


Figure 19: Combining of W_1 and W_2 by a common part CP, located in the middle of workflows

1. A common part CP is located at the beginning of workflows W_1 and W_2, this is equivalent to that sets of executed vertices $XS^e_{W_1}(CP)$ and $XS^e_{W_2}(CP)$ are empty. In this case a common part processes the same datasets in both workflows. Figure 18 shows how workflows are combined by a common part, located at the beginning. Formally, a joint workflow $W_j = (V_j, E_j)$ is following:

   $V_j \leftarrow V_{cp} \wedge (V_1 \setminus V_{cp}) \wedge (V_2 \setminus V_{cp})$

   $E_j \leftarrow E_{cp} \wedge (E_1 \setminus E_{cp}) \wedge (E_2 \setminus E_{cp})$

2. A common part CP is located in the middle of workflows W_1 and W_2, this is equivalent to that sets of executed vertices $XS^e_{W_1}(CP)$ and $XS^e_{W_2}(CP)$ are not empty. In this case, the datasets input to the common part by the different workflows are not equal. However, if the common part consists only of non-blocking operators, then we can optimize as follows: Two datasets D_1 and D_2 (or two sets of datasets) can be split into three datasets $D_1 \cap D_2$, $D_1 \setminus D_2$ and $D_2 \setminus D_1$. Further, these three datasets are processed in three parallel branches CP, which can be executed in parallel. Then, processed data is merged into two datasets (inverse transformation of splitting). Figure 19 displays an example of this split-merge combining. Formally, a joint workflow $W_j = (V_j, E_j)$ is following:

   $V_j \leftarrow XS^e_{W_1}(CP) \wedge XS^e_{W_2}(CP) \wedge v_{split} \wedge 3 \times V_{cp} \wedge v_{merge} \wedge (V_1 \setminus (V_{cp} \wedge XS^e_{W_1}(CP))) \wedge (V_2 \setminus (V_{cp} \wedge XS^e_{W_2}(CP)))$

   $E_j \leftarrow E_{XS^e_{W_1}(CP)} \wedge E_{XS^e_{W_2}(CP)} \wedge \{XS^e_{W_1}(CP), v_{split}\} \wedge \{XS^e_{W_2}(CP), v_{split}\} \wedge 3 \times \{v_{split}, V_{cp}\} \wedge 3 \times E_{cp} \wedge 3 \times \{V_{cp}, v_{merge}\} \wedge \{v_{merge}, V_1 \setminus (V_{cp} \wedge XS^e_{W_1}(CP))\} \wedge \{v_{merge}, V_2 \setminus (V_{cp} \wedge XS^e_{W_2}(CP))\} \wedge (E_1 \setminus (E_{cp} \wedge E_{XS^e_{W_1}(CP)})) \wedge (E_2 \setminus (E_{cp} \wedge E_{XS^e_{W_2}(CP)}))$

   In these equations v_{split} and v_{merge} are vertices, that are placed on the incoming and outgoing edges of common part, respectively. The split-merge combining can be performed if operators *splitter* and *merger* have an appropriate implementations for the datatypes of processed data. Also note, that *merger* is cheap operator and *splitter* is quite expensive. Therefore, the efficiency of the split-merge combining depends on the cost of the common part and the size of common data ($D_1 \cap D_2$). MWO takes into account the cost estimation of the common part, but it doesn't estimate the size of common data. In this regard, MWO can be improved so that a decision on the application of split-merge combining will be

taken dynamically after estimation of a size of the common data: if the common data is big, then apply split-merge combining, if not, then leave as it is.

The pseudo-code of combining is shown in Code List 6, procedures *combine* and *scombine* there. The procedure *combine* takes as input two workflows and their common part. Then, it detects the location of a common part and combines by this common part according to the rules described above. The procedure *scombine* is similar to *combine*, but combines by a common part in one workflow.

4.5 Multi-workflow optimization algorithm

Code List 5: Algorithm of Multi-Workflow Optimization.

```
1  Algorithm MWO(W_1, W_2)
       Input: Two workflows W_1, W_2
       Result: A joint version W_12^min or two versions W_1^min, W_2^min having the minimal cost
2      begin
3          W_1^min = ES(W_1)[min];
4          W_2^min = ES(W_2)[min];
5          versions_1 = ES(W_1)[versions];
6          versions_2 = ES(W_2)[versions];
7          CTs ← Find_Common_Tasks(W_1, W_2);
8          foreach v_1 ∈ versions_1 do
9              foreach v_2 ∈ versions_2 do
10                 CPs ← Find_Common_Parts(v_1, v_2, CTs);
11                 foreach cp ∈ CPs do
12                     W_j ← combine(v_1, v_2, cp);
13                     if (C(W_j) < C(W_min)) then W_min = W_j;
14                     innersearch(W_j, W_min, CPs \ cp);
15         if (C(W_min) ≤ C(W_1^min) + C(W_2^min)) then
16             return W_min
17         else
18             return {W_1^min, W_2^min}
```

We create the Multi-workflow Optimization algorithm (MWO) that explores the space of joint versions of two workflows exhaustively. First, MWO generates a set of equivalent versions of the workflows W_1 and W_2 ($versions_1$ and $versions_2$, respectively) and finds optimal versions W_1^{min} and W_2^{min} from these spaces (lines 3–6 in code list 5). Second, MWO finds common tasks of W_1 and W_2 (line 7). Third, for every pair of versions MWO finds all common parts CPs (line 10). Fourth, MWO produces joint workflows by a single common part exhaustively (line 12). Then, recursively in procedure *innersearch* MWO combines by the rest common parts and finds the optimal joint version W_{12}^{min} with the minimal cost. Finally, the algorithm returns W_{12}^{min} or W_1^{min} and W_2^{min} depending on which execution cost is lower.

Clearly, the search space of joint versions is finite and the algorithm terminates after having generated all possible versions and returning the optimal one.

Code List 6: Procedures for Multi-Workflow Optimization.

Procedure `innersearch`(W, W_{min}, CPs)
  Input: Workflows W, W_{min} and common parts CPs of W
  Result: W_{min} is replaced with a version having the minimal cost
  begin
    foreach $cp \in CPs$ **do**
      $W' \leftarrow$ `scombine`(W, cp);
      if $(C(W') < C(W_{min}))$ **then** $W_{min} = W'$;
      `innersearch`$(W', W_{min}, CPs \setminus cp)$;

Procedure `combine`(W_1, W_2, cp)
  Input: Two workflows W_1, W_2 and their common part cp
  Result: A joint workflow W_j by cp
  begin
    $XS_1 = XS^e_{W_1}(CP)$;
    $XS_2 = XS^e_{W_2}(CP)$;
    $(V_1, E_1) \leftarrow W_1$;
    $(V_2, E_2) \leftarrow W_2$;
    $(V_{XS_1}, E_{XS_1}) \leftarrow XS_1$;
    $(V_{XS_2}, E_{XS_2}) \leftarrow XS_2$;
    $(V_{cp}, E_{cp}) \leftarrow cp$;
    if $XS_1 == \varnothing$ **and** $XS_2 == \varnothing$ **then**
      $V_j \leftarrow V_{cp} \wedge (V_1 \setminus V_{cp}) \wedge (V_2 \setminus V_{cp})$;
      $E_j \leftarrow E_{cp} \wedge (E_1 \setminus E_{cp}) \wedge (E_2 \setminus E_{cp})$;
    else
      $V_j \leftarrow V_{XS_1} \wedge V_{XS_2} \wedge v_{split} \wedge 3 \times V_{cp} \wedge v_{merge} \wedge (V_1 \setminus (V_{cp} \wedge V_{XS_1})) \wedge (V_2 \setminus (V_{cp} \wedge V_{XS_2}))$;
      $E_j \leftarrow E_{XS_1} \wedge E_{XS_2} \wedge \{V_{XS_1}, v_{split}\} \wedge \{V_{XS_2}, v_{split}\} \wedge 3 \times \{v_{split}, V_{cp}\} \wedge 3 \times E_{cp} \wedge 3 \times \{V_{cp}, v_{merge}\} \wedge \{v_{merge}, V_1 \setminus (V_{cp} \wedge V_{XS_1})\} \wedge \{v_{merge}, V_2 \setminus (V_{cp} \wedge V_{XS_2})\} \wedge (E_1 \setminus (E_{cp} \wedge E_{XS_1})) \wedge (E_2 \setminus (E_{cp} \wedge E_{XS_2}))$;
    $W_j \leftarrow (V_j, E_j)$;
    return W_j

Procedure `scombine`(W, cp)
  Input: A workflow W and its common part cp
  Result: A joint workflow W_j by cp
  begin
    $\{XS_1, XS_2\} = XS^e_W(CP)$;
    $(V, E) \leftarrow W$;
    $(V_{XS_1}, E_{XS_1}) \leftarrow XS_1$;
    $(V_{XS_2}, E_{XS_2}) \leftarrow XS_2$;
    $(V_{cp}, E_{cp}) \leftarrow cp$;
    if $XS_1 \cap cp == \varnothing$ **and** $XS_2 \cap cp == \varnothing$ **then**
      $V_j \leftarrow V_{XS_1 \cap XS_2} \wedge V_{XS_2 \setminus XS_1} \wedge V_{XS_1 \setminus XS_2} \wedge v_{split} \wedge 3 \times V_{cp} \wedge v_{merge} \wedge (V \setminus (V_{cp} \wedge V_{XS_1} \wedge V_{XS_2}))$;
      $E_j \leftarrow E_{XS_1 \cap XS_2} \wedge E_{XS_2 \setminus XS_1} \wedge E_{XS_1 \setminus XS_2} \wedge \{V_{XS_1 \cup XS_2}, v_{split}\} \wedge 3 \times \{v_{split}, V_{cp}\} \wedge 3 \times E_{cp} \wedge 3 \times \{V_{cp}, v_{merge}\} \wedge \{v_{merge}, V \setminus (V_{cp} \wedge V_{XS_1 \cup XS_2})\} \wedge (E \setminus (E_{cp} \wedge E_{XS_1 \cup XS_2}))$;
    $W_j \leftarrow (V_j, E_j)$;
    return W_j

5 Benchmarking Workflows

In this section, first, we present several high-level graph patterns that the structure of a workflow may follow, and, second, we describe an algorithm of "filling" workflow structures that follow such patterns with operators.

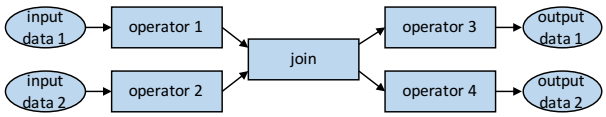


Figure 20: Butterfly configuration

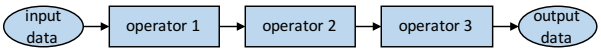


Figure 21: Line configuration

5.1 Generating a workflow structure

Butterflies. We introduce a broad category of workflows, called Butterflies. A butterfly (Figure 20) is a workflow that consists of three distinct components: (a) the left wing, (b) the body, and (c) the right wing. The left and right wings are two non-overlapping subgraphs which are connected to the body of the butterfly. Specifically:

- The left wing of the butterfly includes two (can be the same) or more data sources and operators. Typically, this part of the butterfly performs the extraction and transformation part of the workflow and loads the processed data to the body of the butterfly.
- The body of the butterfly is an operator that takes as input the data produced by the left wing. It is branching operator that merges parallel data flows through some variant of a join (e.g., a relational join, diff, merge) or a union (e.g., the overall sorting of two independently sorted datasets).
- The right wing receives the data output by the body and uses them to support reporting and analysis activity. The right wing consists of operators that materialize views and create reports.

Lines. Lines (see Figure 21) are sequences of an input data source, a series of operators with one input and one output and a target data store and operators. Lines form single data flows.

Forks and Trees. These workflows are structured around a branching operator that either unifies the data flows of multiple lines into one, or distributes the data flow of one line into multiple ones. In the first case the pattern of the structure is called a Tree (Figure 23), and in the second it is called a Fork (Figure 22). In a Tree the branching operator is usually some type of a join. In a Fork the branching operator can be any operator the output data of which is input to multiple operators.

The above patterns form a classification that assists the creation of a benchmark of synthetic workflows. For example, to create a memory intensive workflow, we consider using Tree or Fork patterns, which include joins and a significant number of sorting or aggregating operators. If we want to study pipelining as well, we may consider extending these workflows with line workflows (we need to tune the distribution of

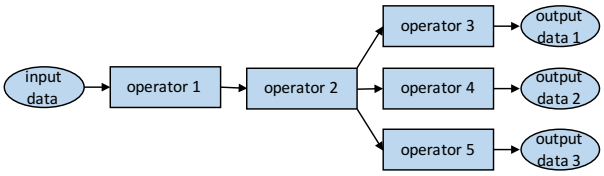


Figure 22: Fork configuration

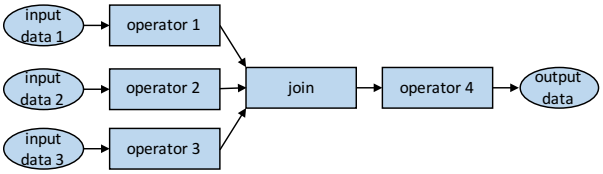


Figure 23: Tree configuration

blocking and non-blocking operations in these workflows too, how it is done described further).

Furthermore, the above classification can be employed to decompose existing complex workflows into sets of primitive structures in order to study their execution behavior in a systematic manner. This decomposition can be used for optimization purposes too. We study the behavior of the workflow patterns in isolation, and then, we can use our findings for optimizing and tuning the whole workflow: the performance of a workflow with complex structure can be derived from the performance of the component primitive ones.

5.2 Generating workflow queries

After creating a workflow graph from the patterns described above we need to fill the vertices with analytic queries and choose input data sources. For this purpose we use TPC-DS [27].

TPC-DS. TPC-DS is a Decision Support (DS) benchmark being developed by the TPC. This benchmark models the decision support system of a retail product supplier, including queries and data maintenance. The relational schema of this benchmark is more complex than the schema presented in TPC-H. There are three sales channels: store, catalog and the web. There are two fact tables in each channel, sales and returns, and a total of seven fact tables. In this dataset, the row counts for tables scale differently per table category: specifically, in fact tables the row count grows linearly, while in dimension tables grows sub-linearly. This benchmark also provides refreshment scenarios for the data warehouse.

In our benchmark of workflows we use two tables: *web_sales* and *customers*. Therefore, we prepare query templates which produce operators, which correspond to the types of operators we define in Section 2.3. One of such query template, shown in Listing 1, produces restrictive non-blocking operators. Templates produce opera-

tors of four combinations of operator types: blocking and restrictive, non-blocking and restrictive, blocking and non-restrictive, non-blocking and non-restrictive. Thus, we can fill the workflow structure with the desired composition of operators, regulating the percentage of blocking, non-blocking and restrictive operators.

Algorithm 1: Example of Query Template

```
define YEAR=random(1996,2001,uniform);
define RANGE=random(1,4,uniform)
define LISTPRICE=ulist(random(0,190,uniform),2);
define _LIMIT=random(1,rowcount(web_sales),uniform);

[_LIMITA] select [_LIMITB] *
  from web_sales
where YEAR(ws_sold_date) between [YEAR]-[RANGE]
        and [YEAR]-[RANGE]
  and ws_list_price between [LISTPRICE.1]
        and [LISTPRICE.2]
order by ws_order_number
[_LIMITC];
```

6 Experiments

In this section we present a thorough experimental study on real and synthetic workflows and data.

6.1 Experimental Setup

Methodology Our experimental study measures the performance of original workflows and their optimizations, the size of the workflow versions space produced by the proposed optimization algorithms ES, HS and MWO for multi-workflow systems, and is divided in two parts: the first part studies real workflows and their execution on real datasets, and the second studies synthetic workflows generated based on the benchmarking method described in Chapter 5. The second part of the study examines the performance behavior of workflows and the relation of performance to the version space size, with respect to the size and the structure of the workflows, the length of paths (single data flows), connectivity (existence of branching operators), and parallelization opportunities (multiple parallel data flows). Also, the second part of the study examines the performance behavior of workflows and the relation of performance to the version space size, with respect to the distribution of operators, as well as to whether there are opportunities for composition/decomposition of operators and execution of alternative implementations of operators on different engine types.

Datasets Our workloads involves three sets of data sources, namely (1) *CDR* and *voronoi*; (2) *tweets*, *sales*, *products*, *region*, and *campaign*; (3) *web_sales* and *customers*.

Figure 24: Telecommunication data structures
CDR (id, user_id, ts, aid)
voronoi (aid, rid)

(1) CDR is an anonymised Call Detail Records data for the City of Rome, from 01 Jun 2014 until 30 June 2015, provided by WIND and stored in CSV format. The whole dataset has size in the order of hundreds of Terabytes. *Voronoi* is a small table that stores links of regions and antennas. Data structures of *CDR* and *voronoi* are displayed in Table 24

Figure 25: Sales data structures
products (product_id, product_name)
tweets (id, reg_id, ts, text)
sales (id, product_id, reg_id, ts)
campaign (id, product_id, reg_id, ts)

Table 1: Data sizes of tweets and sales data stores

	10k rows	100k	1M rows	10M rows
Tweets	1.31MB	12.44MB	124.39MB	1.23GB
Sales	0.33MB	2.72MB	26.84MB	262.1MB

(2) Data stores *tweets* and *sales* are large fact tables stored in PostgreSQL. *Tweets* keeps unstructured information. *Sales* records lineitems for a purchase order. The other two data stores *products* and *campaign* are slowly changing dimensional data. We consider four different data sizes for *tweets* and *sales* comprising 10k, 100k, 1M , 10M rows and with a row size of 26 bytes for *sales* and an average of 124 bytes for *tweets*. The other tables are relatively small. Tables 25 and 1 represent data structures and data sizes, respectively.

(3) *Web-sales* and *customers* are data sources that are generated using TPC-DS and are stored in PostgreSQL. *Web-sales* has around 720k rows and 34 columns. *Customers* has 100k rows and 18 columns. Data structures are shown in Table 26.

Experimental Platform We measured performance of workflow versions on a cluster that consists of 4 server-grade physical nodes. Each one of those is equipped with a 3rd generation i5 CPU (@ 2.90 GHz) and 16GB of physical memory and an array of 2x4TB SSHD on RAID-0. The operating system is Debian 6 (squeeze) Linux. For the time being, the three software platforms running on this setup are Hadoop, Spark, Postgres and Weka. The distribution of Hadoop is CDH 4.6.0 (a popular bundling of Hadoop by Cloudera) which uses Hadoop version 2.0.0 over MapReduce scheduler. The version of Spark is 1.4.1, running in standalone mode. The Hadoop and Spark

Figure 26: Benchmark data structures

web_sales (ws_sold_date_sk, ws_sold_time_sk, ws_ship_date_sk, ws_item_sk, ws_bill_customer_sk, ws_bill_cdemo_sk, ws_bill_hdemo_sk, ws_bill_addr_sk, ws_ship_customer_sk, ws_ship_cdemo_sk, ws_ship_hdemo_sk, ws_ship_addr_sk, ws_web_page_sk, ws_web_site_sk, ws_ship_mode_sk, ws_warehouse_sk, ws_promo_sk, ws_order_number, ws_quantity, ws_wholesale_cost, ws_list_price, ws_sales_price, ws_ext_discount_amt, ws_ext_sales_price, ws_ext_wholesale_cost, ws_ext_list_price, ws_ext_tax, ws_coupon_amt, ws_ext_ship_cost, ws_net_paid, ws_net_paid_inc_tax, ws_net_paid_inc_ship, ws_net_paid_inc_ship_tax, ws_net_profit)

customer (c_customer_sk, c_customer_id, c_current_cdemo_sk, c_current_hdemo_sk, c_current_addr_sk, c_first_shipto_date_sk, c_first_sales_date_sk, c_salutation, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_day, c_birth_month, c_birth_year, c_birth_country, c_login, c_email_address, c_last_review_date)

installation on this cluster is configured so that all the machines run as workers and one of them runs the master.

6.2 Description of Experimental Workloads and Results

In this section, we evaluate our exhaustive and heuristic optimization algorithms. First, we demonstrate the whole process of optimization, including operator training on two real-data driven workflows. Then, we evaluate each algorithm on a benchmark of synthetic workloads of workflows.

We execute approximately 1400 workflows and record the execution times and used system resources. We label each workload with the set of characteristics of the workflows it includes.

1. the size of the workflow (i.e., the number of vertices contained in the graph),
2. the structure of the workflow,
3. the percentage of blocking, non-blocking and restrictive operators,
4. the size of input data sources,
5. the workflow selectivity, based on the selectivities of the workflow operators

We examine the optimization algorithms in order to answer the following questions:

1. How fast does the algorithm produce an optimized version of a workflow? This question is covered by experiments 3.1 – 3.3.
2. What is the performance gain of the optimized version with respect to the performance of the original workflow? This is a basic question and it is answered in all experiments 1.1 – 4.1.
3. How large is the search space generated by the algorithm? This question is answered in all experiments 1.1 – 4.1.
4. What is the impact of workflow characteristics (workflow size, structure, percentage of blocking, non-blocking and restrictive operators, input data size)? This question is fully covered by experiments 3.1 – 3.3.
5. Do the algorithms ES and HS produce the same solutions? What causes the divergence of solutions? This is a basic question and is answered in all single-workflow optimization experiments 1.1 – 3.3.
6. How does the algorithm cope with a workflow that contains an operator, for which the cost function is not defined? This question is answered in experiment 1.2.

We use the following workloads to help answer the above questions:

1. WL-1: Real-time analytics on telecommunication data.
2. WL-2: Analysis of a product marketing campaign.
3. WL-3: Benchmarked workflows.
4. WL-4: Similar workflows

6.2.1 Workload 1 - Real-time analytics on telecommunication data.

With the execution of this workload we demonstrate how the optimization algorithms adapt to changes of the cost functions of operators. This workload consists of two workflows of real-time analytics on telecommunication data. Both workflows involve processing of *CDR* and *voronoi* data sources. One of these workflows is described as a motivating example (Sec. 2.1). Another workflow, called 'User Profiling', divides users based on their call behavior into five following categories: "Commuter", "Static" resident, "Occasional", "Dynamic" resident and "Uncategorized". This workflow is depicted in Figure 27 and an optimized version of it is shown in Figure 28.

Experiment 1.1 In the first experiment of this workload we perform execution and we test optimization of both workflows, 'Peak Detection' and 'User Profiling', for several input data sizes: 10k, 100k, 1M, 2M, 5M, 7.5M and 10M rows in *CDR* data source. The execution times of the original and optimized versions of workflows are shown in

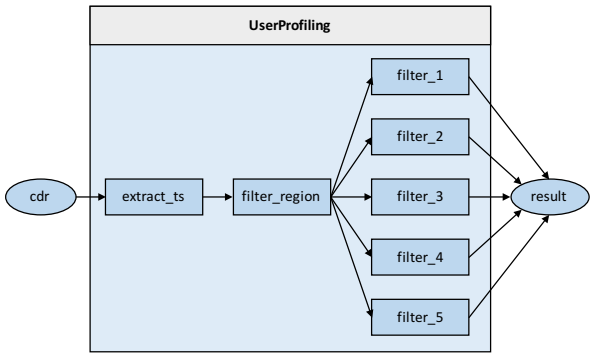


Figure 27: Original 'User Profiling' workflow

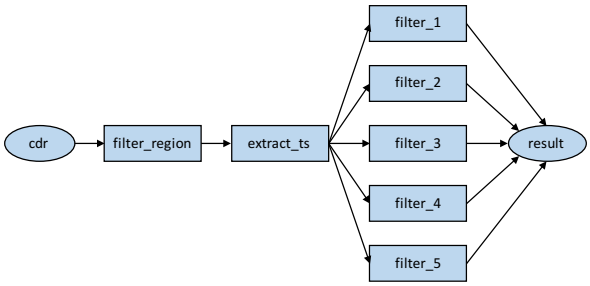


Figure 28: Optimized version of 'User Profiling' workflow

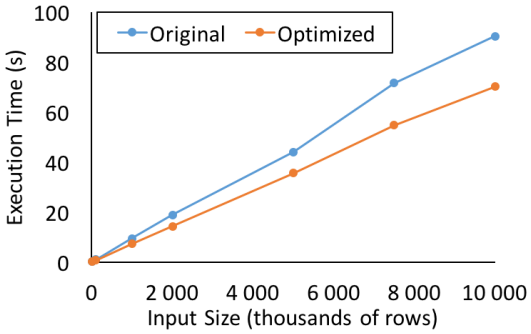


Figure 29: Execution time versus input data size in 'Peak Detection' workflow

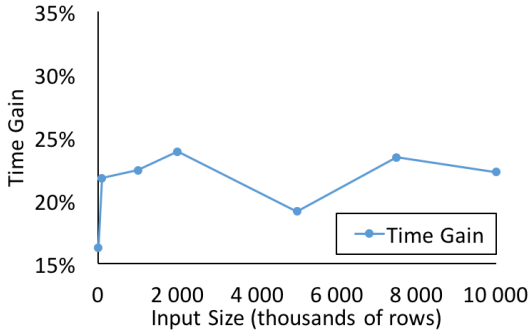


Figure 30: Time gain versus input data size in 'Peak Detection' workflow

Figures 29 and 31. The time gain of 'Peak Detection' workflow is around 20% and that of User Profiling is around 30% (Figures 30 and 32). For both workflows, the versions space size of algorithms is independent of the input data size. For workflow 'User Profiling' ES and HS space sizes are of equal size (4) and for 'Peak Detection' workflow the HS versions space size (15) is smaller than the ES space size (18). An example of pruning by HS is the following: HS doesn't *swap LO_join* with *filter_train* and *filter_test*, because this transition contradicts to a heuristic H4. However, HS produces the same result, as ES. Here is how HS works in this case: first, the pairs of vertices *LO_join* and *filter_region* are composed; next, the composed vertices are *swap*ped with *filter_train* and *filter_test*, this can be done because composed vertices are restrictive operators; and then this composed vertices are distributed over *calc_num* and *swap*ped further to the root of the workflow.

Experiment 1.2 In this experiment, we perform execution of the workflow 'Peak Detection' on 10M rows with two different states: (1) all operators are supplied with cost functions; (2) all operators except *filter_region* have cost functions and *filter_region* doesn't have a cost function. In the first case HS and ES produce identical optimal versions of the workflow. In the second case HS produces the same result (Figure 4) and ES returns the original version as an optimal. This is due to the fact that HS takes

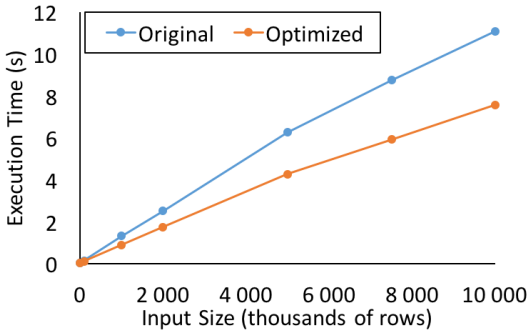


Figure 31: Execution time versus input data size in 'User Profiling' workflow

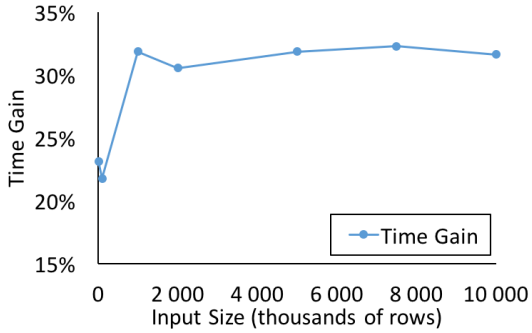


Figure 32: Time gain versus input data size in 'User Profiling' workflow

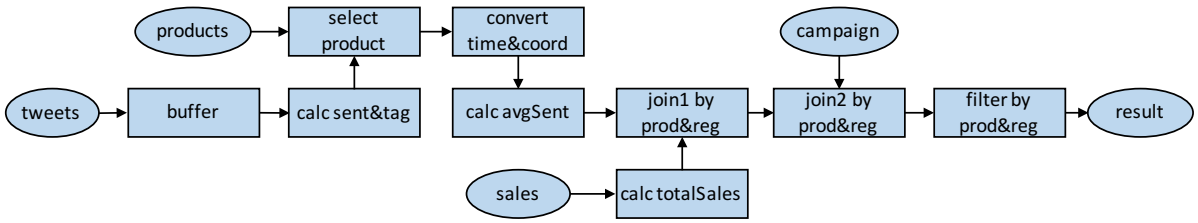


Figure 33: Original workflow of marketing campaign analysis

into account not only cost estimation of a workflow, but also closeness of restrictive operators to the root of a workflow. I.e. if workflow versions have equal cost estimations, HS compares in which one restrictive operators are closer to input datasets.

6.2.2 Workload 2 - Marketing campaign.

In this workload, we demonstrate the impact of different input data sizes to optimization. Figure 33 displays a workflow of an analysis of a product marketing campaign. It combines sales data with sentiments (*join1 by prod&reg*) about that product gleaned from tweets crawled from the Web (*calc sent&tag*). The result consists of total sales (*calc totalSales*) and average sentiment (*calc avgSent*) for each day of the campaign. Campaigns promote a specific product and are targeted at non-overlapping, geographical regions. To simplify the presentation, we assume the sentiment analysis of a tweet yields a single metric, i.e., like or dislike the product over a range of -5 to +5.

Figure 34 depicts the result of optimization. The Optimizer makes several workflow graph manipulations: (a) the vertex *filter by prod&reg* is distributed and pushed closer to input data sources (sales, campaign, tweets); (b) The vertex *convert time&coord* is *swap*ped with *select product* and composed with *calc sent&tag*, producing the vertex *calc&conv*.

Experiment 2.1 In this experiment, the workflow 'Marketing Campaign' runs on several input data sizes: 10k, 100k, 1M, 2M, 5M, 7.5M and 10M rows of *tweets* data-

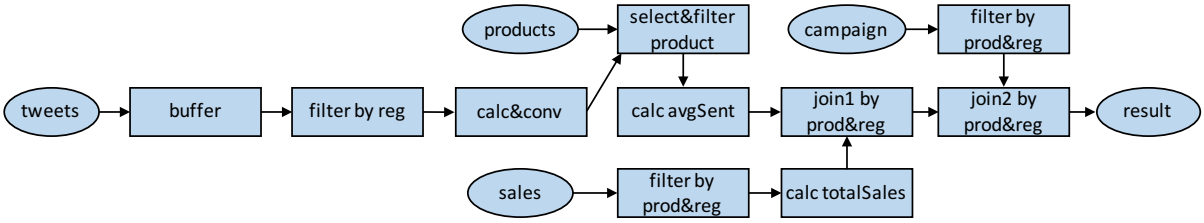


Figure 34: Optimized workflow of marketing campaign analysis

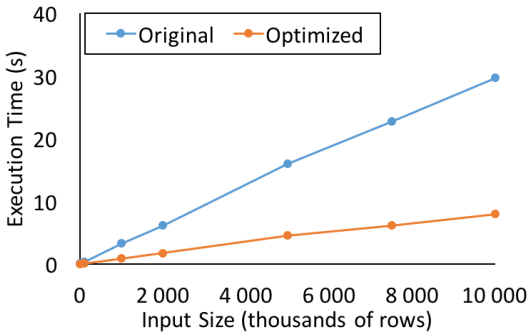


Figure 35: Execution time versus input data size in a workflow of Marketing Campaign analysis

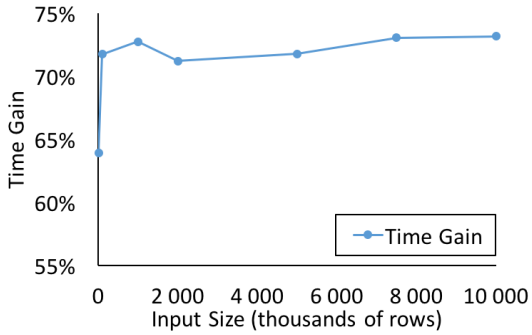


Figure 36: Time gain versus input data size in a workflow of Marketing Campaign analysis

source. In Figures 35 and 36, the effect of optimization is more evident than in previous experiments. This is due to the fact that the highly restrictive operator *filter by prod&reg* in the original workflow is closer to the end, and in optimized version it is moved closer to the input data sources. This results in more than 70% time gain.

6.2.3 Workload 3 - Benchmarking workflows.

Table 2: Benchmark parameters

Parameter	range	constant
Workflow size	10–200	20–50
Workflow structure		
butterfly	10–70%	25%
line	10–70%	25%
fork	10–70%	25%
tree	10–70%	25%
Operators		
blocking	0–100%	25–75%
non-blocking	0–100%	25–75%
restrictive	0–100%	25–75%

This workload comprises synthetic workflows. These are based on several graph patterns and "filled" with queries generated using TPC-DS. A detailed way of constructing them is provided in Section 5. The parameters of workflow generation for this workload are shown in Table 2.

We measure the optimization time, the execution time and we demonstrate the effect of workflow characteristics for each algorithm.

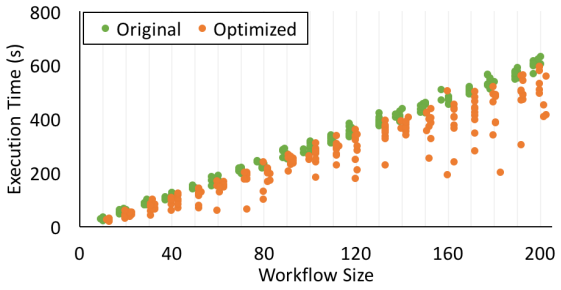


Figure 37: Execution time versus workflow size

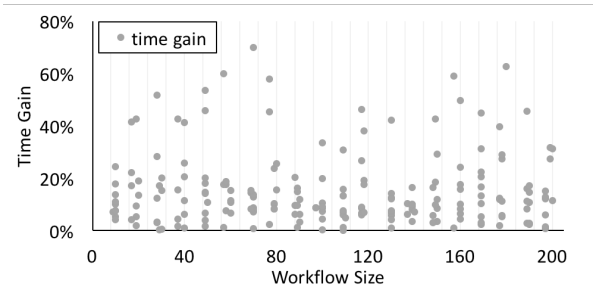


Figure 38: Time gain versus workflow size

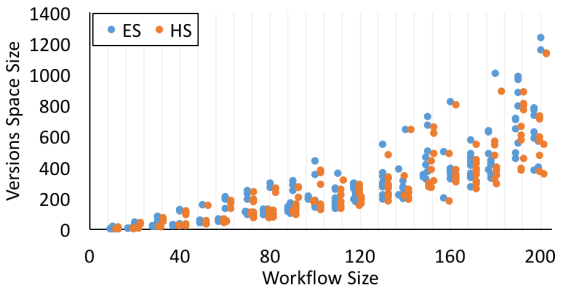


Figure 39: Versions space size versus workflow size

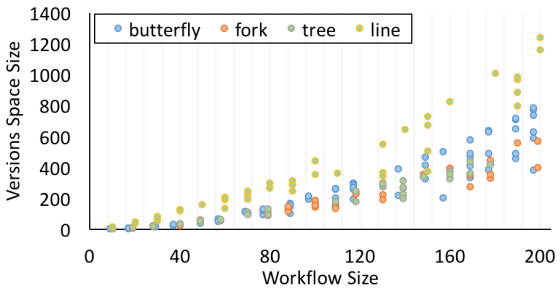


Figure 40: Versions space size versus workflow size broken down by structure type

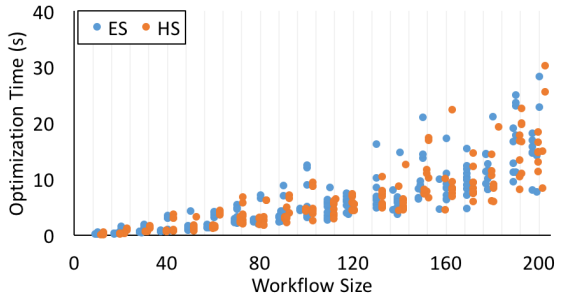


Figure 41: Optimization time versus workflow size

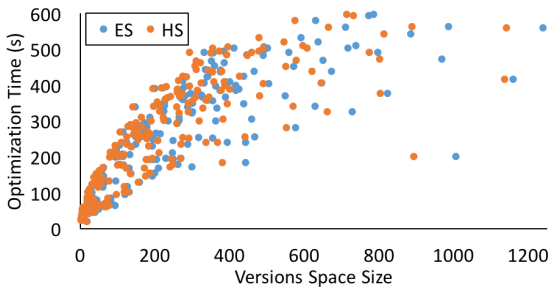


Figure 42: Optimization time versus versions space size

Experiment 3.1: Workflow size effect Figures 37 – 40 display the results of runs of 200 workflows automatically generated for the following configuration: Structure

[butterfly, line, fork, tree] - [25%, 25%, 25%, 25%]; Workflow size - 10–200; Operators [blocking, non-blocking, restrictive] - [25–75%, 25–75%, 25–75%]. In this experiment set we vary workflow size from 10 to 200 with the step of 10 vertices. Figure 37 shows that the execution time is similar to a linear function of the size of the generated workflows and ES execution times are lower than those of the execution of the original workflow. Figure 38 displays the time gain due to optimization. Figure 39 shows the relation of the size of the search space and and the size of workflows. The search space is bigger for bigger workflows, since more vertices provide, usually, more possibilities for workflow manipulations. Figure 40 displays the same data (ES space size versus workflow size) but also broken down by structures: lines have bigger search spaces for the same workflow size than other workflow structures. This is because workflows with long or many lines process in a serial manner data of one input dataset and without the interpolation of branching vertices, which often becomes an obstacle to moving the vertices to other branches in Trees, Forks and Butterflies. Then, Figures 41 – 42 show that an increase of search space size causes an increase of time for choosing an optimal version of a workflow. Figure 42 shows that optimization time increases with a higher rate if the search space size increases from small to medium (0-400), and with a slower rate if the search space increases from medium to large (400 - $>$ 600). The reason is that in the first case, the search space consists of many versions which present with different optimization opportunities, whereas in the second, it comprises also many similar versions which present with the same optimization opportunities, or, equivalently, the same optimization restrictions.

Figure 43: Execution time versus percentage of restrictive operators

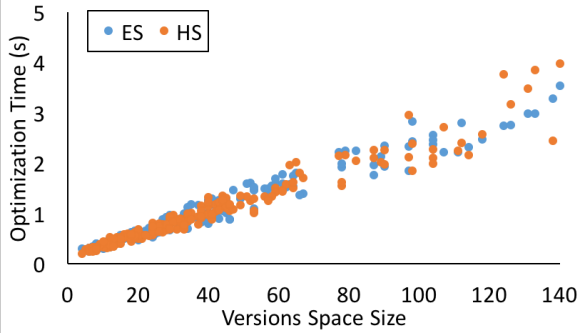


Figure 44: Optimization time versus versions space size

Experiment 3.2: Effect of restrictive operators Figures 43 – 48 display the results of runs of 200 workflows automatically generated for the following configuration: Structure [butterfly, line, fork, tree] - [25%, 25%, 25%, 25%]; Workflow size - 20–50; Operators [blocking, non-blocking, restrictive] - [25–75%, 25–75%, 0–100%]. In this experiment set we vary the percentage of restrictive operators from 0% to 100% with a step of 10%. Figure 47 displays the time gain of optimization. We observe that time gain increases with the increase of the percentage of restrictive operators from 0% up to about 80%; however, as the percentage of restrictive operators increases more and reaches close to 100%, the time gain values decrease. Thus, for workflows with a

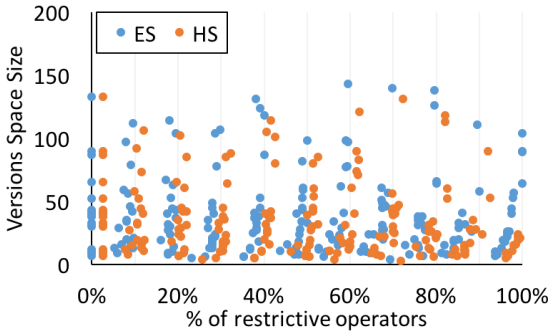


Figure 45: Versions space size versus percentage of restrictive operators

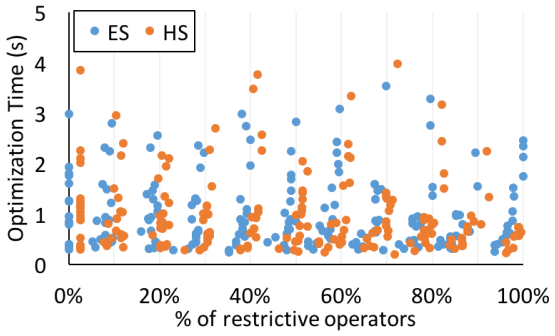


Figure 46: Optimization time versus percentage of restrictive operators

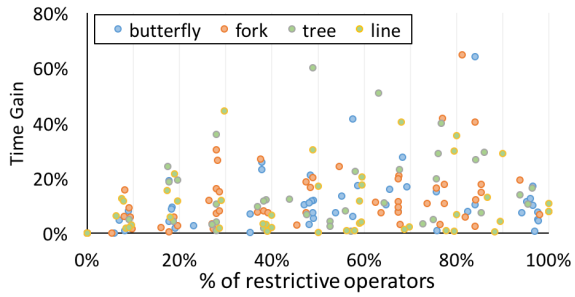


Figure 47: Time gain versus percentage of restrictive operators broken down by structure type

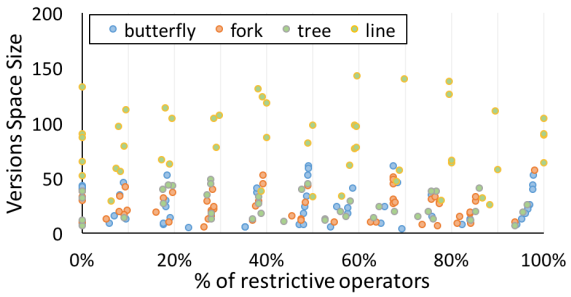


Figure 48: Versions space size versus percentage of restrictive operators broken down by structure type

"medium" number of restrictive operators (30%-80%) we can expect a significant time gain from optimization. Figure 44 shows that an increase of search space size causes an increase of time for choosing an optimal workflow version. This relation is close to linear. Figure 45 shows that there is no evident dependence of search space size from the percentage of restrictive operators in a workflow.

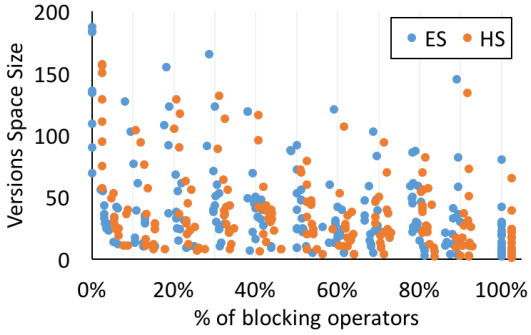


Figure 49: Versions space size versus percentage of blocking operators

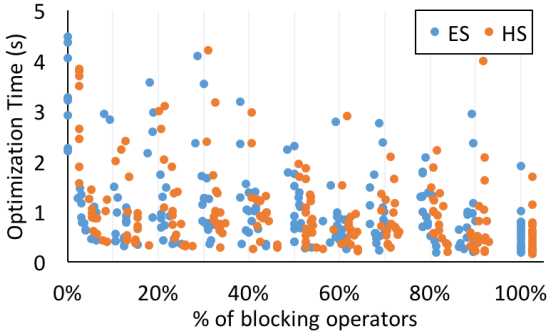


Figure 50: Optimization time versus percentage of blocking operators

Experiment 3.3: Effect of blocking operators Figures 49 – 52 display the results

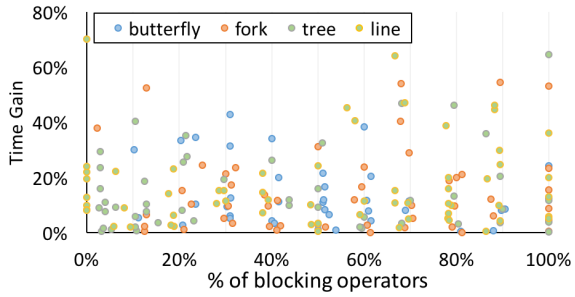


Figure 51: Time gain versus percentage of blocking operators broken down by structure type

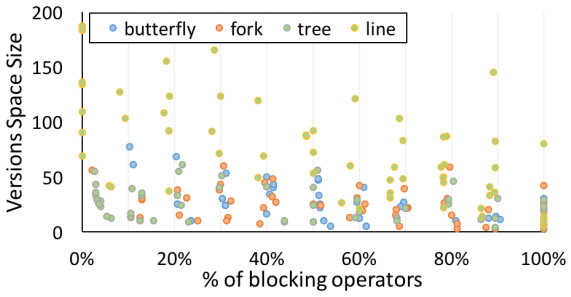


Figure 52: Versions space size versus percentage of blocking operators broken down by structure type

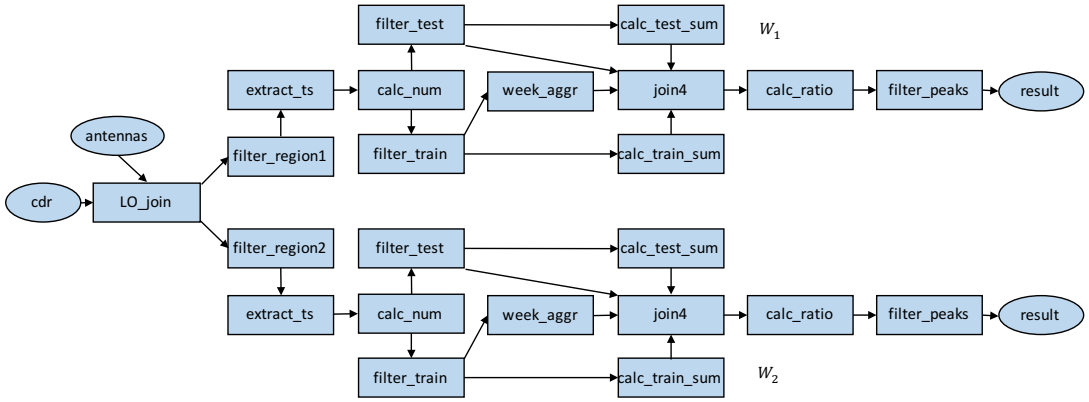


Figure 53: An optimal joint workflow of two 'Peak Detection' workflows in case if total selectivity of *filter_region1* and *filter_region2* is high

of runs of 200 workflows automatically generated for the following configuration: Structure [butterfly, line, fork, tree] - [25%, 25%, 25%, 25%]; Workflow size - 20–50; Operators [blocking, non-blocking, restrictive] - [0–100%, 0–100%, 25–75%]. Figure 49 shows that as the percentage of blocking operators increases the size of the optimization search space decreases and, respectively, the optimization time also decreases (Figure 50). However, Figure 51 shows that there is evident dependence of time gain of optimization algorithm from the structure type.

6.2.4 Workload 4 - Similar workflows.

With this workload we demonstrate how the algorithm of multi-workflow optimization performs depending on the selectivity of operators. We demonstrate this on a system of two similar 'Peak Detection' workflows. The 'Peak Detection' workflow is described in the motivating example (Sec. 2.1). The workflows of this workload differ only in the vertex *filter_region*.

Experiment 4.1: Effect of operators selectivity

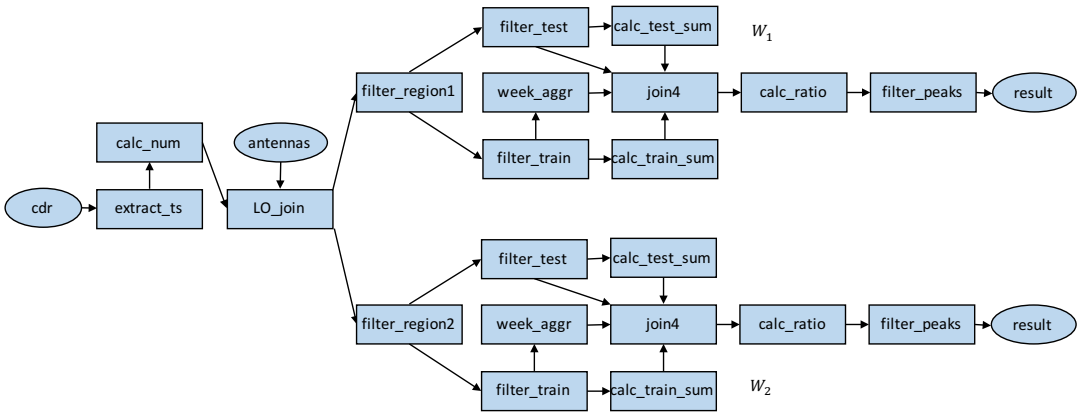


Figure 54: An optimal joint workflow of two 'Peak Detection' workflows in case if total selectivity of *filter_region1* and *filter_region2* is low

In this experiment, we have two workflows W_1 and W_2. Vertices *filter_region1* and *filter_region2* limit different regions of data analysis. There is a total 12 regions in the input dataset *CDR*. In the first run, both workflows limit their analyzed area in 4 regions (the regions may be different for the two workflows). Figure 53 displays the result of multi-workflow optimization for these workflows. This joint workflow is a combination of the two initially optimized workflows (see the result of experiment 1.1) with respect to a small common part, that consists of one vertex *LO_join*. In the second run, the workflows limit their analyzed area in 8 regions (the regions may be different for the two workflows). The join workflow produced by MWO is shown in Figure 54. This result differs from the one in the first run because of the different selectivity of *filter_region1* and *filter_region2*. The sum of the estimated costs of vertices *extract_ts* and *calc_num* of both workflows after filtering by regions, in the first run, is smaller than the cost of executing the same vertices on the unfiltered data. In the second run, *extract_ts* and *calc_num* are swapped with *filter_region1* and *filter_region2*, and, therefore, the common part is bigger and includes vertices *extract_ts* and *calc_num*. MWO also considers three single-vertex common parts in the middle of the workflows: *filter_test*, *filter_train* and *filter_peaks*. However, the split-merge combination only increases the cost of produced workflow version, because the costs of these common parts (vertices) are very low.

The original, the optimized and joint workflows are executed on 10M rows CDR dataset. In the first run, the joint optimized workflow (Figure 53) has an execution time that is 3218 ms less, (2.3% time gain), compared to the sum of execution times of the optimized versions of the two initial workflows. In the second run, the joint optimized workflow (Figure 53) has an execution time that is 9512 ms less (6.0% time gain), compared to the sum of the execution times of the optimized versions of the two initial workflows..

7 Related Work

Most Workflow Management Systems (WMS) are described in detail in the reports D5.1 [29] and D5.2 [28]. In this chapter we overview some of them and discuss works that engage in optimization.

Multi engine systems Many systems have the ability to generate an execution plan from a logical. Here, Garlic [22], and TSIMMIS [5] were early prototypes of such systems. However, no systems are able to generate an execution plan for a variety of engines. Some systems can generate code within a particular family of engines. For example, some database query systems can generate SQL code for a variety of database systems (e.g., PostgreSQL, MariaDB [18]). Some systems have the ability to generate execution plan for more than one processing engine, for example BigDAWG Polystore System [9], that focuses on integrating independent storage engines. BigDAWG offers users *location transparency*, so that application programmers do not need to understand the details about the underlying database(s) that will execute their queries. This has been implemented using *islands of information*. Each island is a front-facing abstraction for the user, and it includes a query language, data model, and a set of connectors or shims for interacting with the underlying storage engines. Our system considers a wider configuration of multi-engine platform, besides a set of datastores, there are many execution engines, such as a MapReduce engine or a scripting engine. Moreover, our model of logical workflow is more flexible and simple: to start using a new engine in BIGDAWG requires to define an *island* for it, in our system user provides implementations in that engine of needed logical operators.

Many Extract-Transform-Load (ETL) systems (e.g., [14], [19]) can generate execution plan to, for example, filter and extract data from several database systems and then import and process that data into there native ETL engine. However, these systems are inflexible in that the data processing is can be performed only on there internal ETL engine. Our system can generate a variety of execution plans from a single logical plan, e.g., given a logical plan, we might create one execution plan that filters a data set in a database system, while another plan might filter that data set using a scripting engine.

Workflow optimization There are many workflow management systems, e.g. Taverna [25] and Pegasus [20]. Taverna is an open source domain-independent workflow management system, which includes a suite of tools used to design and execute scientific workflows. It is most widely used in bioinformatics. Taverna is focused on the issue of the analysis of data from heterogeneous and 'incompatible' sources and does not provide any sophisticated methods for workflow optimisation. Pegasus is another workflow management system that allows users to easily express multi-step computational tasks. Pegasus provide several optimization approaches, such as Data Reuse, Task Clustering, Just In Time Planning etc. These approaches speed up the execution of a single execution plan. In contrast to this, our optimisation technique finds several execution plans for a logical workflow and chooses an optimal one.

Workflow optimization is a relatively new field of research, but there are already

some promising results. Commercial ETL tools (e.g. [14], [19]) provide little support for automatic optimization. They provide hooks for the ETL designer to specify for example which flows may run in parallel or where to partition flows for pipeline parallelism. Some ETL engines such as PowerCenter [14] support PushDown optimization, which pushes operators that can be expressed in SQL from the ETL flow down to the source or target database engine. The rest of the transformations are executed in the data integration server. The challenge of optimizing the entire workflow remains.

Towards this direction, HFMS [23] performs optimization and execution across multiple engines. Work related to HFMS [24] focuses on optimizing flows for several objectives: performance, fault-tolerance and freshness over multiple execution engines. HFMS uses many optimization strategies, such as parallelization, recovery points, function shipping, data shipping, decomposition, etc. Complementary to the above, our work provides detailed optimization algorithm, that uses more general workflow graph reconfiguration that takes into account the semantics of the operators. This semantics include operators categorization and prepared decomposition of complex library operators into a set of operators.

Another system, one of the newest, that provides an abstraction on top of existing data processing platforms is called Rheem [21]. It provides multi-platform data analytics applications execution and optimization. But at the moment, there is not so much information has been published about it.

Let us note, that all these works devoted to a single-workflow optimization. In this report, in addition to a single-workflow we propose a thorough technique for a multi-workflow optimization.

Query optimization Some ETL engines provide limited performance optimization techniques such as pushdown of relational operators [13]. Query optimization focuses on performance and considers a subset of operators typically encountered in our case. Also, we want the optimizer to be independent of the execution engine; in fact, we want to allow the optimized workflow to be execute on more than one engine. Research on federated database systems has considered query optimization for multiple execution engines, but this work was limited to traditional query operators (e.g., see Garlic [22], Pegasus [7], and in papers [8] [22]).

8 Summary

This document describes a thorough technique for the optimization of analytics workflows defined on multi-engine systems. This includes single and multi workflow optimization. Further, we provide a benchmark suited for the problem of experimenting with a broad range of workflows. It provides a principled way for constructing workflows. We propose a categorization of workflow structures, which covers frequent design cases. Finally, we evaluate described optimization algorithms on a set of experiment sets, that include both synthetic workflows, generated in proposed benchmarking tool, and workflows of real-world applications.

References

[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] Alexander Alexandrov, Andreas Salzmann, Georgi Krastev, Asterios Katsifodimos, and Volker Markl. Emma in action: Declarative dataflows for scalable data analysis. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2073–2076, New York, NY, USA, 2016. ACM.

[3] Adaptable scalable analytics platform. `http://www.asap-fp7.eu/`.

[4] R. Bertoldi. Wp 9 - applications: Telecommunication data analytics. D9.2 Use Case Requirements, February 2015.

[5] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, 1994.

[6] Ewa Deelman, Christopher Carothers, Anirban Mandal, Brian Tierney, Jeffrey S. Vetter, Ilya Baldin, Claris Castillo, Gideon Juve, Dariusz Krol, Vickie Lynch, Ben Mayer, Jeremy Meredith, Thomas Proffen, Paul Ruth, and Rafael Ferreira da Silva. Panorama: Modeling the performance of scientific workflows. In *Workshop on Modeling and Simulation of Systems and Applications (ModSim 2015)*, 2015. Funding Acknowledgements: DOE DE-SC0012636.

[7] 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.

[8] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query optimization in a heterogeneous dbms. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 277–291, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[9] Aaron J. Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Çetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, Samuel Madden, David Maier, Timothy G. Mattson, S. Papadopoulos, Jeff Parkhurst, Nesime Tatbul, Manasi Vartak, and Stan Zdonik. A demonstration of the bigdawg polystore system. *PVLDB*, 8(12):1908–1911, 2015.

[10] Apache flink. http://flink.apache.org/.

[11] Measuring the roi of the nintex workflow platform. http://info.nintex.com/TEI-Report-10-2014-LP.html.

[12] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1207–1210, 2009.

[13] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, August 2008.

[14] Informatica "powercenter". http://www.informatica.com/products/powercenter/.

[15] Ibm infosphere information server. http://ibm.com/software/data/integration/info_server/.

[16] D. Tsoumakos K. Doka, N. Papailiou et al. Wp 3 - intelligent, multi-engine resource scheduling platform. D3.2 IReS Platform v.1, August 2015.

[17] Anirban Mandal, Paul Ruth, Ilya Baldin, Dariusz Krol, Gideon Juve, Rajiv Mayani, Rafael Ferreira da Silva, Ewa Deelman, Jeremy Meredith, Jeffrey Vetter, Vickie Lynch, Ben Mayer, James Wynne III, Mark Blanco, Chris Carothers, Justin LaPre, and Brian Tierney. Toward and end-to-end framework for modeling, monitoring, and anomaly detection for scientific workflows. In *Workshop on Large-Scale Parallel Processing (LSPP 2016)*, pages 1370–1379, 2016. Funding Acknowledgments: DOE DE-SC0012636.

[18] Mariadb. https://mariadb.org/.

[19] Oracle warehouse builder 10g. http://www.oracle.com/technology/products/warehouse/.

[20] Pegasus. http://pegasus.isi.edu/.

[21] Rheem. http://da.qcri.org/rheem/about.html.

[22] M. Tork Roth, M. Arya, L. Haas, M. Carey, W. Cody, R. Fagin, P. Schwarz, J. Thomas, and E. Wimmers. The garlic project. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 557–, New York, NY, USA, 1996. ACM.

[23] 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.

[24] 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.

[25] Taverna. `http://www.taverna.org.uk/`.

[26] Christian Thomsen, Torben Bach Pedersen, and Wolfgang Lehner. Rite: Providing on-demand data for right-time data warehousing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 456–465, 2008.

[27] Tpc-ds: The benchmark standard for decision support solutions including big data. `http://www.tpc.org/tpcds/`.

[28] M. Filatov V. Kantere. Wp 5 - adaptive data analytics. D5.2 Workflow management tool, August 2015.

[29] M. Filatov V. Kantere. Wp 5 - adaptive data analytics. D5.1 Workflow Management Model, February 2015.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D5.3
Data Processing Deployment

WP 5 – Adaptive Data Analytic

Nature: Report

Dissemination: Public